Systemprogrammierung

Grundlagen von Betriebssystemen

Teil A – III. Vom C-Programm zum laufenden Prozess

---

9. Mai 2023

Jürgen Kleinöder

Überblick

- Vom C-Programm zum ausführbaren Programm (*Executable*)
  - Präprozessor
  - Compilieren
  - (Assemblieren)
  - Binden (statisch / dynamisch)
- Programme und Prozesse
  - Speicherorganisation eines Programms
  - Speicherorganisation eines Prozesses
  - Laden eines Programms (statisch gebunden / dynamisch gebunden)
- Prozesse
  - Prozesszustände
  - Prozesse erzeugen
  - Programme ausführen
  - weitere Operationen auf Prozessen

Übersetzen - Objektmodule

- 1. Schritt: Präprozessor
  - entfernt Kommentare, wertet Präprozessoranweisungen aus
    - ➤ fügt include-Dateien ein
    - ➤ expandiert Makros
    - ➤ entfernt Makro-abhängige Code-Abschnitte (*conditional code*) Beispiel:

```
#define DEBUG
...
#ifdef DEBUG
    printf("Zwischenergebnis = %d\n", wert);
#endif DEBUG
```

  - Zwischenergebnis kann mit `cc -P datei.c` als `datei.i` erzeugt werden oder mit `cc -E datei.c` ausgegeben werden

Übersetzen - Objektmodule (2)

- 2. Schritt: Compilieren
  - übersetzt C-Code in Assembler
  - wenn Assemblercode nicht explizit angefordert wird, direkter Übergang zu 3.
  - Zwischenergebnis kann mit `cc -S datei.c` als `datei.s` erzeugt werden
- 3. Schritt: Assemblieren
  - assembliert Assembler-Code, erzeugt Maschinencode (Objekt-Datei)
  - standardisiertes Objekt-Dateifomat: ELF (Executable and Linking Format) (vereinfachte Darstellung) - in nicht-UNIX-Systemen andere Formate
    - ➤ Maschinencode
    - ➤ Informationen über Variablen mit Lebensdauer *static* (ggf. Initialisierungswerte)
    - ➤ Symboltabelle: wo stehen welche globale Variablen und Funktionen
    - ➤ Relokierungsinformation: wo werden welche "nicht gefundenen" globalen Variablen bzw. Funktionen referenziert
  - Zwischenergebnis kann mit `cc -c datei.c` als `datei.o` erzeugt werden

Binden und Bibliotheken

- 4. Schritt: Binden
  - Programm **`ld`** : (*linker*), erzeugt ausführbare Datei (*executable file*)
    - ➤ ebenfalls ELF-Format (früher a.out-Format oder COFF)
  - Objekt-Dateien (.o-Dateien) werden zusammengebunden
    - ➤ noch nicht abgesättigte Referenzen auf globale Variablen und Funktionen in anderen Objekt-Dateien werden gebunden (Relokation)
  - nach fehlenden Funktionen wird in Bibliotheken gesucht

Binden und Bibliotheken (2)

- statisch binden
  - alle fehlenden Funktionen werden aus Bibliotheken genommen und in die ausführbare Datei einkopiert
    - ausführbare Datei ggf. sehr groß
    - Funktionen die in vielen Programmen benötigt werden (z. B. printf) werden überall einkopiert
- dynamisch binden
  - Funktionen aus gemeinsam nutzbaren Bibliotheken (*shared libraries*) werden nicht in die ausführbare Datei einkopiert
    - ausführbare Datei enthält weiterhin nicht-relokierte Referenzen
    - ausführbare Dateien sind kleiner, mehrfach genutzte Funktionen sind nur einmal in der shared library abgelegt
    - Relokation erfolgt beim Laden

Programme und Prozesse

- **Programm:** Folge von Anweisungen
  (hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)

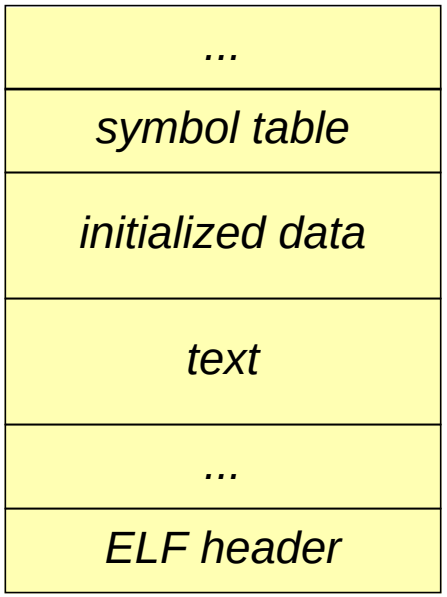
- **Prozess:** Programm, das sich in Ausführung befindet, und seine Daten
  (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - ein Prozess ist erst mal ein **abstraktes Gebilde** (= Funktionen und Datenstrukturen zur Verwaltung von Programmausführungen)
  - im objektorientierten Sinn eine *Klasse*

- **Prozessinstanz** (Prozessinkarnation):
  eine physische Instanz des abstrakten Gebildes "Prozess"
  - eine konkrete Ausführungsumgebung für ein Programm (Speicher, Rechte, Verwaltungsinformation)
  - im objektorientierten Sinn die *Instanz*

- Sprachgebrauch in der Praxis etwas schlampig:
  mit "Prozess" wird meistens eine Prozessinstanz gemeint

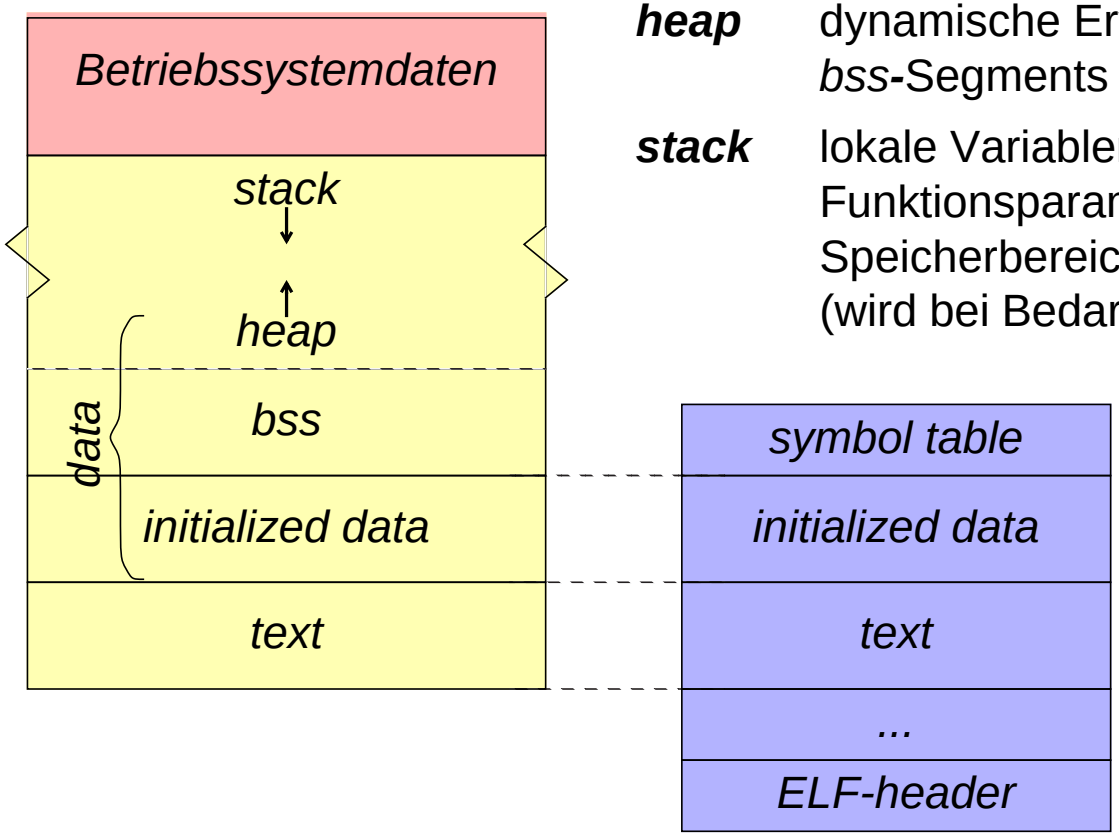
Speicherorganisation eines Programms

- definiert durch das ELF-Format
- wichtigste Elemente (stark vereinfach dargestellt)



ELF header	Identifikator und Verwaltungsinformationen (z. B. verschiedene *executable* Formate möglich)
text	Programmkode
initialized data	initialisierte globale und *static* Variablen
symbol table	Zuordnung der im Programm verwendeten symbolischen Namen von Funktionen und globalen Variablen zu Adressen (z. B. für Debugger)

Speicherorganisation eines Prozesses



bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (***sbrk(2)***, ***malloc(3)***)

stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

Laden eines Programms

- in eine konkrete Ausführungsumgebung (Prozessinstanz) kann ein Programm geladen werden
  - ➤ Loader
- Laden statisch gebundener Programme
  - Segmente der ausführbaren Datei werden in den Speicher geladen
    - ➤ abhängig von der jeweiligen Speicherorganisation des Betriebssystems
  - Speicher für nicht-initialisierte globale und *static* Variablen (bss) wird bereitgestellt und mit 0 vorbelegt
  - Speicher für lokale Variablen (stack) wird bereitgestellt
  - Aufrufparameter werden in Stack- oder Datensegment kopiert, argc und argv-Zeiger werden entsprechend initialisiert
  - main-Funktion wird angesprungen

Laden eines Programms (2)

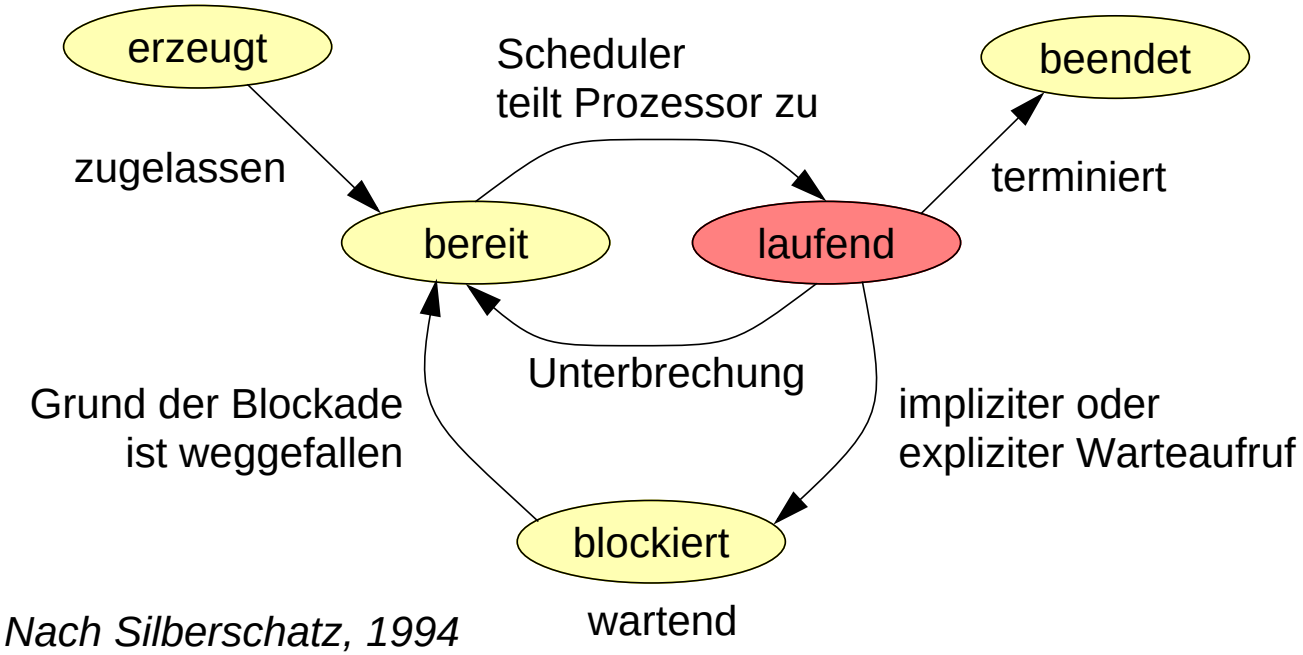
- Laden dynamisch gebundener Programme
  - spezielles Lade-Programm wird gestartet: **`ld.so`** (*dynamic linker/loader*)
    ld.so erledigt die weiteren Aufgaben
    - Segmente der ausführbaren Datei werden in den Speicher geladen und Speicher für nicht-initialisierte globale und *static* Variablen (bss) wird angelegt
    - fehlende Funktionen werden aus shared libraries geladen (ggf. rekursiv)
    - noch offene Referenzen werden abgesättigt (Relokation)
    - wenn notwendig werden Initialisierungsfunktionen der shared libraries aufgerufen (z. B. Klasseninitialisierungen bei C++)
    - Parameter für main werden bereigestellt
    - main-Funktion wird angesprungen
    - bei Bedarf können auch während der Laufzeit des Programms auf Anforderung des Programms weitere Funktionen nachgeladen werden (z. B. für plugins)

Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** *(New)*
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** *(Ready)*
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** *(Running)*
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** *(Blocked/Waiting)*
    Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - **Beendet** *(Terminated)*
    Prozess ist beendet; einige Betriebsmittel sind aber noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

Prozesszustände (2)

- Zustandsdiagramm



Nach Silberschatz, 1994

- Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt.

Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Elternprozess

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Elternprozess

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kindprozess

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Prozesserzeugung (2)

- Der Kindprozess ist eine perfekte Kopie des Elternprozesses
  - gleiches Programm
  - gleiche Daten (gleiche Werte in Variablen)
  - gleicher Programmzähler (nach der Kopie)
  - gleicher Eigentümer
  - gleiches aktuelles Verzeichnis
  - gleiche Dateien geöffnet (selbst Schreib-/Lesezeiger ist gemeinsam)
  - ...
- Unterschiede:
  - verschiedene PIDs
  - **`fork()`** liefert verschiedene Werte als Ergebnis für Eltern- und Kindproz.

Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execve( const char *path, char *const argv[],
                    char *const envp[] );
```

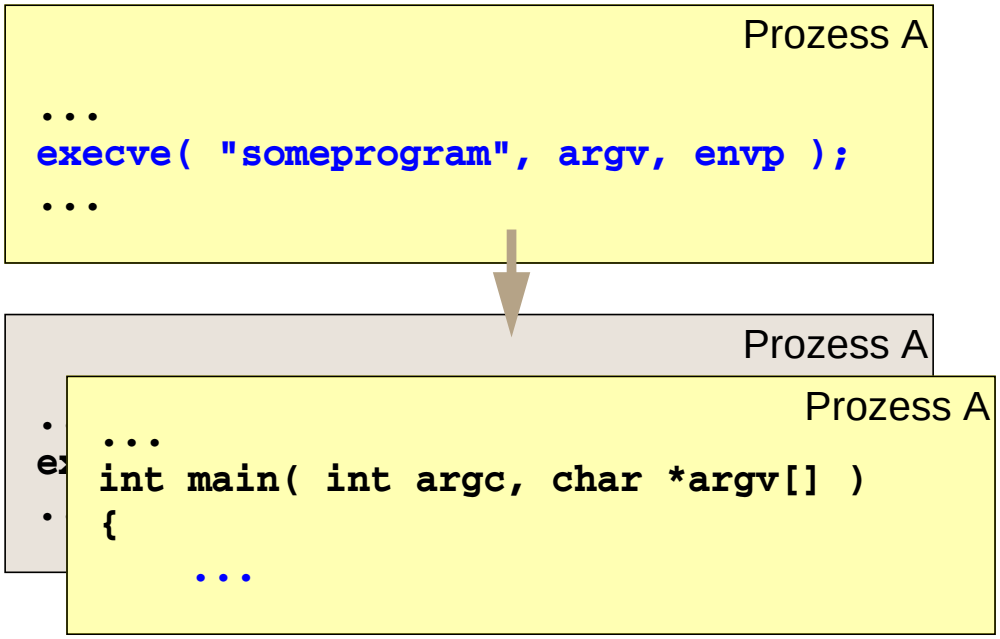
Prozess A

```
...
execve( "someprogram", argv, envp );
...
```

Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execve( const char *path, char *const argv[],
                    char *const envp[] );
```



das vorher ausgeführte Programm ist dadurch endgültig beendet

➤ execve kehrt im Erfolgsfall nie zurück

Operationen auf Prozessen (UNIX)

- Prozess beenden

```
void exit( int status );
```

  - ➤ Prozess terminiert - exit kehrt nicht zurück

- Prozessidentifikator

```
pid_t getpid( void );                    /* eigene PID */
pid_t getppid( void );                   /* PID des Elternprozesses */
```

- Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```

  - ➤ Prozess wird so lange blockiert bis Kindprozess terminiert
  - ➤ über den Parameter werden Informationen über den exit-Status des Kindprozesses zurückgeliefert

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – IV. Einleitung

15. Mai 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

- Einordnung
- Fallstudie
  - Speicherbelegung
  - Analyse
- Begriffsdeutung
  - Literaturauszüge
- Zusammenfassung

Gliederung

Systemprogrammierung $\rightsquigarrow$ Betriebssysteme

- Infrastruktursoftware für Rechensysteme
  - was Betriebssysteme sind, hat schon „Glaubenskriege“ hervorgerufen
    - das Spektrum reicht von „Winzlingen“ bis hin zu „Riesen“
    - simple Prozeduren $\Leftrightarrow$ komplexe Programmsysteme
  - entscheidend ist, dass Betriebssysteme nie dem Selbstzweck dienen
- jedes Rechensystem wird durch ein Betriebssystem betrieben
  - Ausnahmen bestätigen die Regel…





„Sein oder Nichtsein, das ist hier die Frage“

- Betriebssysteme sind **Handwerkszeug** der Informatik
  - mit dem umzugehen ist zur Benutzung eines Rechensystems
  - das gelegentlich zu beherrschen, anzupassen und auch anzufertigen ist

- IBM: z/VM (vormals VM/CMS), z/OS

- DEC: VAX/VMS

- DOS (16-/32-Bit)-, NT- und CE-Linie

- 





[1] Shakespeare zur Unabänderlichkeit oder Beeinflussbarkeit des Schicksals.

Variantenvielfalt

„Kleinvieh macht Mist“

- funktionale und nichtfunk. Eigenschaften von Betriebssystemen werden durch die **Anwendungsdomäne** vorgegeben
  - gelegentlich passen bestehende „unspezifische“ Lösungen (z.B. Linux)
  - viel häufiger sind jedoch **anwendungsspezifische Lösungen** erforderlich
- Systeme für den Allgemeinzweck
  - Rechensysteme für die gängigsten Aufgaben einer Anwendungsdomäne
  - *die* Domäne der umseitig (S. 5) genannten Vertreter
- Systeme für den Spezialzweck
  - Rechensysteme zur Steuerung oder Regelung „externer“ Prozesse
  - für gewöhnlich mit vorhersagbarem Laufzeitverhalten (**Echtzeitsysteme**)

↪ **eingebettete Systeme** (vgl. [11])

- (das „Smartphone“ nicht mitgerechnet)

Gliederung

Fallstudie

Speicherbelegung

Untersuchung eines Phänomens

Ganzheitsmethode[2]

Die Funktionsweise (auch) von Betriebssystemen zu verstehen, hilft bemerkenswerte Erscheinungen innerhalb eines Rechensystems zu begreifen und in ihrer Bedeutung besser einzuschätzen.

- **Eigenschaften** (*features*) von Betriebssystemen erkennen:
  - **funktionale**
    - Verwaltung der Betriebsmittel (Prozessor, Speicher, Peripherie) für eine Anwendungsdomäne
  - **nichtfunktionale**
    - anfallender Zeit-, Speicher-, Energieverbrauch
    - d.h., **Gütemerkmale** einer Implementierung
- aus den funktionalen Eigenschaften resultierendes **Systemverhalten** unterscheiden von Fehlern (*bugs*) des Systems
  - um Fehler kann ggf. „herum programmiert“ werden
  - um zum Anwendungsfall unpassende Eigenschaften oft jedoch nicht

[2] Analytische Lernmethode, die die Vermittlung eines Stoffes als Gesamtheit in den Mittelpunkt stellt, um dann konstituierende Elemente weiter zu untersuchen.

Vorbelegung einer $N \times N$ Matrix

- **zeilenweises Vorgehen**: Spaltenelemente `j` von Zeile `i` aufzählen

```
1  void by_row (int mx[], unsigned int n, int v) {
2      unsigned int i, j;
3      for (i = 0; i < n; i++)
4          for (j = 0; j < n; j++)
5              mx[i * n + j] = v;        /* "mx[i][j]" = v */
6  }
```

- **spaltenweises Vorgehen**: Zeilenelemente `i` von Spalte `j` aufzählen

```
7  void by_column (int mx[], unsigned int n, int v) {
8      unsigned int i, j;
9      for (j = 0; j < n; j++)
10         for (i = 0; i < n; i++)
11             mx[i * n + j] = v;        /* "mx[i][j]" = v */
12 }
```

Gemeinsamkeit und Unterschied (von vertauschten Zeilen abgesehen)

funktional erfüllen beide Varianten denselben Zweck

nichtfunktional unterscheiden sie sich ggf. im Laufzeitverhalten

[3] Erst zur Laufzeit bekannte Feldgrenzen (vgl. S. 40). Beachte: `mx[]` $\equiv$ `*mx`.

Instanzenbildung und Initialisierung der Matrix

```
13  #include <stdlib.h>
14
15  main (int argc, char *argv[]) {
16      if (argc == 3) {
17          unsigned int n = atol(argv[2]);
18          if (n != 0) {
19              int *mx = (int*)calloc(n*n, sizeof(int));
20              if (mx != 0) {
21                  if (*argv[1] == 'R') by_row(mx, n, 42);
22                  else by_column(mx, n, 42);
23                  free(mx);
24              }
25          }
26      }
27  }
```

- **16** Verwendung: `<name> <way> <count>`
- **17** Größe einer Zeile bzw. Spalte einlesen
- **19** Matrixspeicher anfordern und löschen
- **21–22** zeilen-/spaltenweise Initialisierung mit 42
- **23** Matrixspeicher der Halde zurückgeben

Laufzeitverhalten[4]

$N \times N$ **Matrix, mit** $N = 11\,174 \approx$ **500 MB**

Betriebssystem	Zentraleinheit	Speicher	`by_row()`	`by_column()`	

Solaris	2 × 1 GHz UltraSPARC IIIi	8 GB	`3.64r`
			`2.09u`
			`1.11s`
				`27.07r`
				`24.68u`
				`1.10s`
					(a)

Windows XP (Cygwin)	2 × 3 GHz Pentium 4 XEON	4 GB	`0.87r`
			`0.65u`
			`0.21s`
				`11.94r`
				`11.62u`
				`0.21s`
					(b)

Linux 2.6.20	2 × 3 GHz Pentium 4 XEON	4 GB	`0.89r`
			`0.48u`
			`0.40s`
				`14.73r`
				`14.34u`
				`0.39s`
					(c)

	2 × 2.8 GHz Pentium 4	512 MB	`51.23r`
			`0.47u`
			`2.17s`
				`39.84r`
				`14.34u`
				`2.09s`
					(d)

Mac OS X 10.4	1.25 GHz PowerPC G4	512 MB	`10.24r`
			`0.69u`
			`2.12s`
				`106.72r`
				`23.15u`
				`17.08s`
					(e)

	1.5 GHz PowerPC G4	512 MB	`10.11r`
			`0.46u`
			`2.08s`
				`93.68r`
				`23.71u`
				`6.85s`
					(f)

		1.25 GB	`2.17r`
			`0.43u`
			`1.50s`
				`27.95r`
				`22.35u`
				`4.22s`
					(g)

[4] `time ./main x 11174`, mit `x` ∈ (`R`, `C`)

Fallstudie

Analyse

Wo uns der Schuh drückt…

(a)–(g) Linearisierung AuD, GRA

- zweidimensionales Feld $\mapsto$ eindimensionaler Arbeitsspeicher

(a)–(g) Zwischenspeicher (*cache*) GRA

- Zugriffsfehler (*cache miss*), Referenzfolgen

(d) Kompilierer & Magie UEB

- semantische Analyse, Erkennung gleicher Zugriffsmuster

(d)–(f) virtueller Speicher SP

- Seitenfehler (*pagefault*), Referenzfolgen

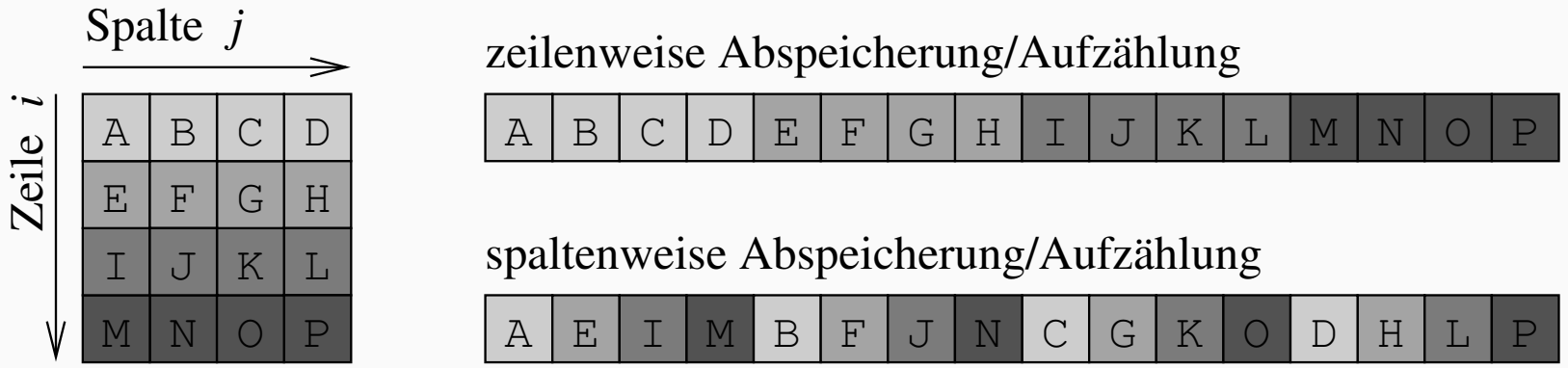
(e)–(g) Betriebssystemarchitektur SP

- Verortung der Funktion zur Seitenfehlerbehandlung

Sir Isaac Newton

Was wir wissen, ist ein Tropfen, was wir nicht wissen, ist ein Ozean.

Linearisierung I



- im Abstrakten könnte uns diese **Abbildung** egal sein
- im Konkreten jedoch nicht

Abspeicherung	Aufzählung	
	zeilenweise	spaltenweise
zeilenweise	☺	☹
spaltenweise	☹	☺

Fälle (a)–(g)

- Abspeicherung zeilenweise (C bzw. `main()`)
- Aufzählung zeilen- (`by_row()`) und spaltenweise (`by_column()`)

Entwicklung der Adresswerte A beim Zugriff auf die Elemente einer zeilenweise abgespeicherten Matrix mit Anfangsadresse γ:

$i \setminus j$	0	1	2	$\cdots$	$N-1$
0	0	1	2	$\cdots$	$N-1$
1	$N+0$	$N+1$	$N+2$	$\cdots$	$N+N-1$
2	$2N+0$	$2N+1$	$2N+2$	$\cdots$	$2N+N-1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$N-1$	$(N-1)N+0$	$(N-1)N+1$	$(N-1)N+2$	$\cdots$	$(N-1)N+N-1$

- $A = \gamma + (i * N + j) * \texttt{sizeof(int)}$, für $i, j = 0, 1, 2, \ldots, N-1$

Fälle (a)–(g)

- linear im homogenen Fall (`by_row()`), Abspeicherung und Aufzählung sind gleichförmig $\rightsquigarrow$ **starke Lokalität**
- sprunghaft, sonst (`by_column()`) $\rightsquigarrow$ schwache Lokalität

Zwischenspeicher (*cache*)

- **Normalfall**: Datum befindet sich im Zwischenspeicher
  - Zugriffszeit $\approx$ Zykluszeit der CPU, Wartezeit $= 0$
- **Ausnahmefall**: Datum befindet sich *nicht* im Zwischenspeicher

$\hookrightarrow$ Zugriffsfehler $\rightsquigarrow$ Einlagerung (Zwischenspeicherzeile, *cache line*)

  - Zugriffszeit $\geq$ Zykluszeit des RAM, Wartezeit > 0
  - **schlimmster Fall** (*worst case*): Zwischenspeicher ist voll $\mapsto$ **GAU**

    $\hookrightarrow$ Zugriffsfehler $\rightsquigarrow$ Ein- und ggf. Auslagerung (Zwischenspeicherzeile)

    – Zugriffszeit $\geq 2\times$Zykluszeit des RAM, Wartezeit $\gg 0$
- die Effektivität steht und fällt mit der **Lokalität** der Einzelzugriffe
  - starke Lokalität erhöht die **Trefferwahrscheinlichkeit** erheblich

Fälle (a)–(g)

- beide Varianten verursachen bei Ausführung den GAU
- `by_column()` $\rightsquigarrow$ schwache Lokalität: schlechte Trefferquote

- **funktionale Eigenschaft** $\mapsto$ „was“ etwas tut
  - beide Varianten tun das gleiche — nur in verschiedener Weise
- **nichtfunktionale Eigenschaft** $\mapsto$ „wie“ sich etwas ausprägt
  - `by_row()` zählt Feldelemente entsprechend Feldabspeicherung auf
  - `by_row()` zeigt für gegebene Hardware günstigere Zugriffsmuster
  - ⋮
  - `by_row()` wird schneller als `by_column()` ablaufen können

Fall (d): Beispiel eines wahren Mysteriums…

- die Übersetzung[a] zeigte identischen Assembliersprachenkode
  - `by_column()` ist Kopie von `by_row()`
  - statische Analyse sagt gleiches Verhalten beider Varianten voraus
- Experiment brachte Messreihen mit extremen Ausschlägen hervor
  - „dynamische Umgebung“ verhält sich zugunsten von `by_column()`

[a] „`gcc -O -m32 -fomit-frame-pointer -fno-pic -static -S`“ hier und im weiteren Vorlesungsverlauf, soweit nicht anders angegeben.

Arbeitsspeicher hat mehr Kapazität als der Hauptspeicher[a]

[a] Begrifflich sind Arbeits- und Hauptspeicher verschiedene Dinge!

- Hauptspeicher ist Zwischenspeicher $\mapsto$ **Vordergrundspeicher**
  - von Programm- bzw. Adressraumteilen eines oder mehrerer Prozesse
- ungenutzte Bestände im Massenspeicher $\mapsto$ **Hintergrundspeicher**
  - z.B. Plattenspeicher, SSD oder gar Hauptspeicher anderer Rechner
- gleiches Problem wie beim Zwischenspeicher (vgl. S. 17)
  - **Seitenumlagerung** (*paging*) $\rightsquigarrow$ zeitaufwendige Ein-/Ausgabevorgänge
  - Zugriffszeit verlangsamt sich um einige Größenordnungen: ns $\rightsquigarrow$ ms

Fälle (a)–(g): Zwickmühle wegen Hauptspeicherkapazität...

(d)–(f) beide Varianten verursachen den GAU (S. 17)

- kontraproduktiver Seitenvorabruf (*prepaging*) SP2

sonst fallen „nur“ Einlagerungsvorgänge an

- Prozess zieht sein Programm selbst in den Hauptspeicher

Betriebssystemarchitektur

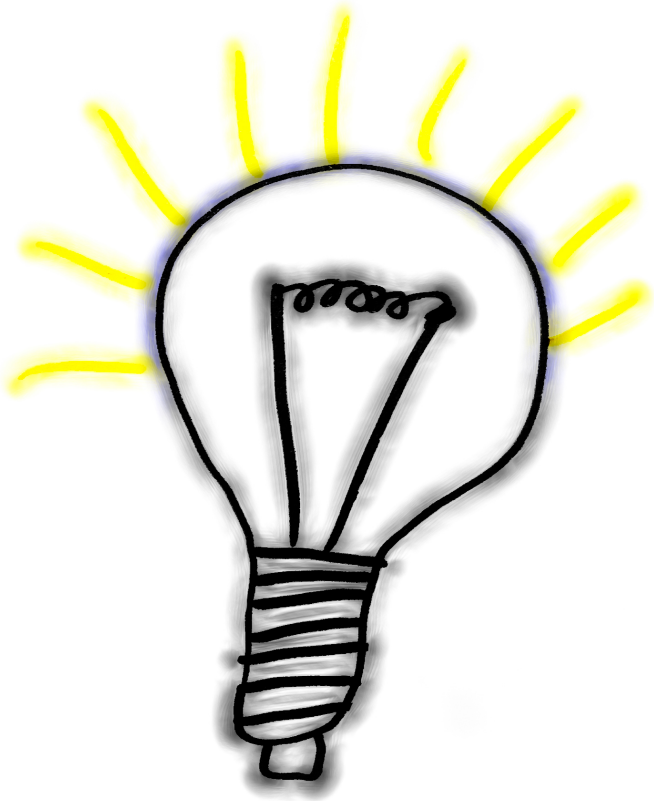
Schönheit, Stabilität, Nützlichkeit — Venustas, Firmitas, Utilitas: Die drei Prinzipien von Architektur [7]

- Systemfunktionen sind architektonisch verschieden ausgeprägt
  - sie teilen sich dieselben Domänen oder eben auch nicht
  - bzgl. Adressraum, Ausführungsstrang, Prozessor oder Rechnersystem
- architektonische und funktionale Merkmale widersprechen sich nicht
  - beide Arten bewirken jedoch gewisse **nichtfunktionale Eigenschaften**
    - z.B. verursachen domänenübergreifende Aktionen ggf. Mehraufwand

Fälle (e)–(g)

- Mac OS X = NeXTStep ∪ Mach 2.5 ⇝ **mikrokernbasiert**
  - Systemfunktionen laufen als *Tasks* in eigenen Adressräumen ab
  - Tasks bieten Betriebsmittel für ggf. mehrere Ausführungsstränge
  - Ausführungsstränge sind die Zuteilungseinheiten für Prozessoren
- **externer Seitenabrufer** (*external pager*) zur Seitenumlagerung
  - außerhalb des klassischen Kerns ⇝ domänenübergreifende Aktionen

„…und denen […] ist ein Licht aufgegangen“ [5]



[5] Matthäus 4.16

Gliederung

Phänomen hin oder her…

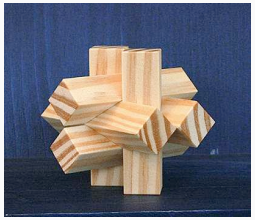
Was macht ein Betriebssystem [aus] ?

Begriffsdeutung

Literaturauszüge

Nachschlagewerke

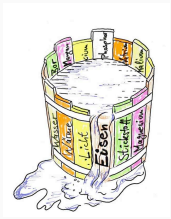
Summe derjenigen Programme, die als residenter Teil einer EDV-Anlage für den Betrieb der Anlage und für die Ausführung der Anwenderprogramme erforderlich ist. [8]

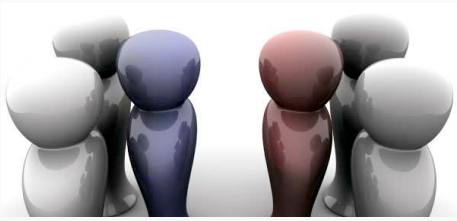


Be'triebs·sys·tem *<n.; -s, -e; EDV> Programmbündel, das die Bedienung eines Computers ermöglicht. [12]*

Lehrbücher I

Der Zweck eines Betriebssystems [besteht] in der Verteilung von Betriebsmitteln auf sich bewerbende Benutzer. [4]



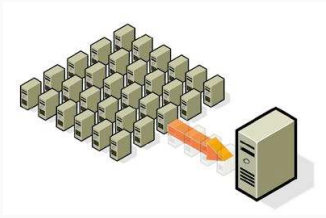



Eine Menge von Programmen, die die Ausführung von Benutzerprogrammen und die Benutzung von Betriebsmitteln steuern. [3]

Lehrbücher II

Eine Softwareschicht, die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine virtuelle Maschine anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware]. [10]




Ein Programm, das als Vermittler zwischen Rechnernutzer und Rechnerhardware fungiert. Der Sinn des Betriebssystems ist eine Umgebung bereitzustellen, in der Benutzer bequem und effizient Programme ausführen können. [9]

Philosophische Lektüre

The operating system is itself a programm which has the functions of shielding the bare machine from access by users (thus protecting the system), and also of insulating the programmer from the many extremely intricate and messy problems of reading the program, calling a translator, running the translated program, directing the output to the proper channels at the proper time, and passing control to the next user. [5]

Ein Betriebssystem kennt auf jeden Fall keinen Prozessor mehr, sondern ist neutral gegen ihn, und das war es vorher noch nie. Und auf diese Weise kann man eben jeden beliebigen Prozessor auf jedem beliebigen anderen emulieren, wie das schöne Wort lautet. [6]

Sachbücher und Normen

Es ist das Betriebssystem, das die Kontrolle über das Plastik und Metall (Hardware) übernimmt und anderen Softwareprogrammen (Excel, Word, …) eine standardisierte Arbeitsplattform (Windows, Unix, OS/2) schafft. [2]




Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen. [1]

☺

Gliederung

Resümee

Be'triebs·sys·tem <n.; -s, -e; EDV> (*operating system*)

- eine Menge von Programmen, die
  - Programme, Anwendungen oder BenutzerInnen assistieren sollen
  - die Ausführung von Programmen überwachen und steuern
  - den Rechner für eine Anwendungsklasse betreiben
  - eine abstrakte Maschine implementieren
- verwaltet die Betriebsmittel eines Rechensystems
  - kontrolliert die Vergabe der (Software/Hardware) Ressourcen
  - verteilt diese ggf. gerecht an die mitbenutzenden Rechenprozesse
- definiert sich nicht über die Architektur, sondern über Funktionen

Ausblick

- wir werden…

  SP1
  1. einen „Hauch“ von Rechnerorganisation „einatmen“
  2. Betriebssysteme in ihrer Grobfunktion „von aussen“ betrachten
  3. Rechnerbetriebsarten kennen- und unterscheiden lernen

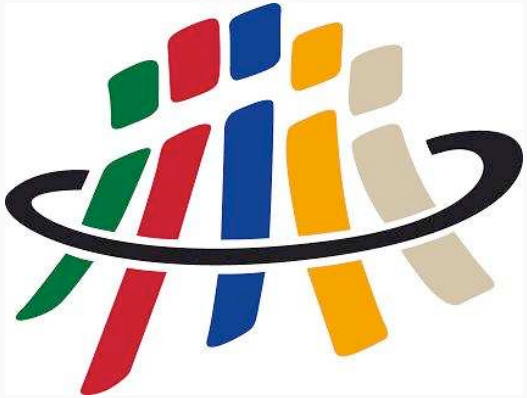
  SP2
  4. eine kurze Zwischenbilanz von SP1 ziehen
  5. Funktionen von Betriebssystemen im Detail untersuchen
  6. den Stoff rekapitulieren

- Zusammenhänge stehen im Vordergrund!
  - Leitfaden ist die ganzheitliche Betrachtung von Systemfunktionen
  - skizziert wird eine logische Struktur ggf. vieler Ausprägungsformen
  - klassischer Lehrbuchstoff wird ergänzt, weniger repetiert oder vertieft

Nachwort

- Typen von Betriebssystemen dürfen nicht dogmatisiert werden
  - etwa: ✯❉■◆❙ „ist besser als“ ✶❉■❄❑◑▲ — umgekehrt dito
  - oder: ★❁❋✯✳ „schlägt beide um Längen“…

- Betriebssysteme sind immer im **Anwendungskontext** zu beurteilen
  - „Universalbetriebssysteme“ gibt es nicht wirklich, wird es nie geben
  - allen Anwendungsfällen wird nie gleich gut Genüge getragen



„Universalbetriebssystem“

„Spezialbetriebssystem“

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Deutsches Institut für Normung:
Informationsverarbeitung — Begriffe.
Berlin, Köln, 1985 (DIN 44300)

[2] Ewert, B. ; Christoffer, K. ; Christoffer, U. ; Ünlü, S. :
FreeHand 10.
Galileo Design, 2001. –
ISBN 3–898–42177–5

[3] Habermann, A. N.:
Introduction to Operating System Design.
Science Research Associates, 1976. –
ISBN 0–574–21075–X

Literaturverzeichnis (2)

[4] Hansen, P. B.:
Betriebssysteme.
Carl Hanser Verlag, 1977. –
ISBN 3–446–12105–6

[5] Hofstadter, D. R.:
Gödel, Escher, Bach: An Eternal Golden Braid — A Metaphorical Fugue on Minds and Machines in the Spirit of Lewis Carrol.
Penguin Books, 1979. –
ISBN 0–140–05579–7

[6] Kittler, F. :
Interview.
`http://www.hydra.umn.edu/kittler/interview.html`, 1993

[7] Pollio, V. M. V.:
De Architectura Libris Decem.
Primus Verlag, 1996 (Original 27 v. Chr.)

Literaturverzeichnis (3)

[8] Schneider, H.-J. :
Lexikon der Informatik und Datenverarbeitung.
München, Wien : Oldenbourg-Verlag, 1997. –
ISBN 3–486–22875–7

[9] Silberschatz, A. ; Galvin, P. B. ; Gagne, G. :
Operating System Concepts.
John Wiley & Sons, Inc., 2001. –
ISBN 0–471–41743–2

[10] Tanenbaum, A. S.:
Operating Systems: Design and Implementation.
Prentice-Hall, Inc., 1997. –
ISBN 0–136–38677–6

[11] Tennenhouse, D. :
Proactive Computing.
In: *Communications of the ACM* 43 (2000), Mai, Nr. 5, S. 43–50

Literaturverzeichnis (4)

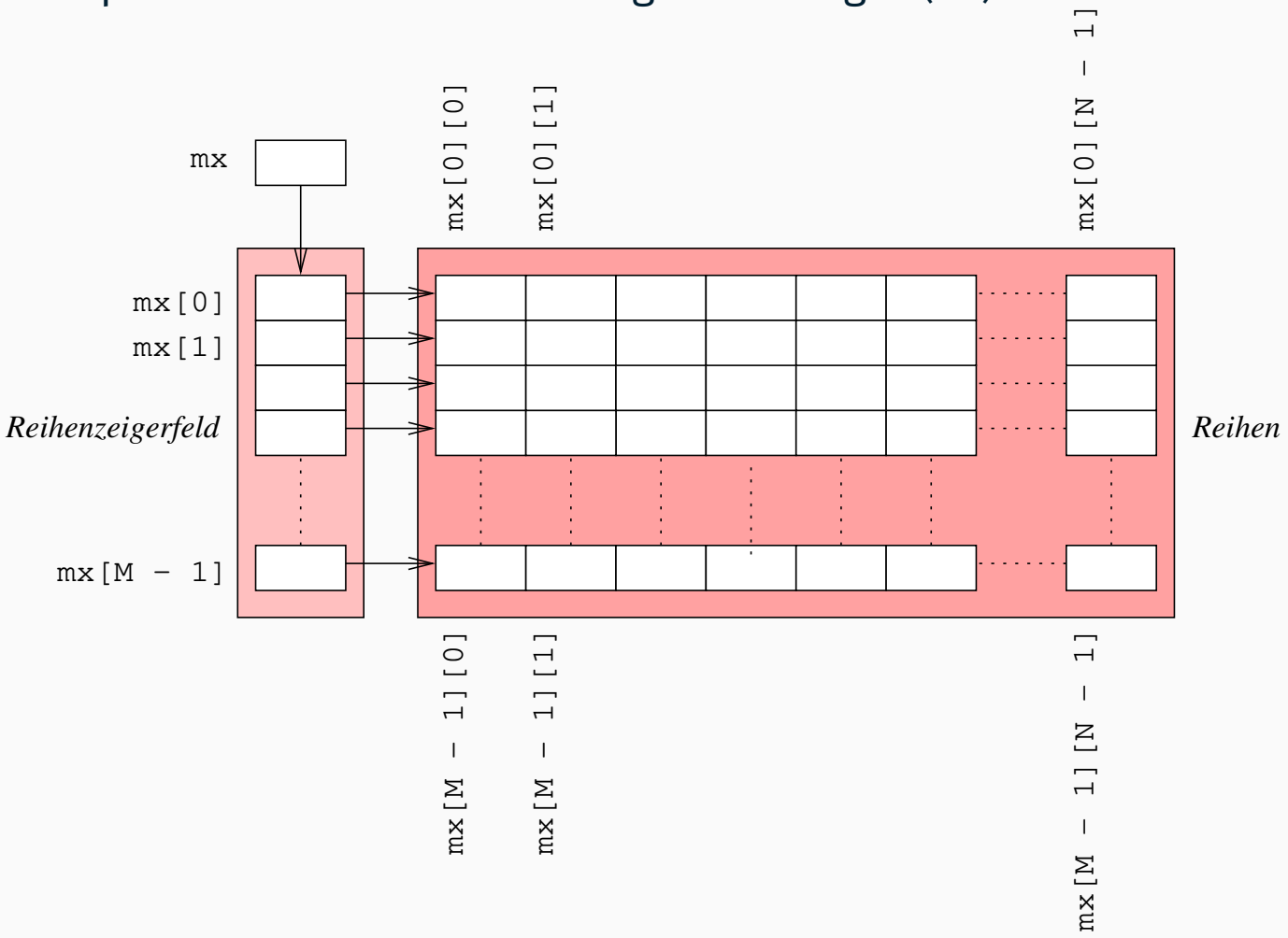
[12] Wahrig-Burfeind, R. :
Universalwörterbuch Rechtschreibung.
Deutscher Taschenbuch Verlag, 2002. –
ISBN 3–423–32524–0

Anhang

Fallstudie

Matrix als zweidimensionales offenes Feld

- **offene** (dynamische) **Felder** als Datentypen kennt C nicht, sie sind bei Bedarf durch das **Zeigerkonzept** zu implementieren
  - hier beispielsweise durch einen Zeiger auf Zeiger (`mx`)



Instanzenbildung einer $N \times N$ Matrix

```c
#include <stdlib.h>
#include <stdbool.h>

int main (int argc, char *argv[]) {
    if (argc == 3) {
        unsigned int n = atol(argv[2]);
        if (n != 0) {                          /* allocate row pointer field */
            int **mx = (int**)calloc(n, sizeof(int*));
            if (mx != 0) {                     /* succeeded, setup 2nd dimension */
                bool goon = true;
                for (int i = 0; i < n; i++) {  /* allocate integer rows */
                    mx[i] = (int*)calloc(n, sizeof(int));
                    if (mx[i] == 0) {          /* rows incomplete, skip */
                        goon = false;
                        break;
                    }
                }

                if (goon) {                    /* all complete, initialize...*/
                    if (*argv[1] == 'R') by_row(mx, n, 42);
                    else by_column(mx, n, 42);
                }

                for (int i = 0; i < n; i++) {  /* deallocate integer rows */
                    if (mx[i] == 0) break;
                    free(mx[i]);
                }
                free(mx);                      /* deallocate row pointer field */
            }
        }
    }
}
```

Initialisierung der Matrix

`*mx[]` ≡ `**mx`

- **zeilenweises Vorgehen**: Spaltenelemente `j` von Zeile `i` aufzählen

```
1  void by_row (int *mx[], unsigned int n, int v) {
2      unsigned int i, j;
3      for (i = 0; i < n; i++)
4          for (j = 0; j < n; j++)
5              mx[i][j] = v;
6  }
```

- **spaltenweises Vorgehen**: Zeilenelemente `i` von Spalte `j` aufzählen

```
7  void by_column (int *mx[], unsigned int n, int v) {
8      unsigned int i, j;
9      for (j = 0; j < n; j++)
10         for (i = 0; i < n; i++)
11             mx[i][j] = v;
12 }
```

Gemeinsamkeit und Unterschied (vgl. S. 10)

funktional nutzen beide ein zweidimensionales dynamisches Feld

nichtfunktional unterscheiden sie sich im Laufzeitverhalten

Hinter den Kullissen

`mx[i * n + j]` „<“ `mx[i][j]`

- kritischer Lastpunkt (*hotspot*) ist die Zuweisung der inneren Schleife[6]
  - dynamisch angelegtes Feld festen Ausmaßes (S. 10)
    - `by_row`

```
1   LBB0_3:
2       movl %ecx, (%edx,%ebx,4)
3       incl %ebx
4       cmpl %ebx, %eax
5       jne  LBB0_3
```

    - `by_column`

```
6   LBB1_3:
7       movl %ecx, (%ebx)
8       addl %esi, %ebx
9       decl %eax
10      jne  LBB1_3
```

  - dynamisch angelegtes Feld offenen Ausmaßes (S. 42)
    - `by_row`

```
11  LBB0_3:
12      movl %ecx, (%edi,%ebx,4)
13      incl %ebx
14      cmpl %ebx, %eax
15      jne  LBB0_3
```

    - `by_column`

```
16  LBB1_3:
17      movl (%edx,%edi,4), %ebx
18      movl %ecx, (%ebx,%esi,4)
19      incl %edi
20      cmpl %edi, %eax
21      jne  LBB1_3
```

  - zeilenweises Vorgehen ist in beiden Varianten im Zeitverhalten gleich, aber der offene Fall benötigt mehr Speicher (Reihenzeigerfeld)
  - spaltenweises Vorgehen benötigt bei der offenen Variante mehr Zeit, da komplexere und mehr Befehle benötigt werden

[6] `gcc -O -m32 -fomit-frame-pointer -fno-pic -static -S`

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – V.1 Rechnerorganisation: Virtuelle Maschinen

16. Mai 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

Gliederung

Lehrstoff

- Rechensysteme als eine **Schichtenfolge** von Maschinen
  - die eine **funktionale Hierarchie** [7] von spezifischen Maschinen zur Ausführung von Programmen darstellt
  - wobei manche dieser Maschinen nicht in Wirklichkeit vorhanden sind, sein müssen oder sein können
  - die somit jeweils als eine **virtuelle Maschine** [11] in Erscheinung treten

Lehrstoff

- **Abstraktionshierarchie** für Rechensystemkonstruktionen verstehen
  - in der die einzelnen Schichten durch **Prozessoren** implementiert werden, die vor (*off-line*) oder zur (*on-line*) Programmausführungszeit wirken
  - wobei ein Prozessor als **Übersetzer** oder **Interpreter** ausgelegt ist

Lehrstoff

- Platz für das **Betriebssystem** in dieser Hierarchie ausmachen
  - erkennen, dass ein Betriebssystem ein spezieller Interpreter ist und den Befehlssatz wie auch die Funktionalität einer CPU erweitert
  - die **Symbiose** insbesondere von Betriebssystem und CPU verinnerlichen

Lehrstoff

- Rechensysteme als eine **Schichtenfolge** von Maschinen
- **Abstraktionshierarchie** für Rechensystemkonstruktionen verstehen
- Platz für das **Betriebssystem** in dieser Hierarchie ausmachen
- Grundlagen eines „Weltbilds“ legen, das zentral für SP sein wird

Gliederung

Schichtenstruktur

Semantische Lücke

Verschiedenheit zwischen Quell- und Zielsprache

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

Verschiedenheit zwischen Quell- und Zielsprache

Semantische Lücke (*semantic gap*, [14])

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.

Verschiedenheit zwischen Quell- und Zielsprache

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

Semantische Lücke (*semantic gap*, [14])

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

Schichtenstruktur

Fallstudie

Beispiel: Matrizenmultiplikation

Problemraum

(Mathematik)

Lösungsraum

(Informatik)

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.

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“, Quellmodul (`multiply.c`):

```
1  typedef int Matrix [N][N];
2
3  void multiply(in Matrix a, in Matrix b, out Matrix c) {
4      unsigned int i, j, k;
5      for (i = 0; i < N; i++)
6          for (j = 0; j < N; j++) {
7              c[i][j] = 0;
8              for (k = 0; k < N; k++)
9                  c[i][j] += a[i][k] * b[k][j];
10         }
11 }
```

Ebene informatischer Sprache: C

- Skalarprodukt oder „inneres Produkt“, Quellmodul (`multiply.c`):

```
1  typedef int Matrix [N][N];
2
3  void multiply(in Matrix a, in Matrix b, out Matrix c) {
4      unsigned int i, j, k;
5      for (i = 0; i < N; i++)
6          for (j = 0; j < N; j++) {
7              c[i][j] = 0;
8              for (k = 0; k < N; k++)
9                  c[i][j] += a[i][k] * b[k][j];
10         }
11 }
```

- **Konkretisierung** für zwei $N \times N$ Matrizen: $c = a \times b$
  - ausgelegt als Unterprogramm: Prozedur $\mapsto$ *C function*

Ebene informatischer Sprache: C

- Skalarprodukt oder „inneres Produkt“, Quellmodul (`multiply.c`):

```
1  typedef int Matrix [N][N];
2
3  void multiply(in Matrix a, in Matrix b, out Matrix c) {
4      unsigned int i, j, k;
5      for (i = 0; i < N; i++)
6          for (j = 0; j < N; j++) {
7              c[i][j] = 0;
8              for (k = 0; k < N; k++)
9                  c[i][j] += a[i][k] * b[k][j];
10         }
11 }
```

- insgesamt sechs Varianten (d.h., Schleifenanordnungen)
  - $\{\boldsymbol{ijk}, jik, ikj, jki, kij, kji\}$: funktional gleich, nichtfunktional ggf. ungleich

Ebene informatischer Sprache: ASM [8, 4]

```
1     .file "multiply.c"
2     .text
3     .p2align 4,,15
4     .globl multiply
5     .type multiply,@function
6   multiply:
7     pushl %ebp
8     movl  %esp,%ebp
9     pushl %edi
10    pushl %esi
11    pushl %ebx
12    subl  $4,%esp
13    movl  16(%ebp),%esi
14    movl  $0,-16(%ebp)
15  .L2:
16    movl  8(%ebp),%edi
17    xorl  %ebx,%ebx
18    addl  -16(%ebp),%edi
19    .p2align 4,,7
20    .p2align 3
21  .L4:
22    movl  12(%ebp),%eax
23    xorl  %edx,%edx
24    movl  $0,(%esi,%ebx,4)
25    leal  (%eax,%ebx,4),%ecx
26    .p2align 4,,7
27    .p2align 3
28  .L3:
29    movl  (%ecx),%eax
30    addl  $400,%ecx
31    imull (%edi,%edx,4),%eax
32    addl  $1,%edx
33    addl  %eax,(%esi,%ebx,4)
34    cmpl  $100,%edx
35    jne   .L3
36    addl  $1,%ebx
37    cmpl  $100,%ebx
38    jne   .L4
39    addl  $400,-16(%ebp)
40    addl  $400,%esi
41    cmpl  $40000,-16(%ebp)
42    jne   .L2
43    addl  $4,%esp
44    popl  %ebx
45    popl  %esi
46    popl  %edi
47    popl  %ebp
48    ret
49    .size multiply, .-multiply
50    .ident "GCC: (Debian 4.3.2-1.1) 4.3.2"
51    .section .note.GNU-stack,"",@progbits
```

Ebene informatischer Sprache: ASM [8, 4]

```
1     .file "multiply.c"
2     .text
3     .p2align 4,,15
4     .globl multiply
5     .type multiply,@function
6   multiply:
7     pushl %ebp
8     movl  %esp,%ebp
9     pushl %edi
10    pushl %esi
11    pushl %ebx
12    subl  $4,%esp
13    movl  16(%ebp),%esi
14    movl  $0,-16(%ebp)
15  .L2:
16    movl  8(%ebp),%edi
17    xorl  %ebx,%ebx
18    addl  -16(%ebp),%edi
19    .p2align 4,,7
20    .p2align 3
21  .L4:
22    movl  12(%ebp),%eax
23    xorl  %edx,%edx
24    movl  $0,(%esi,%ebx,4)
25    leal  (%eax,%ebx,4),%ecx
26    .p2align 4,,7
27    .p2align 3
28  .L3:
29    movl  (%ecx),%eax
30    addl  $400,%ecx
31    imull (%edi,%edx,4),%eax
32    addl  $1,%edx
33    addl  %eax,(%esi,%ebx,4)
34    cmpl  $100,%edx
35    jne   .L3
36    addl  $1,%ebx
37    cmpl  $100,%ebx
38    jne   .L4
39    addl  $400,-16(%ebp)
40    addl  $400,%esi
41    cmpl  $40000,-16(%ebp)
42    jne   .L2
43    addl  $4,%esp
44    popl  %ebx
45    popl  %esi
46    popl  %edi
47    popl  %ebp
48    ret
49    .size multiply, .-multiply
50    .ident "GCC: (Debian 4.3.2-1.1) 4.3.2"
51    .section .note.GNU-stack,"",@progbits
```

- **Kompilation** der Quelle in ein semantisch äquivalentes Programm
  - Trick: Übersetzung der Quelle vor dem Assemblieren beenden
    - Übersetzung [1] von `multiply.c` mit `-DN=100`: *C function* $\mapsto$ ASM/x86

[1] `gcc -O -m32 -fomit-frame-pointer -fno-pic -static -S`

Ebene informatischer Sprache: `a.out` [8, 2]

```
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
```

- **Assemblieren** der kompilierten Quelle und Ausgabeaufbereitung
  - Hexadezimalkode

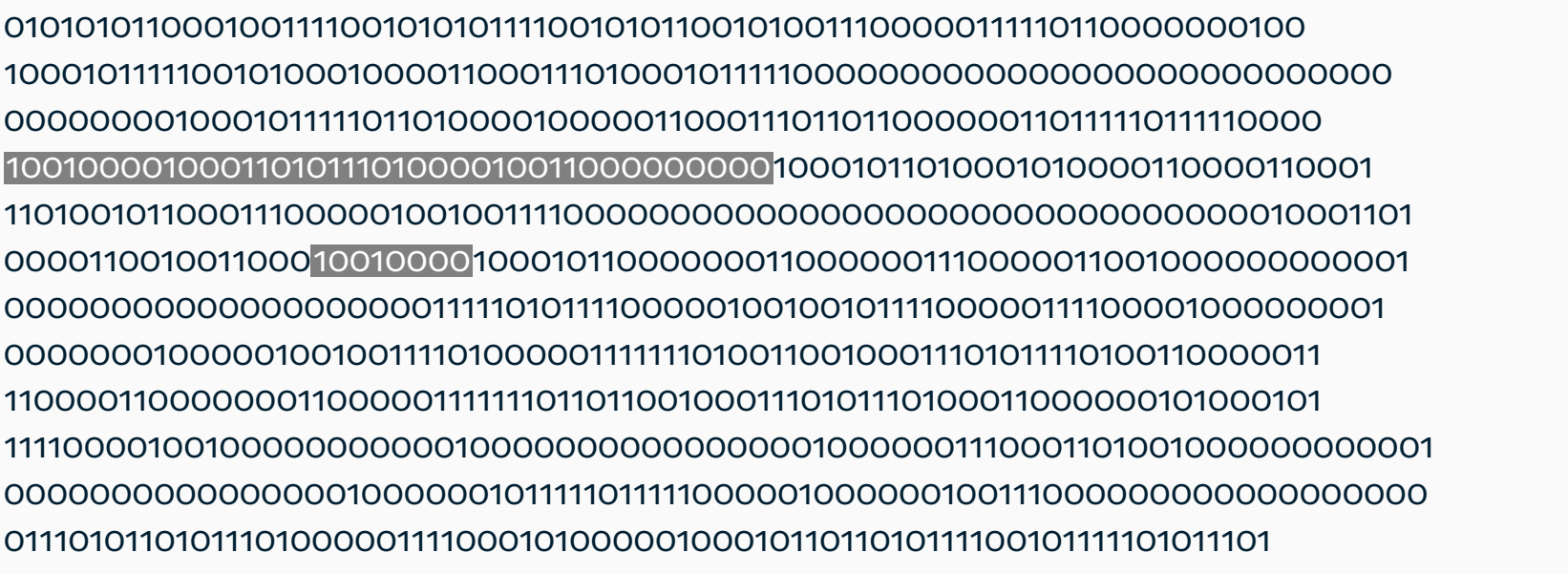
Ebene informatischer Sprache: `a.out` [8, 2]

```
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
```

- **Assemblieren** der kompilierten Quelle und Ausgabeaufbereitung
  - Hexadezimalkode ausführbar — jedoch kein ausführbares Programm!
    1. `as multiply.s`: ASM/x86 $\mapsto$ `a.out`/x86 (Binde-/Lademodul)
    2. `od -x a.out` $\rightsquigarrow$ hexadezimaler (`-x`) **Speicherauszug** (*dump*, `od`)

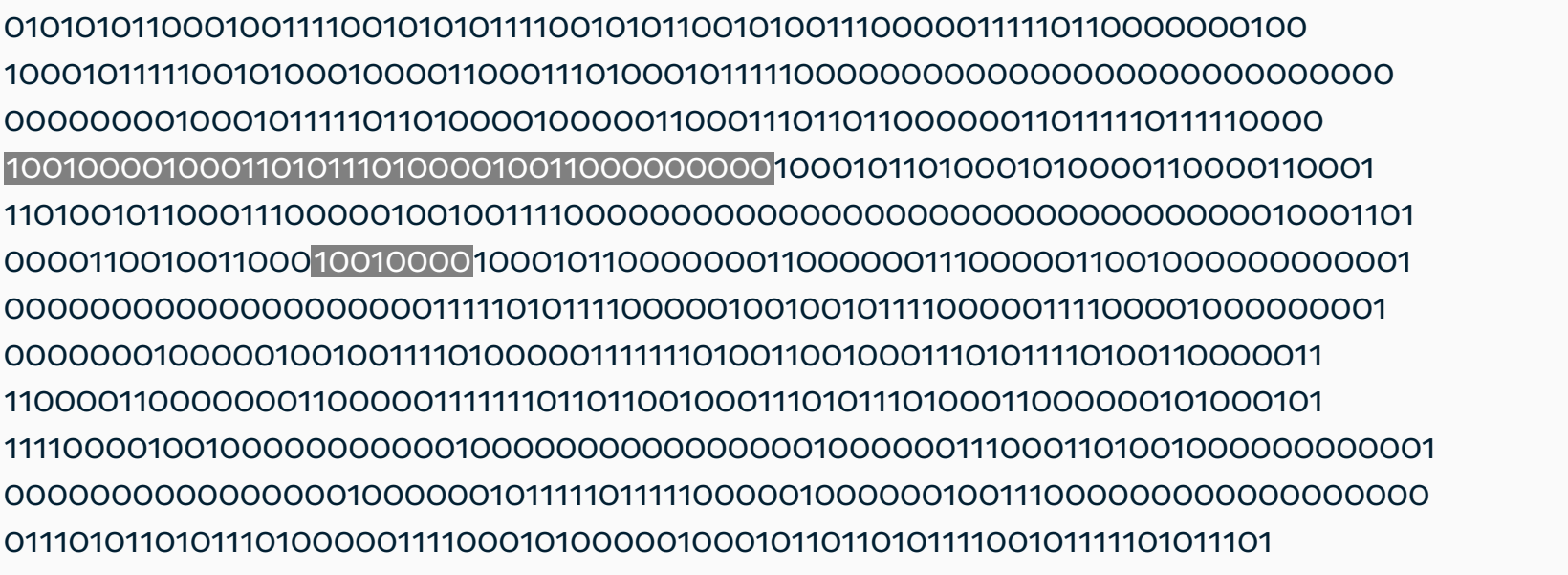
Ebene informatischer Sprache: Binärkode



hervorgehobene Bitfolgen repräsentieren (durch `.p2align`, vgl. S. 21) aufgefüllte Nulloperationen

- **Auflösung** des ausführbaren Hexadezimalkodes zur **Bitfolge**
  - die Befehlsverarbeitung geschieht bitweise, nicht byte- oder wortweise

Ebene informatischer Sprache: Binärkode


01010101100010011111001010101111001010110010100111000001111101100000000100
10001011111100101000100001100011101000101111100000000000000000000000000000
00000000100010111110110100001000001100011101101100000011011111011111000 0
10010000100011010111010000100110000000000100010110100010100001100001100 01
11010010110001110000010010011110000000000000000000000000000000000010001101
00001100100110001001000010001011000000001100000011100000110010000000000001
00000000000000000000011111010111100000100100101111000001111000010000000001
00000001000001001001111010000011111111010011001000111010111010011000000 11
11000011000000011000001111111011011001000111010111010001100000010100010 1
11110000100100000000000100000000000000000100000011100011010010000000000001
00000000000000000100000010111110111110000010000001001110000000000000000000
01110101101011101000001111000101000001000101101101011110010111110101110 1
11000011

hervorgehobene Bitfolgen repräsentieren (durch `.p2align`, vgl. S. 21) aufgefüllte Nulloperationen

Die für einen Digitalrechner letztendlich benötigte Form — obwohl die CPU wortweise[a] auf den Speicher zugreift, mit einer Wortbreite von 8–64 Bits, je nach Prozessortechnologie.

[a] Gemeint ist das **Maschinenwort**, keine plattformunabhängige Dateneinheit.

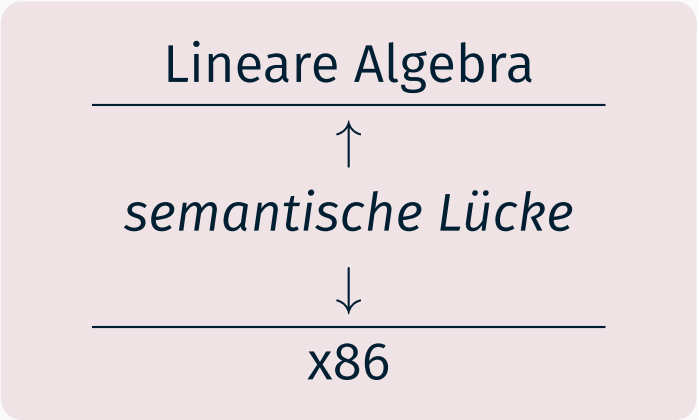
Abstraktionshierarchie von Sprachsystemen

- **Modellsprache** (Lineare Algebra) $\leadsto$ 1 Produktsummenformel
- **Programmiersprache** (C) $\leadsto$ 5 Komplexschritte
- **Assembliersprache** (ASM/x86) $\leadsto$ 35+n Elementarschritte
- **Maschinensprache** (Linux/x86) $\leadsto$ 109 Bytes Programmtext
  (x86) $\leadsto$ 872 Bits

Abstraktionshierarchie von Sprachsystemen

- **Modellsprache** (Lineare Algebra) $\rightsquigarrow$ 1 Produktsummenformel
- **Programmiersprache** (C) $\rightsquigarrow$ 5 Komplexschritte
- **Assembliersprache** (ASM/x86) $\rightsquigarrow$ $35+n$ Elementarschritte
- **Maschinensprache** (Linux/x86) $\rightsquigarrow$ 109 Bytes Programmtext
  (x86) $\rightsquigarrow$ 872 Bits

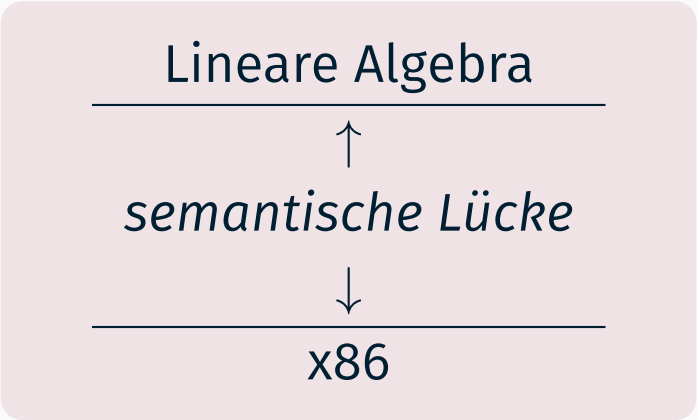
$\hookrightarrow$ eine einzelne komplexe und überwältigende Aufgabe

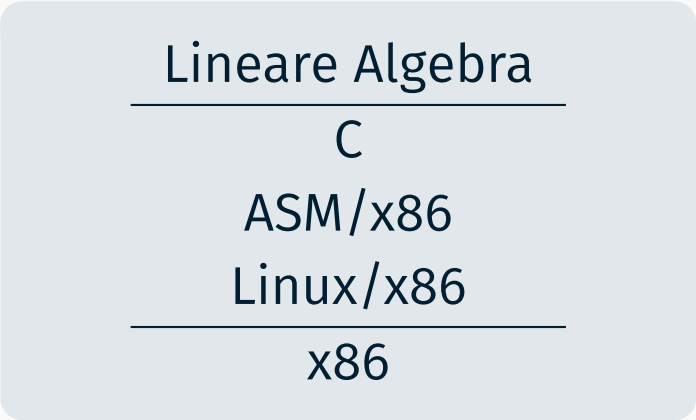


Abstraktionshierarchie von Sprachsystemen

- **Modellsprache** (Lineare Algebra) $\rightsquigarrow$ 1 Produktsummenformel
- **Programmiersprache** (C) $\rightsquigarrow$ 5 Komplexschritte
- **Assembliersprache** (ASM/x86) $\rightsquigarrow$ $35+n$ Elementarschritte
- **Maschinensprache** (Linux/x86) $\rightsquigarrow$ 109 Bytes Programmtext
  (x86) $\rightsquigarrow$ 872 Bits

$\hookrightarrow$ eine einzelne komplexe und überwältigende Aufgabe in mehrere kleine und handhabbare unterteilen



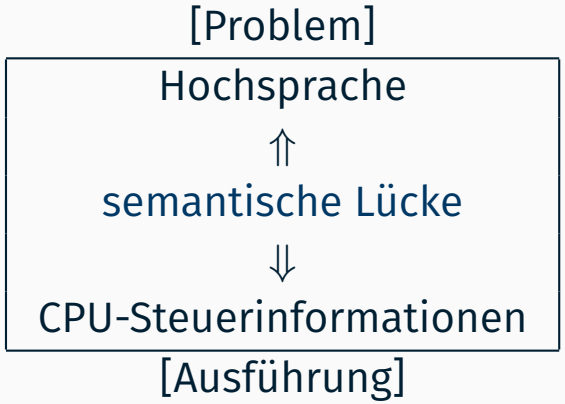


Gliederung

Mehrebenenmaschinen

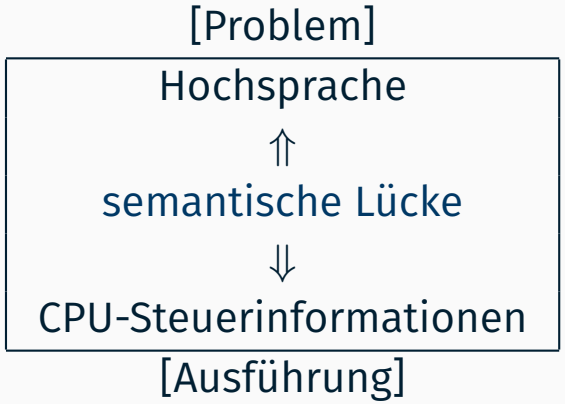
Maschinenhierarchie

Aufgabenstellung $\mapsto$ Programmlösung

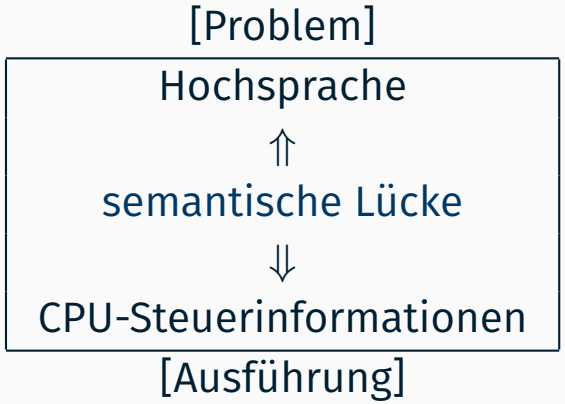


Aufgabenstellung $\mapsto$ Programmlösung

- das Ausmaß der semantischen Lücke gestaltet sich fallabhängig:
  - bei gleich bleibendem Problem mit der Plattform (dem System)
  - bei gleich bleibender Plattform mit dem Problem (der Anwendung)

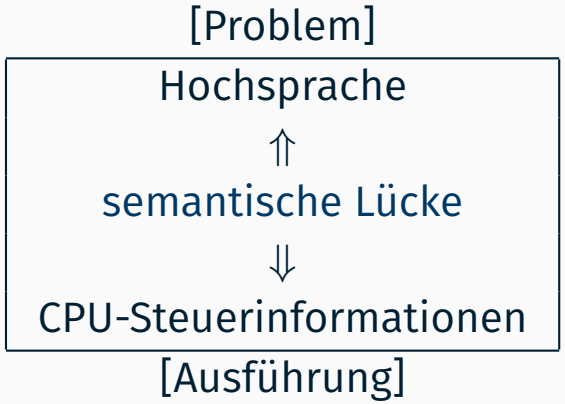


Aufgabenstellung $\mapsto$ Programmlösung



- der Lückenschluss ist ganzheitlich zu sehen und auch anzugehen
  - Schicht für Schicht die innere (logische) Struktur des Systems herleiten
  - das System, das die Lücke schließen soll, als Ganzes als „Bild“ erfassen
    - hinsichtlich benötigter funktionalen und nicht-funktionalen Eigenschaften

Aufgabenstellung $\mapsto$ Programmlösung



- Kunst der kleinen Schritte: semantische Lücke schrittweise schließen
  - durch hierachisch angeordnete virtuelle Maschinen Programmlösungen auf die reale Maschine herunterbrechen [12]
  - Prinzip *divide et impera* („teile und herrsche“)
    - einen „Gegner“ in leichter besiegbare „Untergruppen“ aufspalten

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren)

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren):

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

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren):

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
⋮	⋮	⋮

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren):

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

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren):

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

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren):

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

Hierarchie virtueller Maschinen [13, S. 3]

- **Interpretation** und **Übersetzung** (Kompilation, Assemblieren)

- Techniken, die einander unterstützend — teils sogar „symbiotisch“ — Verwendung finden, um Programme zur Ausführung zu bringen

Abstrakter Prozessor

- jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen Prozessor implementiert

Abstrakter Prozessor

- jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen Prozessor implementiert:

  Kom|pi|la|tor *lat.* (Zusammenträger)
  - ein **Softwareprozessor**, transformiert in einer *Quellsprache* vorliegende Programme in eine semantisch äquivalente Form einer *Zielsprache*
    - {Ada, C, C++, Eiffel, Modula, Fortran, Pascal, …} $\mapsto$ Assembler
    - aber ebenso: C++ $\mapsto$ C $\mapsto$ Assembler

Abstrakter Prozessor

- jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen Prozessor implementiert:

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

- ein **Hard-, Firm- oder Softwareprozessor**, der die Programme direkt ausführt $\rightsquigarrow$ ausführbares Programm (*executable*)
  - z.B. Basic, Perl, C, `sh(1)`, x86

Abstrakter Prozessor

- jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen Prozessor implementiert:

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

- ein **Hard-, Firm- oder Softwareprozessor**, der die Programme direkt ausführt $\rightsquigarrow$ ausführbares Programm (*executable*)
  - z.B. Basic, Perl, C, `sh(1)`, x86
- ggf. **Vorübersetzung** durch einen Kompilierer, um die Programme in eine für die Interpretation günstigere Repräsentation zu bringen
  - z.B. Pascal P-Code, Java Bytecode, x86-Befehle

Abstrakter Prozessor

- jede einzelne Ebene (d.h., Schicht) in der Hierarchie wird durch einen spezifischen Prozessor implementiert:

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

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

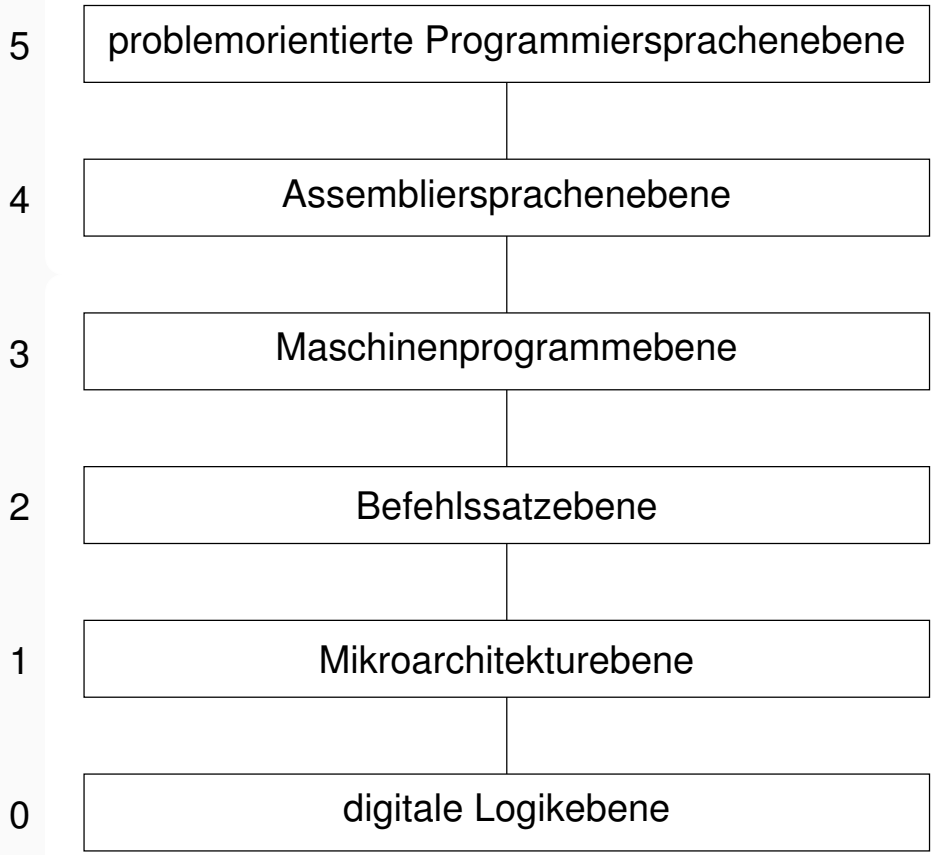
- also abstrakte oder reale Prozessoren, die vor beziehungsweise zur Ausführungszeit des Programms wirken, das sie verarbeiten

Mehrebenenmaschinen

Maschinen und Prozessoren

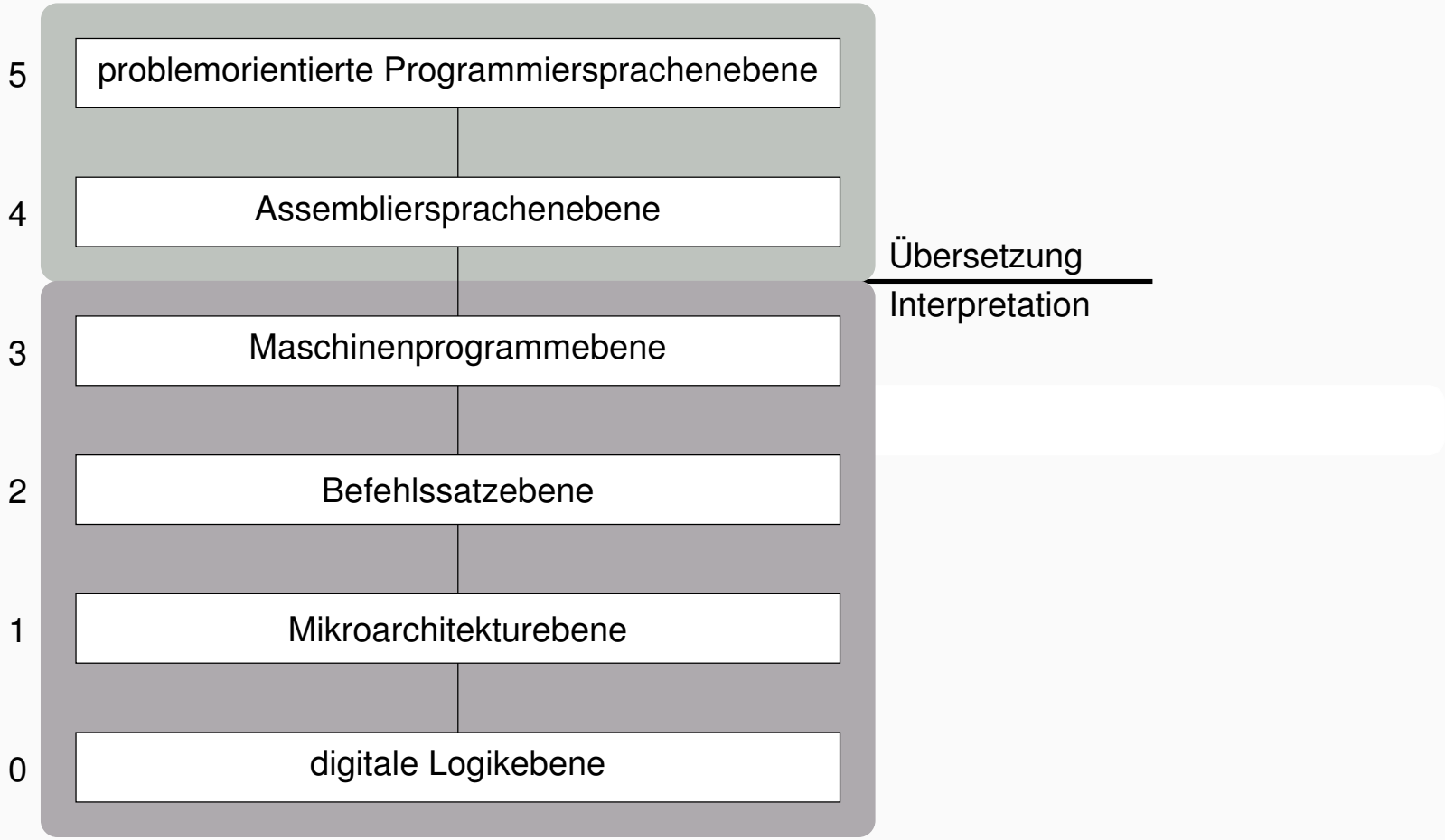
Schichtenfolge in Rechensystemen I

in Anlehnung an [12]



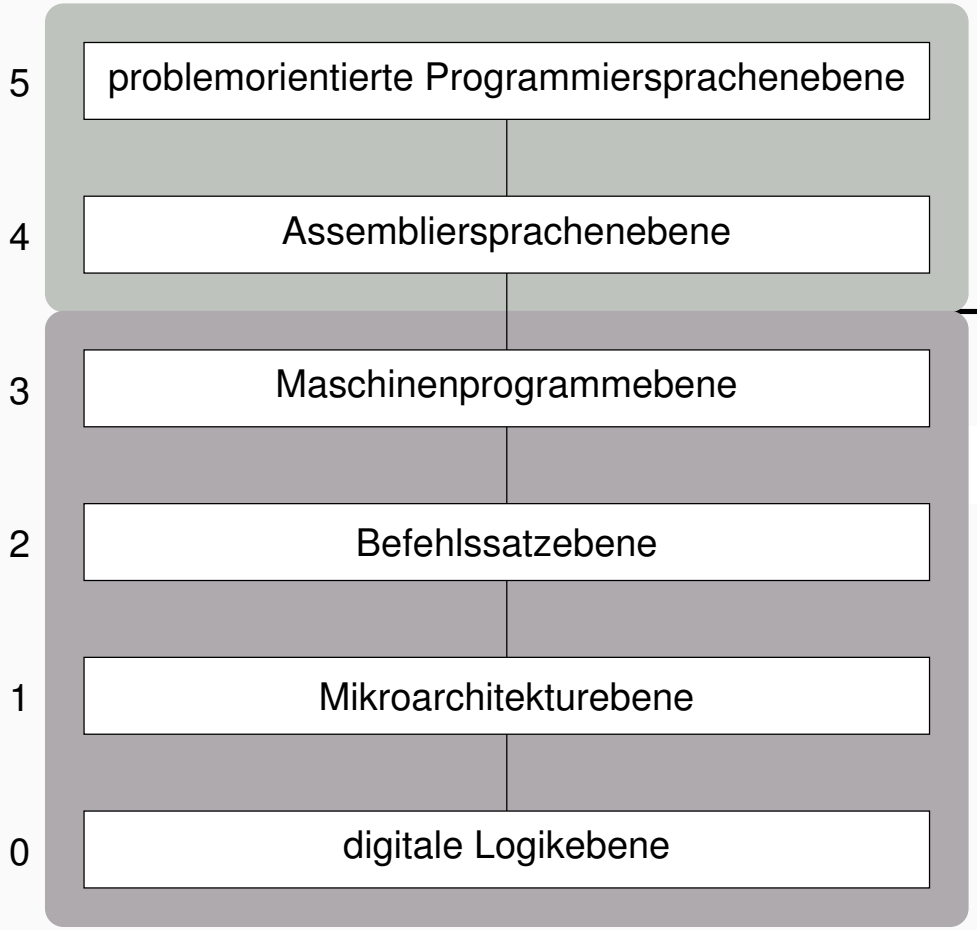
Schichtenfolge in Rechensystemen I

in Anlehnung an [12]



Schichtenfolge in Rechensystemen I

in Anlehnung an [12]



Übersetzung

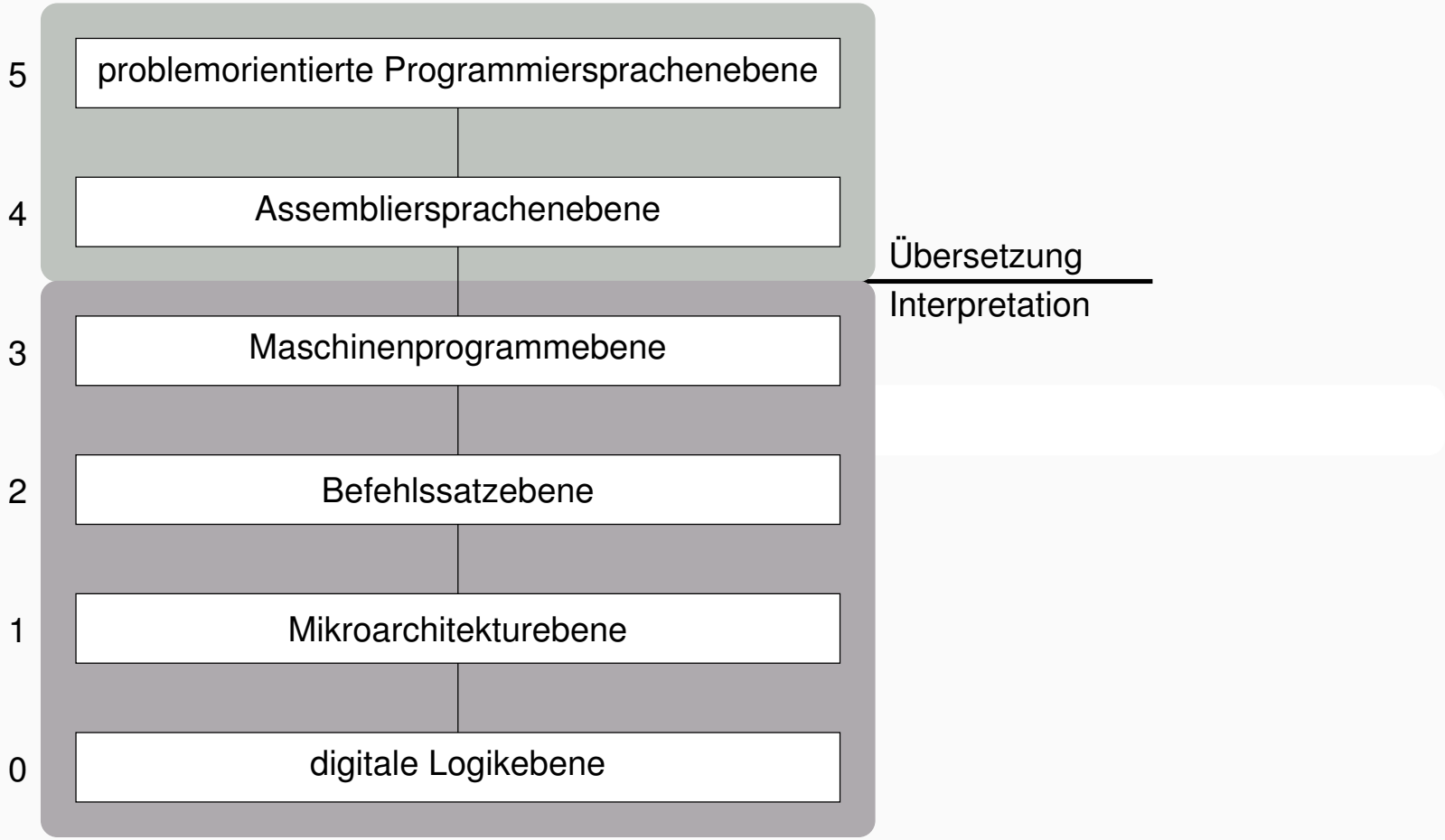
Interpretation

Demarkationslinie

- Methode der Abbildung
  - ≥ 4 Übersetzung vs.
  - ≤ 3 Interpretation
- Art der Programmierung
  - ≥ 4 Anwendung vs.
  - ≤ 3 System
- Natur der Sprache
  - ≥ 4 symbolisch vs.
  - ≤ 3 numerisch

Schichtenfolge in Rechensystemen I

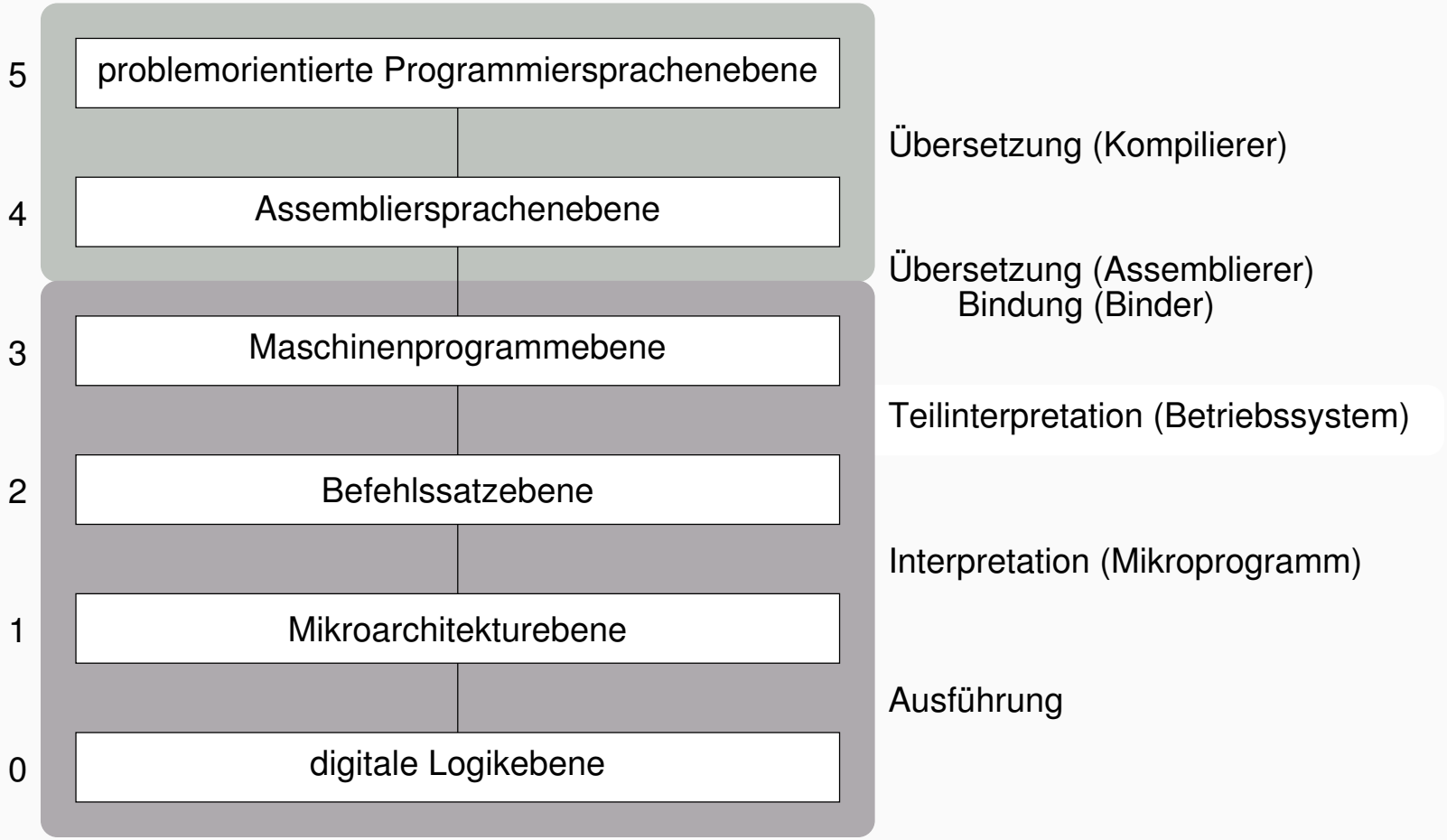
in Anlehnung an [12]



- Schichten der Ebene$_{[4,5]}$ sind nicht wirklich existent sondern
  - werden durch Übersetzung aufgelöst und auf tiefere Ebenen abgebildet
  - so dass am Ende nur ein Maschinenprogramm (Ebene$_3$) übrig bleibt

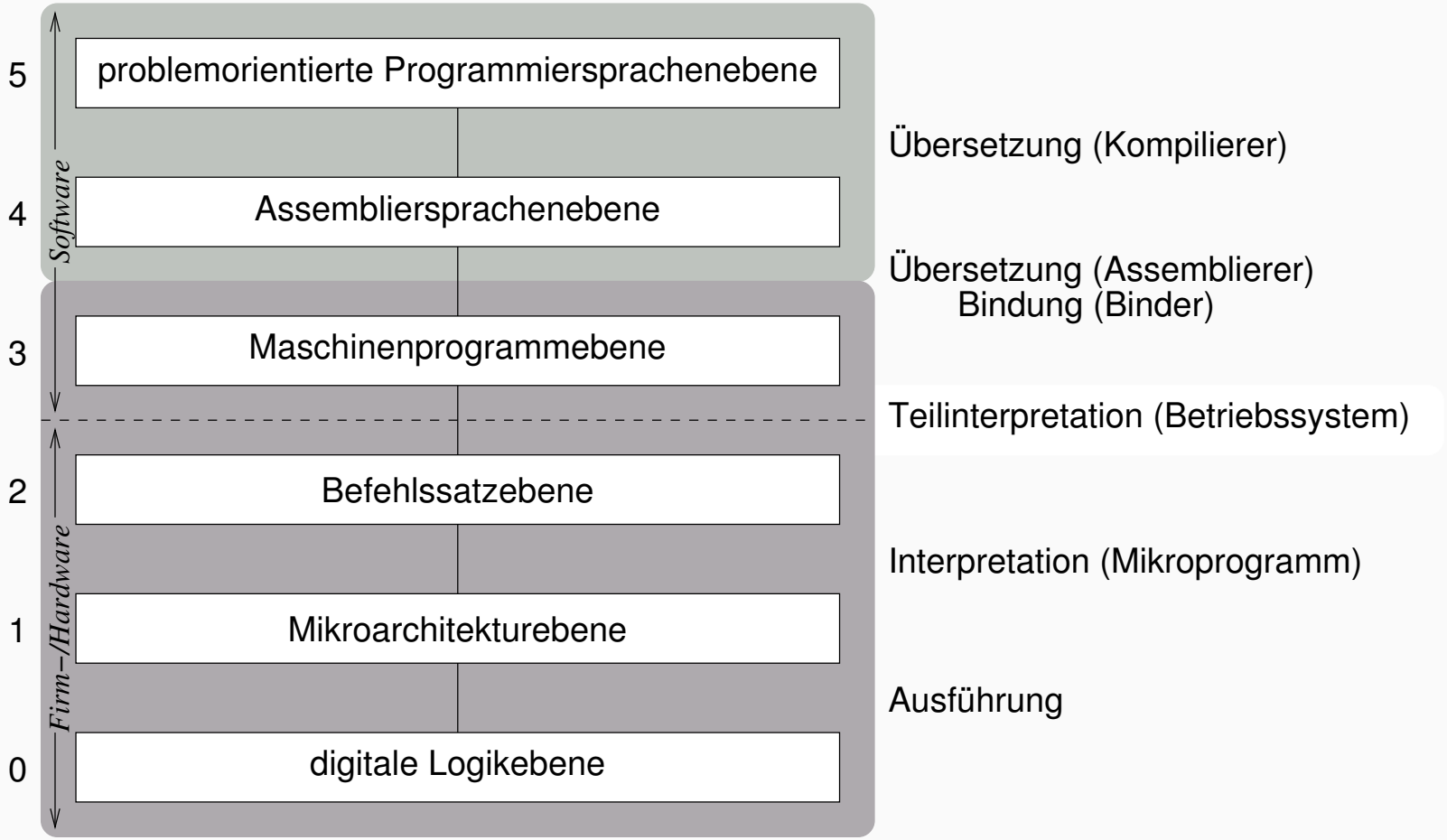
Schichtenfolge in Rechensystemen II

in Anlehnung an [12]



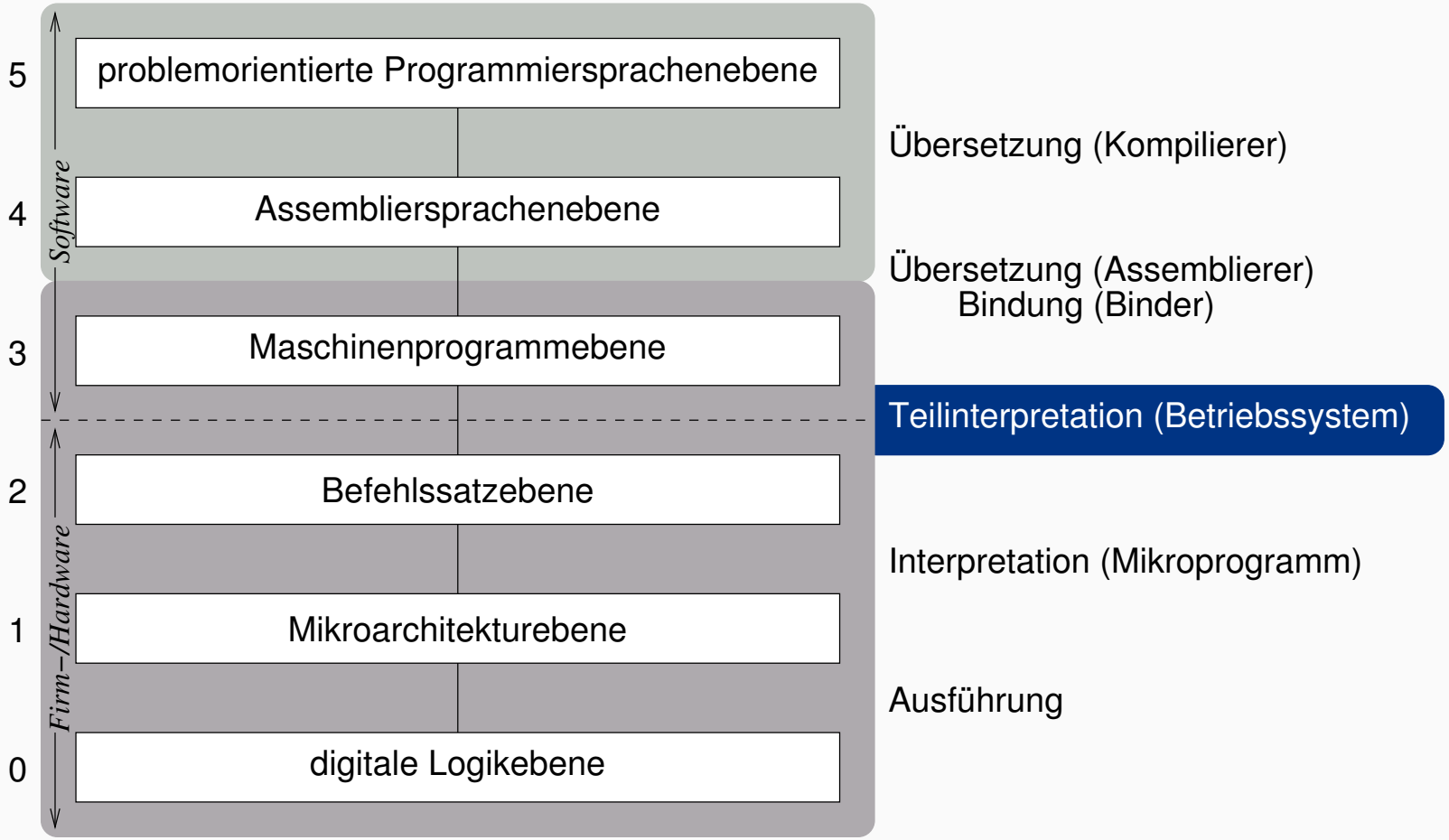
Schichtenfolge in Rechensystemen II

in Anlehnung an [12]



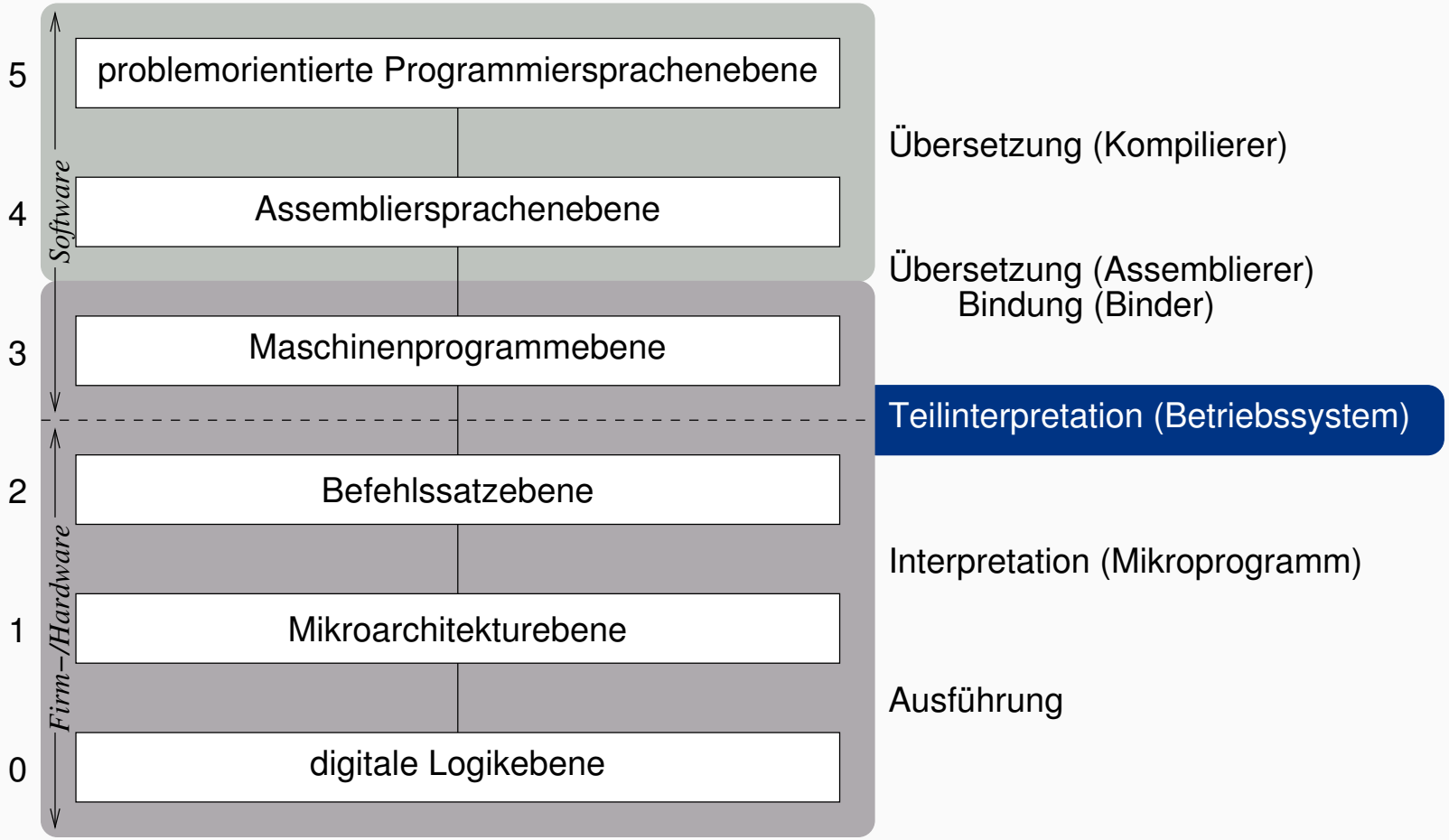
Schichtenfolge in Rechensystemen II

in Anlehnung an [12]



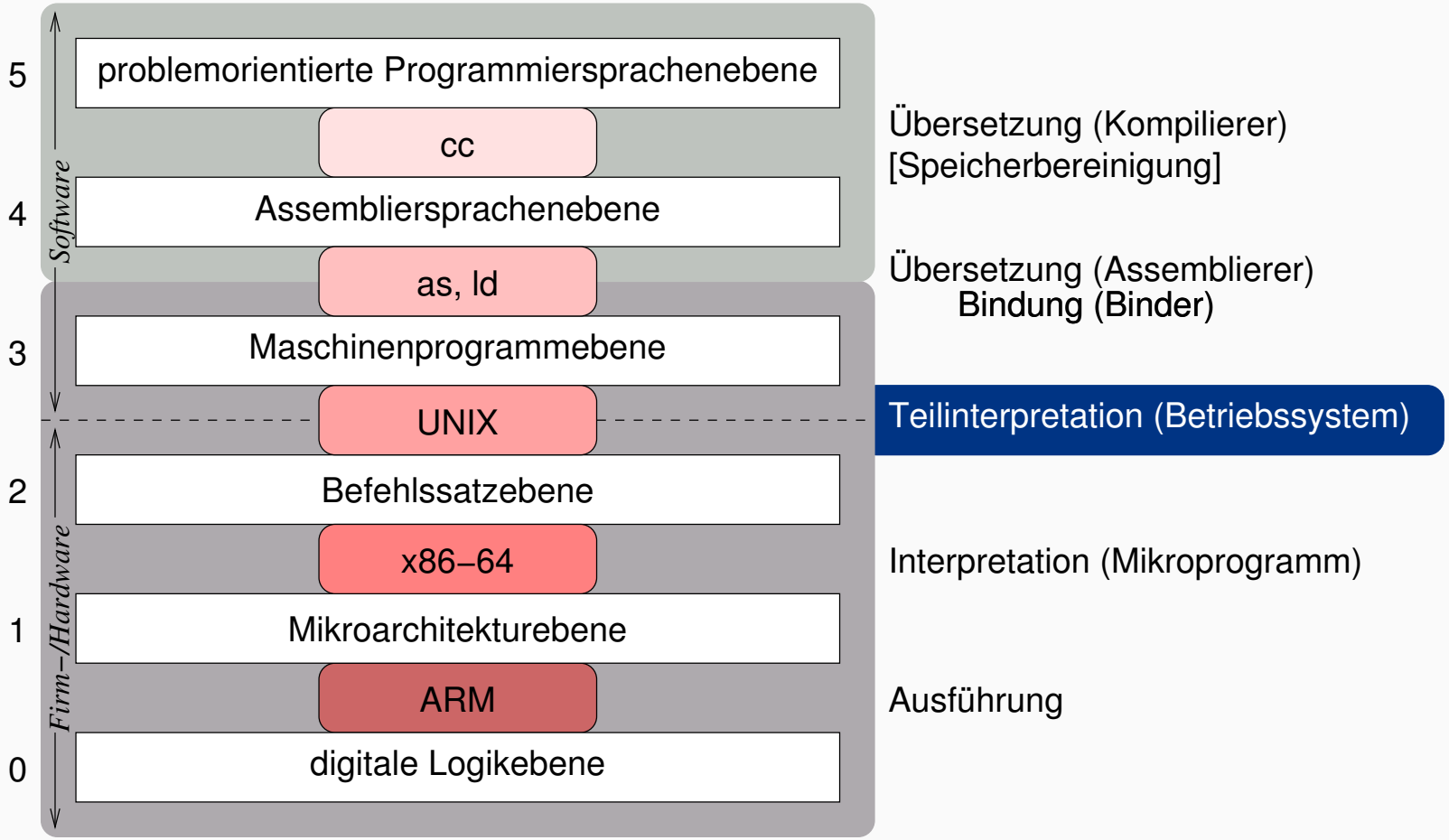
Schichtenfolge in Rechensystemen II

in Anlehnung an [12]

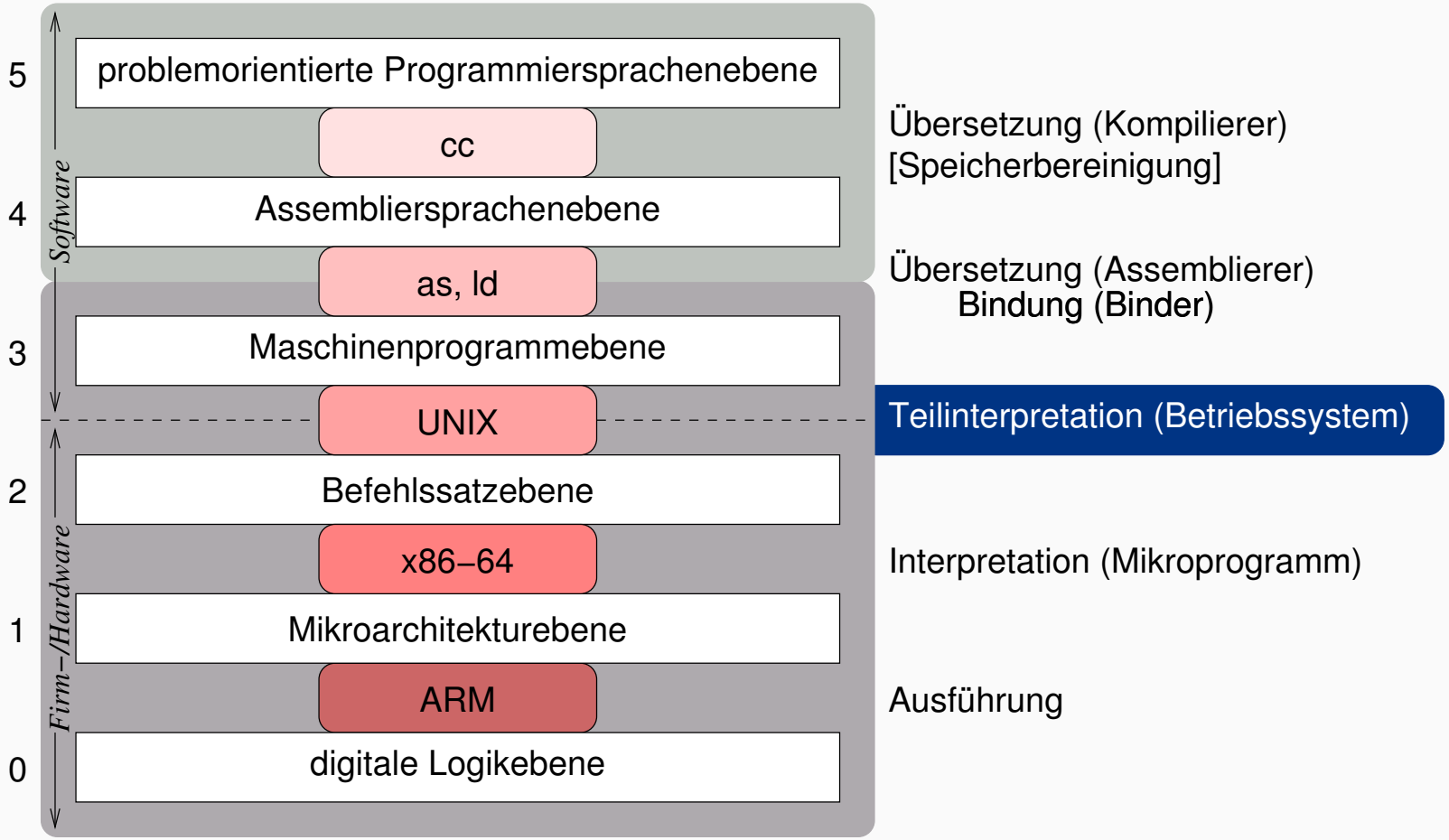


- Schichten der Ebene$_{[0,2]}$ liegen normalerweise nicht in SW vor
  - sie können aber in Software simuliert/emuliert oder virtualisiert werden
  - dadurch lassen sich Rechensysteme grundsätzlich **rekursiv** organisieren

Schichtenfolge in Rechensystemen III

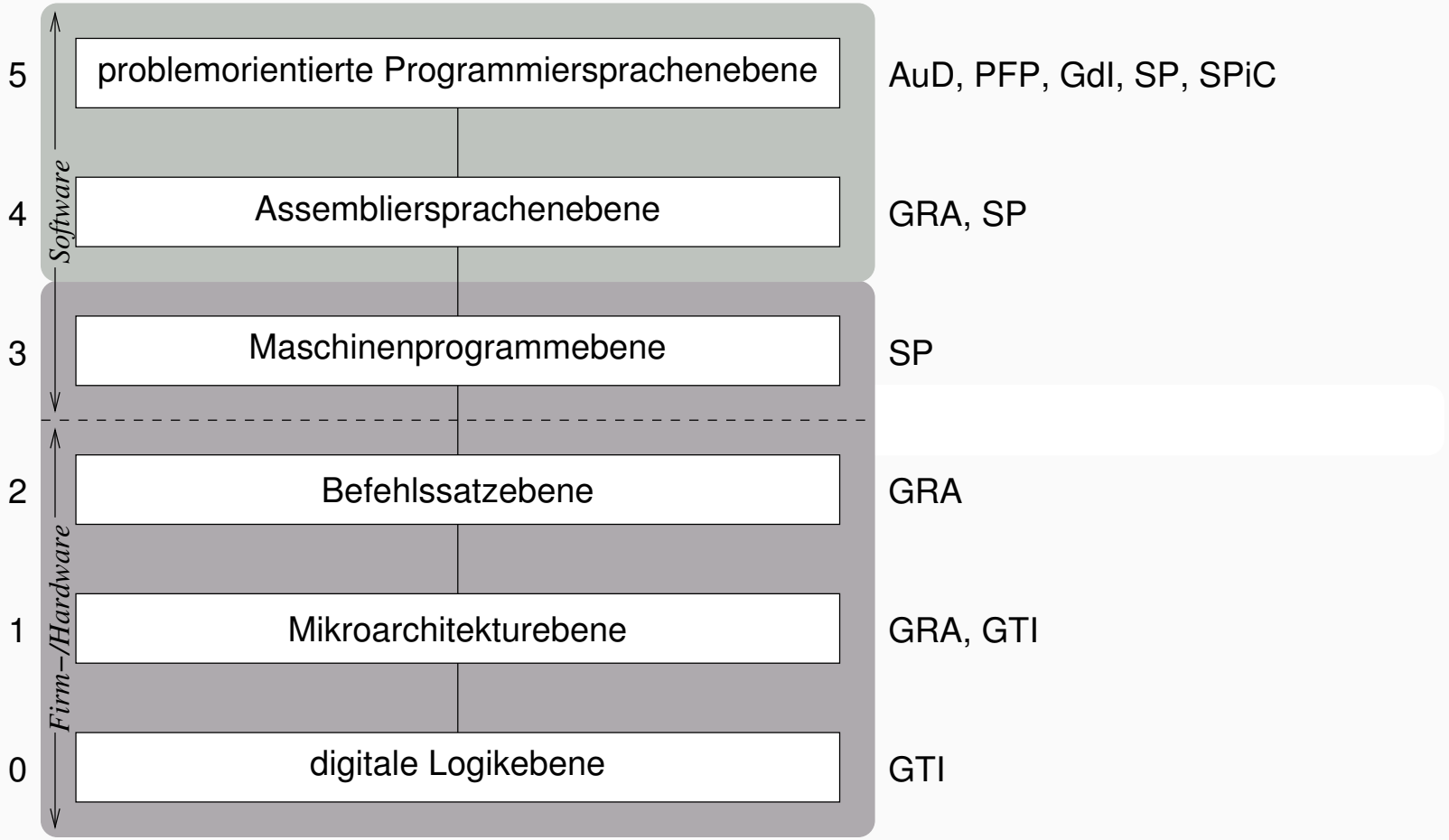


Schichtenfolge in Rechensystemen III

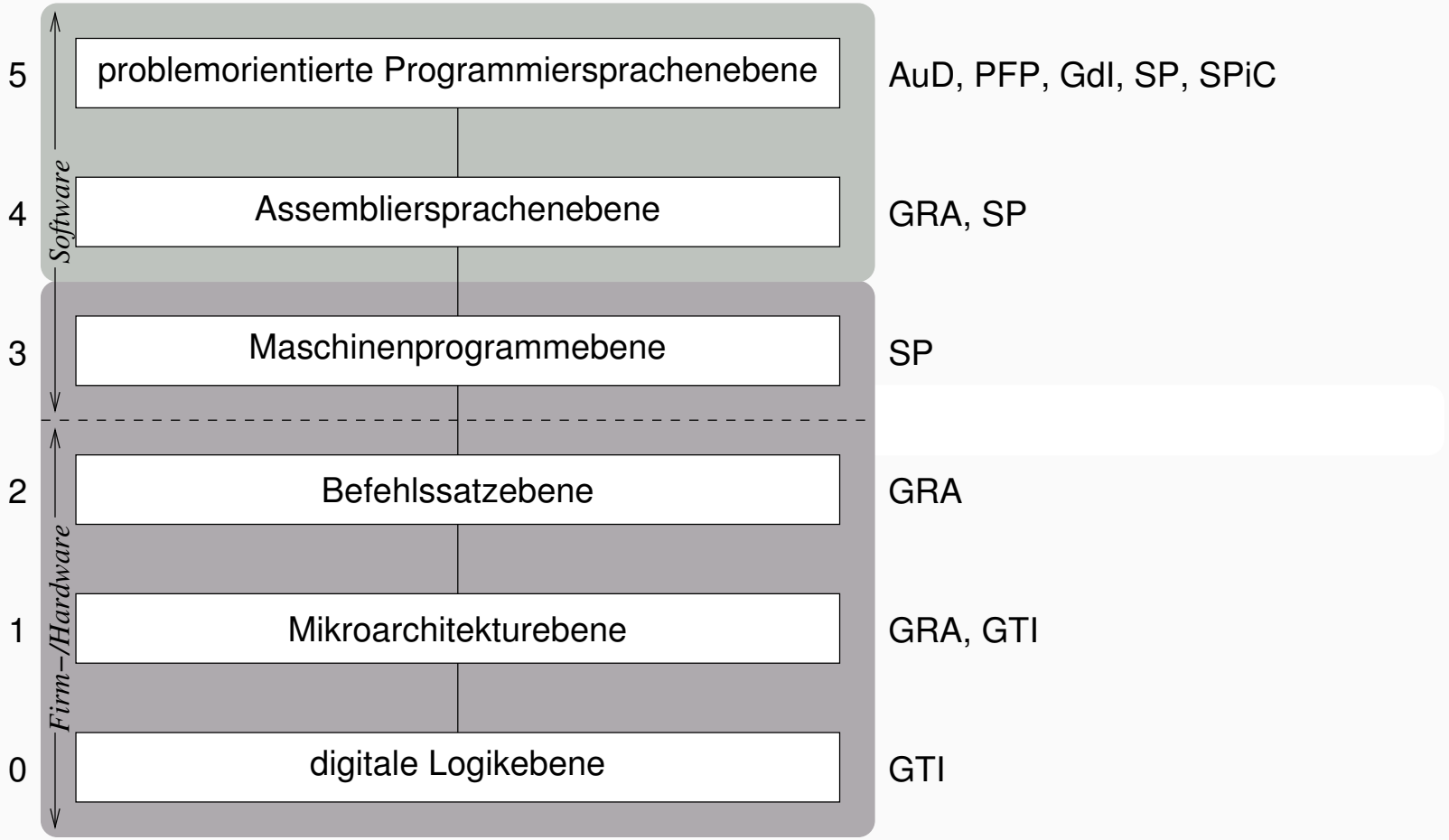


- RISC auf Ebene$_1$ und gegebenenfalls (hier) CISC auf Ebene$_2$
  - nach außen „*complex*“, innen aber „*reduced instruction set computer*“
  - Intel *Core* oder *Haswell* $\leftrightarrow$ AMD *Bulldozer* oder *Zen* (ARM)

Schichtenfolge eingebettet im Informatikstudiengang



Schichtenfolge eingebettet im Informatikstudiengang



- die Schicht auf Ebene$_4$ ist auch hier eher nur logisch existent ☺
  - Programmierung in Assembliersprache hat (leider) an Bedeutung verloren
  - Prinzipien werden in GRA vermittelt [5], in SP nur bei Bedarf behandelt

Mehrebenenmaschinen

Entvirtualisierung

Abbildung der Schichten

Abbildung der Schichten

- Schichten der Ebene$_{[3,5]}$ repräsentieren **virtuelle Maschinen**, die auf die eine **reale Maschine** (Ebene$_{[0,2]}$) abzubilden sind
  - dabei werden diese Schichten „entvirtualisiert“, aufgelöst und zu einem **Maschinenprogramm** „verschmolzen“
  - dieser Vorgang hängt stark ab von der Art einer virtuellen Maschine[2]

[2] vgl. insb. [10]: die Folien sind Teil des ergänzenden Materials zu SP.

Abbildung der Schichten

- **Übersetzung**
  - aller Befehle des Programms, das der Ebene$_i$ zugeordnet ist
  - in eine semantisch äquivalente Folge von Befehlen der Ebene$_j$, mit $j \leq i$
  - dadurch **Generierung** eines Programms, das der Ebene$_j$ zugeordnet ist

Abbildung der Schichten

- **Interpretation**
  - **total**
    - aller Befehle des Programms, das der Ebene$_i$ zugeordnet ist
  - **partiell**
    - nur der Befehle des Programms, die der Ebene$_i$ zugeordnet sind
    - wobei das Programm der Ebene$_k$, $k \geq i$, zugeordnet sein kann
  - durch **Ausführung** eines Programms der Ebene$_j$, mit $j \leq i$

Abbildung durch Übersetzung

$\text{Ebene}_5 \mapsto \text{Ebene}_4$

- Ebene_5-Befehle „1:N“, $N \geq 1$, in Ebene_4-Befehle übersetzen
  - einen Hochsprachenbefehl als mögliche Sequenz von Befehlen einer Assembliersprache implementieren
  - eine **semantisch äquivalente Befehlsfolge** generieren
- im Zuge der Transformation ggf. Optimierungsstufen durchlaufen

Abbildung durch Übersetzung

$\text{Ebene}_4 \mapsto \text{Ebene}_3$

- Ebene_4-Befehle „1:1“ in Ebene_3-Befehle übersetzen
  - ein **Quellmodul** in ein **Objektmodul** umwandeln
  - mit **Bibliotheken** zum Maschinenprogramm zusammenbinden
    - ein **Lademodul** erzeugen

Abbildung durch Übersetzung

$\text{Ebene}_4 \mapsto \text{Ebene}_3$

- Ebene_4-Befehle „1:1“ in Ebene_3-Befehle übersetzen

- dabei den symbolischen Maschinenkode (d.h., die Mnemone) auflösen
  - in binären Maschinenkode umwandeln
    - `ADD EAX` (Mnemon) $\mapsto 05_{16}$ (Hexadezimalkode) $\mapsto 00000101_2$ (Binärkode)
    - hier: Beispiel für den Befehlssatz x86-kompatibler Prozessoren

Abbildung durch Übersetzung

$\text{Ebene}_5 \mapsto \text{Ebene}_4$ (Kompilation)

$\text{Ebene}_4 \mapsto \text{Ebene}_3$ (Assemblieren)

Abbildung durch Interpretation

$\text{Ebene}_3 \mapsto \text{Ebene}_2$

- Ebene_3-Befehle typ- und zustandsabhängig verarbeiten:
  i als Folgen von Ebene_2-Befehlen ausführen
    - **Systemaufrufe** annehmen und befolgen, sensitive Ebene_2-Befehle emulieren
    - synchrone/asynchrone **Unterbrechungen** behandeln
  ii „1:1“ auf Ebene_2-Befehle abbilden (nach unten „durchreichen“)

Abbildung durch Interpretation

$\text{Ebene}_3 \mapsto \text{Ebene}_2$

- Ebene_3-Befehle typ- und zustandsabhängig verarbeiten:
  i als Folgen von Ebene_2-Befehlen ausführen
  - **Systemaufrufe** annehmen und befolgen, sensitive Ebene_2-Befehle emulieren

- ein Ebene_3-Befehl aktiviert im Fall von i ein Ebene_2-Programm
  - verursacht durch eine **Ausnahmesituation**, die durch Ebene_2 erkannt und zur Behandlung an ein Programm der Ebene_2 „hochgereicht“ wird
  - Ebene_2 stellt eine Falle (*trap*), bedient von einem Ebene_2-Programm

Abbildung durch Interpretation

$\text{Ebene}_2 \mapsto \text{Ebene}_1$

- Ebene_2-Befehle als Folgen von Ebene_1-Aktionen ausführen
  - **Abruf- und Ausführungszyklus** (*fetch-execute-cycle*) der CPU
- ein Ebene_2-Befehl löst Ebene_1-Steueranweisungen aus

Abbildung durch Interpretation

$\text{Ebene}_3 \mapsto \text{Ebene}_2$ (*partielle* Interpretation, Teilinterpretation)

$\text{Ebene}_2 \mapsto \text{Ebene}_1$ (Interpretation)

Zeitpunkte der Abbildungsvorgänge

Bezugspunkt ist das jeweils zu „prozessierende“ Programm:

- **vor Laufzeit** ($\text{Ebene}_5 \mapsto \text{Ebene}_3$) $\rightsquigarrow$ **statisch**
  - Vorverarbeitung (*preprocessing*)
  - Vorübersetzung (*precompilation*)
  - Übersetzung: Kompilation, Assemblieren
  - Binden (*static linking*)

Zeitpunkte der Abbildungsvorgänge

Bezugspunkt ist das jeweils zu „prozessierende“ Programm:

- **zur Laufzeit** ($\text{Ebene}_5 \mapsto \text{Ebene}_1$) $\rightsquigarrow$ **dynamisch**
  - bedarfsorientierte Übersetzung (*just in time compilation*)
  - Binden (*dynamic linking*)
  - bindendes Laden (*linking loading*, *dynamic loading*)
  - Teilinterpretation
  - Interpretation

Zeitpunkte der Abbildungsvorgänge

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

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

Betriebssysteme entvirtualisieren zur Laufzeit

$\hookrightarrow$ dynamisches Binden, bindendes Laden, Teilinterpretation

Mehrebenenmaschinen

Ausnahmesituation

Abweichung vom normalen Programmablauf

Abweichung vom normalen Programmablauf

- **Ausnahme** (*exception*), Sonderfall, der die **Unterbrechung** oder den **Abbruch** der Ausführung des Maschinenprogramms bedeutet

Abweichung vom normalen Programmablauf

- **Ausnahme** (*exception*), Sonderfall, der die **Unterbrechung** oder den **Abbruch** der Ausführung des Maschinenprogramms bedeutet
  - Feststellung einer **Ausnahmesituation** beim Abruf-/Ausführungszyklus
    - ungültiger Maschinenbefehl oder Systemaufruf
    - Schutz-/Zugriffsverletzung, Seitenfehler, Unterbrechungsanforderung

Abweichung vom normalen Programmablauf

- **Ausnahme** (*exception*), Sonderfall, der die **Unterbrechung** oder den **Abbruch** der Ausführung des Maschinenprogramms bedeutet

  - zieht die Reaktion in Form einer **Ausnahmebehandlung** nach sich
    - realisiert durch ein spezielles Programm, einem Unterprogramm ähnlich
    - das durch erheben (*raise*) einer Ausnahme implizit aufgerufen wird

Abweichung vom normalen Programmablauf

- **Ausnahme** (*exception*), Sonderfall, der die **Unterbrechung** oder den **Abbruch** der Ausführung des Maschinenprogramms bedeutet

- die Behandlung eines solchen Sonderfalls verläuft je nach Art und Schwere der Ausnahme nach verschiedenen Modellen:

  Wiederaufnahme
  - Ausführungsfortsetzung nach erfolgter Behandlung

  ↪ Seitenfehler, Unterbrechungsanforderung

  Termination
  - Ausführungsabbruch, schwerwiegender Fehler

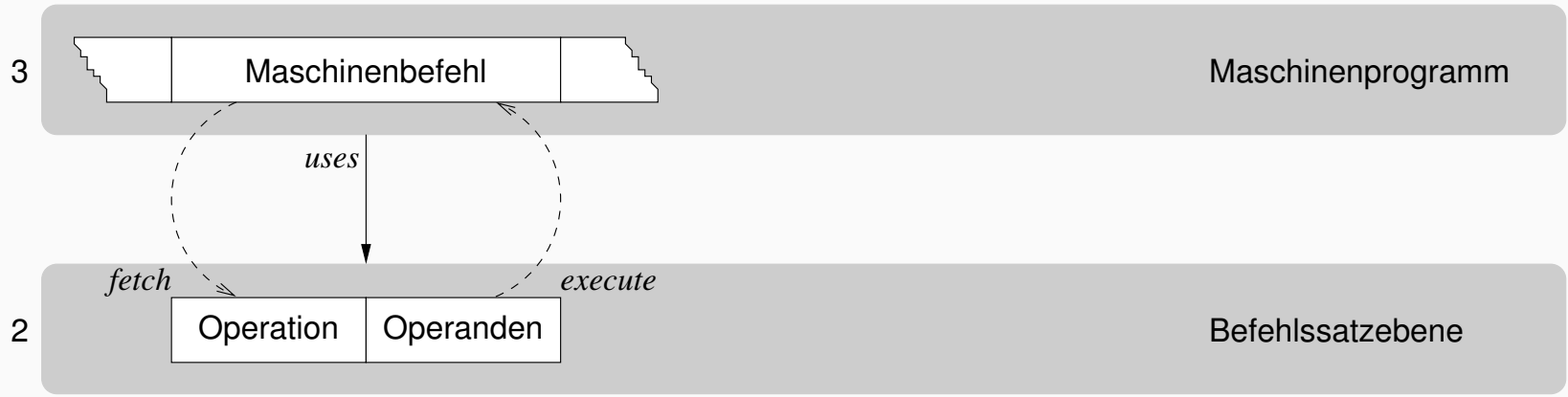
  ↪ ungültiger Befehl, Schutz-/Zugriffsverletzung

Abweichung vom normalen Programmablauf

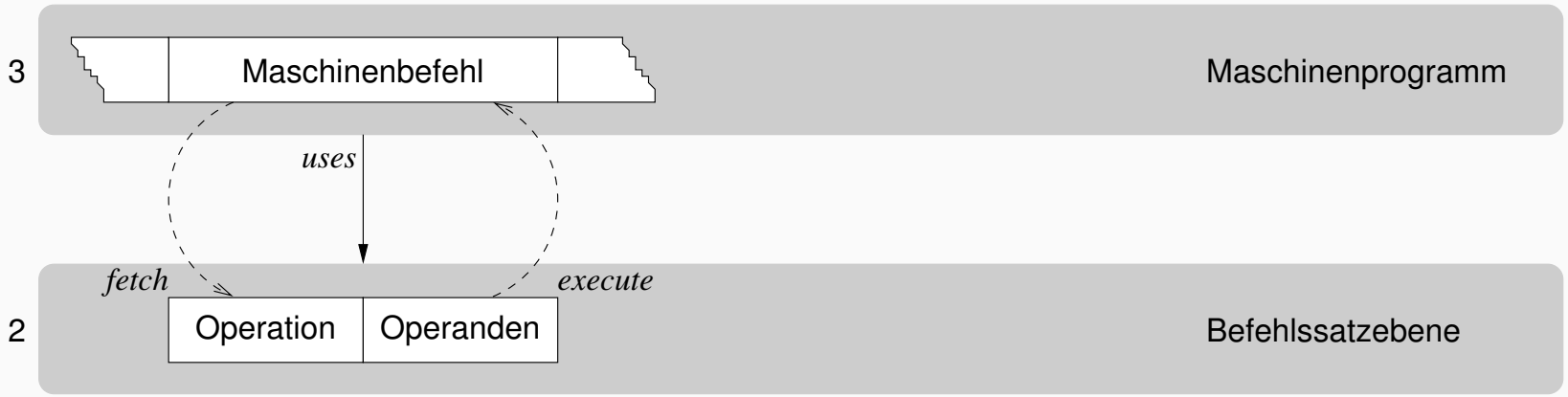
- **Ausnahme** (*exception*), Sonderfall, der die **Unterbrechung** oder den **Abbruch** der Ausführung des Maschinenprogramms bedeutet

- manche Programmiersprachen (z.B. Java, C++) bieten Konstrukte zum Umgang mit solchen Ausnahmen

Sonderfallbehandlung I

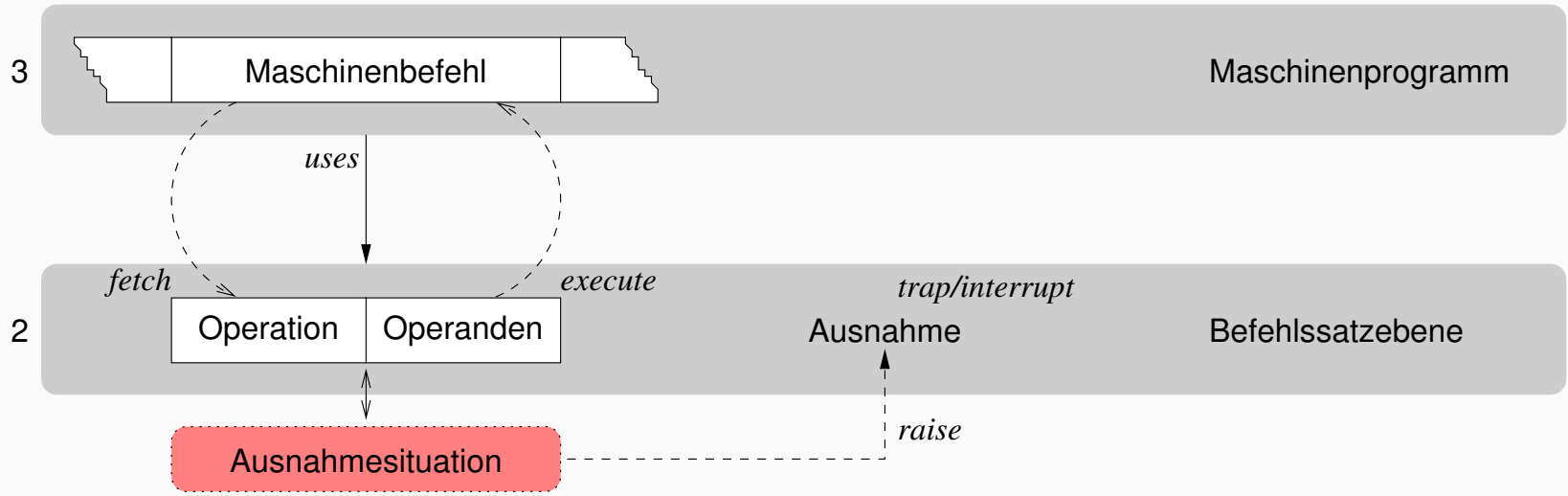


Sonderfallbehandlung I



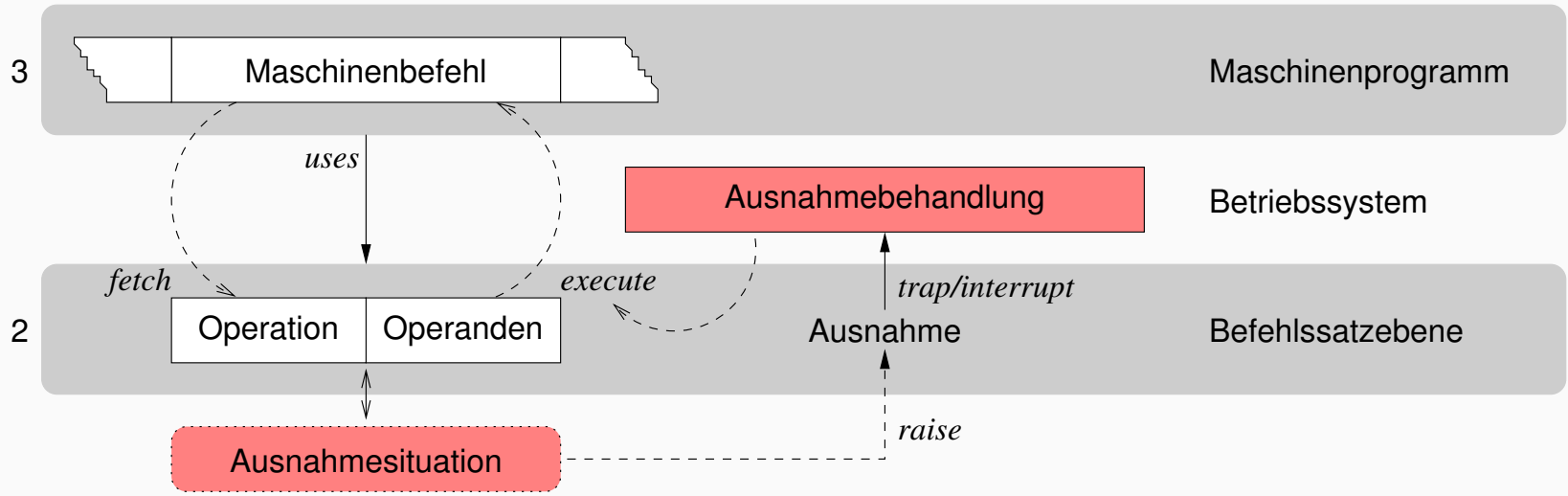
- im Abruf- und Ausführungszyklus interpretiert die CPU den nächsten Maschinenbefehl, führt so das Programm weiter aus
  - ein solcher Befehl hat einen Operations- und ggf. Operandenteil

Sonderfallbehandlung I



- im Abruf- und Ausführungszyklus interpretiert die CPU den nächsten Maschinenbefehl, führt so das Programm weiter aus
- bei der Interpretation dieses Befehls tritt eine Ausnahmesituation auf, die CPU erhebt (*raise*) eine Ausnahme
  - die Operation wird abgefangen (*trap*) bzw. unterbrochen (*interrupt*)

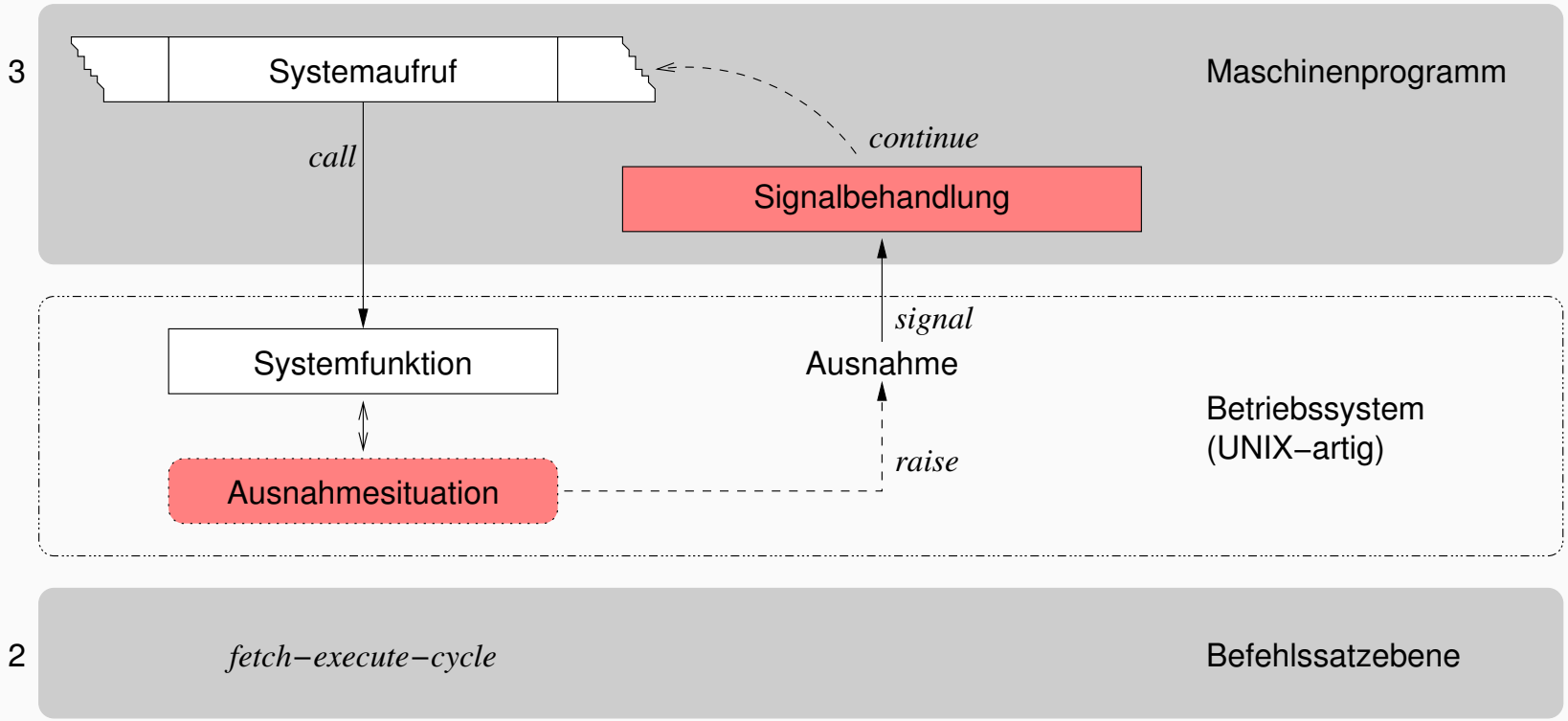
Sonderfallbehandlung I



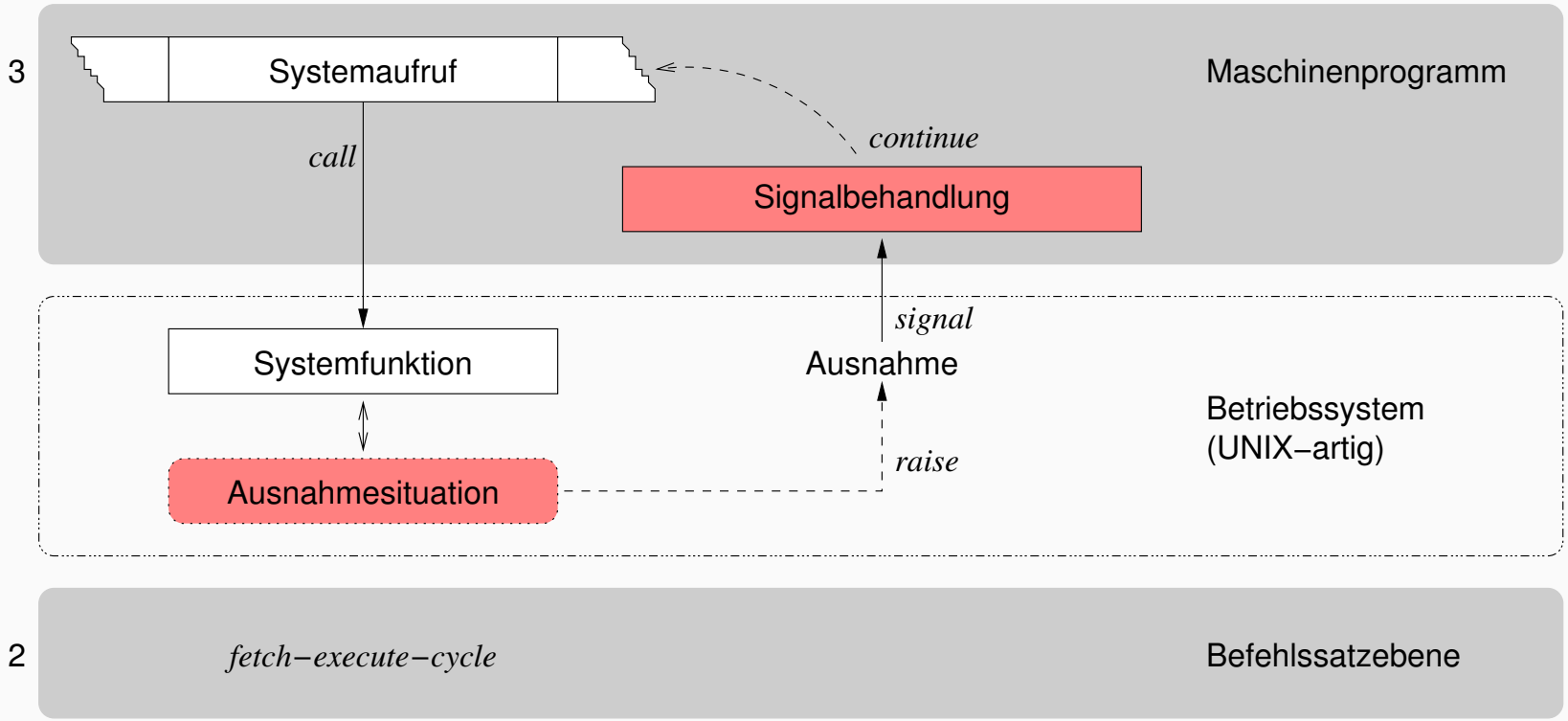
- die Ausnahmebehandlung erfolgt durch das Betriebssystem, das dazu durch die CPU aktiviert wird
  - ggf. wird die CPU instruiert, die Operation wieder aufzunehmen

Sonderfallbehandlung II

Ebene$_3$ $\leftrightarrow$ Betriebssystem

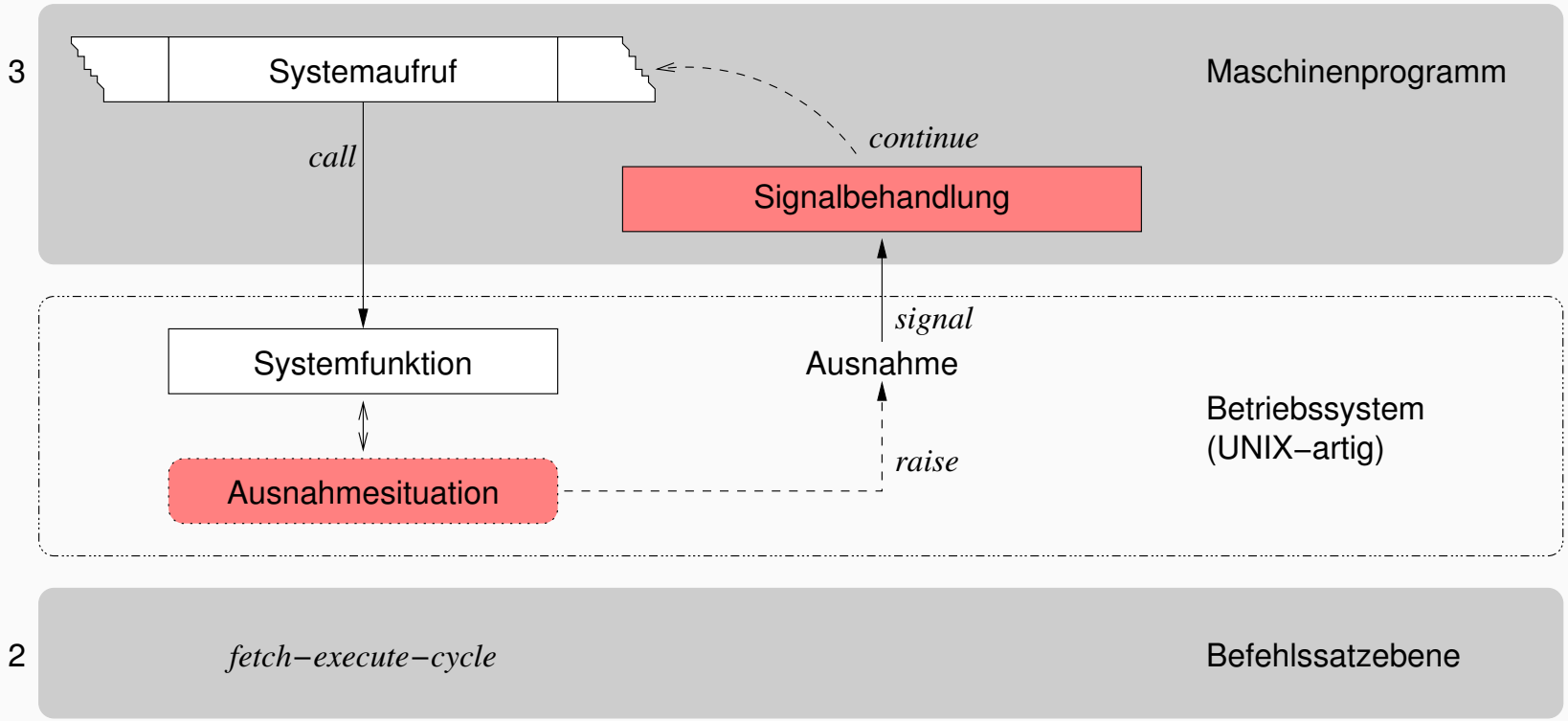


Sonderfallbehandlung II



- bei Ausführung der Systemfunktion tritt eine Ausnahmesituation auf, das Betriebssystem erhebt (*raise*) eine Ausnahme
  - die auf ein Signal abgebildet und zur Behandlung hoch gereicht wird

Sonderfallbehandlung II



- die Signalbehandlung erfolgt im Kontext des Maschinenprogramms, sie setzt am Ende die Ausführung des Maschinenprogramms fort

Gliederung

Resümee

Resümee

- Rechensysteme zeigen eine bestimmte innere **Schichtenstruktur**
  - die **semantische Lücke** zwischen Anwendungsprogramm und Hardware
  - die Kluft zwischen gedanklich Gemeintem und sprachlich Geäußertem

Resümee

- jedes Rechensystem ist als **Mehrebenenmaschine** ausgeprägt
  - eine Hierarchie virtueller Maschinen: **Interpretation** und **Übersetzung**
  - **Demarkationslinie** bzw. ein grundlegender Bruch zwischen Ebene$_{[3,4]}$
    - Methode der Abbildung, Art der Programmierung, Natur der Sprache
  - Abbildung der Schichten und Zeitpunkte der Abbildungsvorgänge
    - Betriebssysteme entvirtualisieren zur Laufzeit
  - Kunst der kleinen Schritte: semantische Lücke schrittweise schließen

Resümee

- **Ausnahmesituationen** bilden Ebenenübergänge „von unten nach “
  - im Sonderfall bei der Programmausführung kooperieren die Maschinen
  - Analogie zwischen Betriebssystem und CPU: abstrakter/realer Prozessor

Resümee

↪ ergänzend dazu zeigt der Anhang weitere **Interpretersysteme**

- **Virtualisierungssystem** realisiert als VMM (*virtual machine monitor*)

Resümee

…virtuelle Maschinen existieren vor oder zur Programmlaufzeit

- jedes Rechensystem ist als **Mehrebenenmaschine** ausgeprägt
  - eine Hierarchie virtueller Maschinen: **Interpretation** und **Übersetzung**

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Apple Computer, Inc.:
Rosetta.
In: *Universal Binary Programming Guidelines*.
Apple Computer, Inc., Jun. 2006 (Appendix A), S. 65–74

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

[3] Connectix Corp.:
***Connectix Virtual PC*.**
Press Release, Apr. 1997

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

Literaturverzeichnis (2)

[5] Fey, D. :
Hardwarenahe Programmierung in Assembler.
In: Lehrstuhl Informatik 3 (Hrsg.): *Grundlagen der Rechnerarchitektur und -organisation.*
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 2

[6] Goldberg, R. P.:
Architectural Principles for Virtual Computer Systems / Harvard University, Electronic Systems Division.
Cambridge, MA, USA, Febr. 1973 (ESD-TR-73-105). –
PhD Thesis

[7] Habermann, A. N. ; Flon, L. ; Cooprider, L. W.:
Modularization and Hierarchy in a Family of Operating Systems.
In: *Communications of the ACM* 19 (1976), Mai, Nr. 5, S. 266–272

Literaturverzeichnis (3)

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

[9] Robin, J. S. ; Irvine, C. E.:
Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor.
In: *Proceedings of 9th USENIX Security Symposium (SSYM'00)*, USENIX Association, 2000, S. 1–16

[10] Schröder-Preikschat, W. :
Virtuelle Maschinen.
Sept. 2013. –
Eingeladener Vortrag, INFORMATIK 2013, Workshop „Virtualisierung: gestern, heute und morgen“, Koblenz

Literaturverzeichnis (4)

[11] Smith, J. E. ; Nair, R. :
Virtual Machines: Versatile Platforms for Systems and Processes.
Morgan Kaufmann Publishers Inc., 2005. –
656 S. –
ISBN 9781558609105

[12] Tanenbaum, A. S.:
Multilevel Machines.
In: Structured Computer Organization[13], Kapitel 7, S. 344–386

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

[14] `http://www.hyperdictionary.com/computing/semantic+gap`

Anhang

Interpretersysteme

Architektonische Prinzipien virtueller Rechnersysteme

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

  real
  - beschränkt auf Ebene$_2$, nämlich die **physische CPU** (z.B. x86)

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

virtuell

- für beide jeweils durch ein spezifisches Programm in **Software**
- im Falle von Ebene$_3$ das **Betriebssystem** (nur partiell)
- bezüglich Ebene$_2$ ein **Virtualisierungssystem** (total/partiell)

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren
  - gelegentlich ist aber auch **Binärübersetzung** anzufinden (z.B. [1])

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

- dabei interpretiert das Virtualisierungssystem alle oder nur einen Teil der Befehle der Programme der virtuellen Maschine

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

- dabei interpretiert das Virtualisierungssystem alle oder nur einen Teil der Befehle der Programme der virtuellen Maschine

  total
  - als **Emulator** der eigenen oder einer fremden realen Maschine [3]
  - „*complete software interpreter machine*“ (CSIM, [6, S. 21])

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

- dabei interpretiert das Virtualisierungssystem alle oder nur einen Teil der Befehle der Programme der virtuellen Maschine

partiell

- als ***virtual machine monitor*** (VMM, [6, S. 21]), Typ I oder II
- der nur „sensitive Befehle“ abfängt und (in SW) emuliert

Architektonische Prinzipien virtueller Rechnersysteme

- Schichten der Ebene$_{[2,3]}$ werden durch reale oder virtuelle Maschinen implementiert, die normalerweise als **Interpreter** fungieren

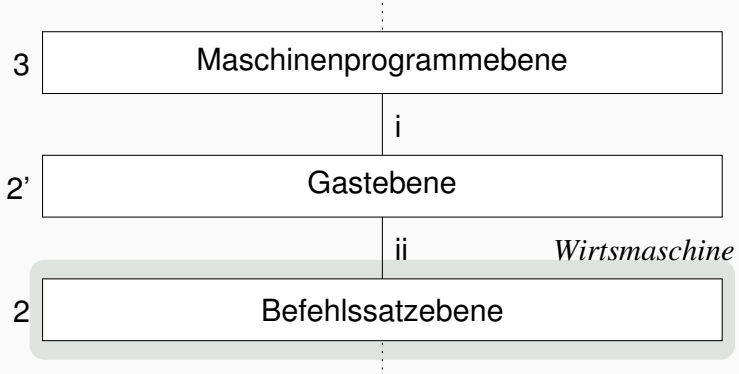
- dabei interpretiert das Virtualisierungssystem alle oder nur einen Teil der Befehle der Programme der virtuellen Maschine

- je nach VMM ist der Übereinstimmungsgrad von virtueller und realer Maschine (Wirt) möglicherweise unterschiedlich [6, S. 17]
  - bei **Selbstvirtualisierung** besteht 100% funktionale Übereinstimmung
  - im Gegensatz zur **Familienvirtualisierung**, bei der die virtuelle Maschine lediglich Mitglied der Rechnerfamilie der Wirtsmaschine ist

Wächter virtueller Maschinen

Wächter virtueller Maschinen

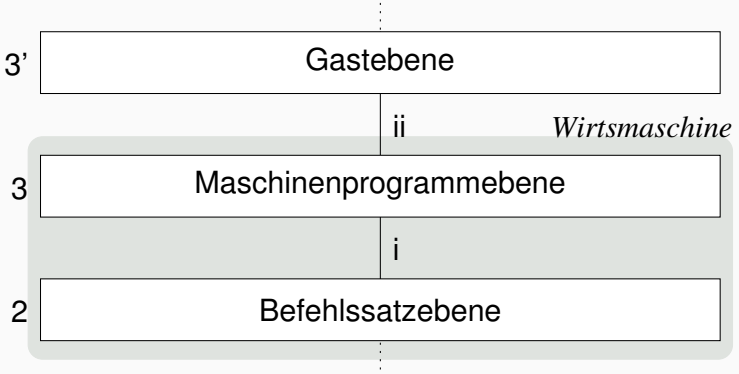
- Typ I VMM



  - läuft auf einer „nackten“ Wirtsmaschine
  - unter keinem Betriebssystem

Wächter virtueller Maschinen

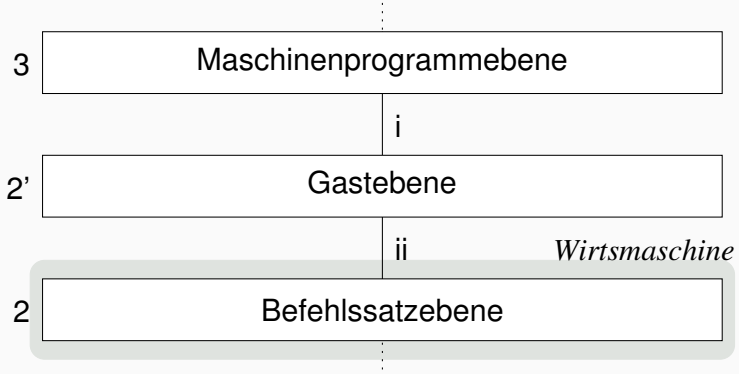
- Typ II VMM



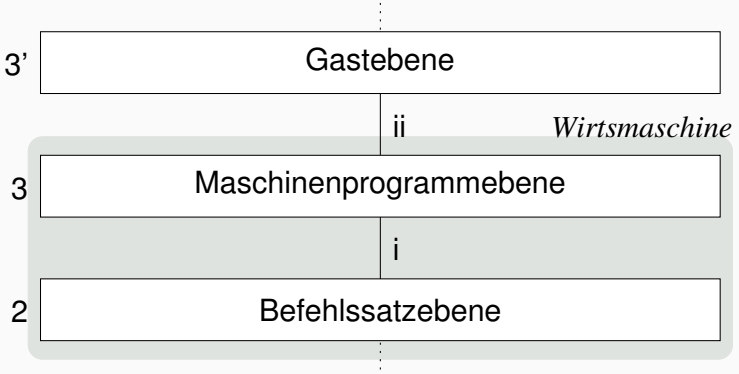
  - läuft auf einer erweiterten Wirtsmaschine
  - unter dem Wirtsbetriebssystem

Wächter virtueller Maschinen

- Typ I VMM



- Typ II VMM



- beiden gemeinsames Operationsprinzip ist die **Teilinterpretation**:
  - i durch das Betriebssystem (Typ I) bzw. Wirtsbetriebssystem (Typ II)
  - ii durch den VMM

Wächter virtueller Maschinen

- Gegenstand der Teilinterpretation sind **sensitive Befehle**
  - jeder Befehl, dessen direkte Ausführung durch die VM nicht tolerierbar ist
    - privilegierte Befehle ausgeführt im unprivilegierten Modus $\leadsto$ *Trap*
    - aber leider auch unprivilegierte Befehle mit kritischen Seiteneffekten

Virtualisierbare Reale Maschine

Virtualisierbare Reale Maschine

- typische Anforderungen an die Befehlssatzebene [6, S. 47–53]:

1. annähernd äquivalente Ausführung der meisten unprivilegierten Befehle im System- und Anwendungsmodus des Rechnersystems
2. Schutz von Programmen, die im Systemmodus ausgeführt werden
3. Abfangvorrichtung („Falle“, *trap*) für **sensitive Befehle**:
   a. Änderung/Abfrage des Systemzustands (z.B. Arbeitsmodus des Rechners)
   b. Änderung/Abfrage des Zustands reservierter Register oder Speicherstellen
   c. Referenzierung des (für 2. erforderlichen) Schutzsystems
   d. Ein-/Ausgabe

Virtualisierbare Reale Maschine

- **unprivilegierte sensitive Befehle** sind kritisch

Virtualisierbare Reale Maschine

2. Schutz von Programmen, die im Systemmodus ausgeführt werden
3. Abfangvorrichtung („Falle“, *trap*) für **sensitive Befehle**:
   - b Änderung/Abfrage des Zustands reservierter Register oder Speicherstellen
   - c Referenzierung des (für 2. erforderlichen) Schutzsystems

- **unprivilegierte sensitive Befehle** sind kritisch, Intel Pentium [9]:
  - **verletzt 3.b**
    - `SGDT`, `SIDT`, `SLDT`; [`SMSW`;] `POPF`, `PUSHF`
  - **verletzt 3.c**
    - `LAR`, `LSL`, `VERR`, `VERW`; `POP`, `PUSH`; `STR`, `MOVE`
    - `CALL`, `INT n`, `JMP`, `RET`

Virtualisierbare Reale Maschine

- **unprivilegierte sensitive Befehle** sind kritisch
  - bei Vollvirtualisierung (VMware), ist **partielle Binärübersetzung** eine Lösung, oder eben **Paravirtualisierung** (VM/370, Denali, Xen)
  - in beiden Fällen sind aber Softwareänderungen unvermeidbar, entweder am Maschinenprogramm oder am Betriebssystem

Transparenz für das Betriebssystem

Transparenz für das Betriebssystem

- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent
  - bis auf Zeitmessung hat das Betriebssystem sonst keine Möglichkeit, in Erfahrung zu bringen, ob es eine virtuelle oder reale Maschine betreibt
  - vorausgesetzt der Abwesenheit (unprivilegierter) sensitiver Befehle und damit der Nichterfordernis von Binärübersetzung

↪ Betriebssystem und VMM wissen nicht voneinander

Transparenz für das Betriebssystem

- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent

- anders verhält es sich mit **Paravirtualisierung** $\leadsto$ intransparent
  - Grundidee dabei ist, dass das Betriebssystem gezielt mit dem VMM in Interaktion tritt und bewusst auf Transparenz verzichtet

Transparenz für das Betriebssystem

- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent

- anders verhält es sich mit **Paravirtualisierung** $\leadsto$ intransparent
  - Grundidee dabei ist, dass das Betriebssystem gezielt mit dem VMM in Interaktion tritt und bewusst auf Transparenz verzichtet
  - Hintergrund ist die **Deduplikation** von Funktionen aber auch Daten, die sowohl im Betriebssystem als auch im VMM vorhanden sein müssen
    - Betriebsmittelverwaltung, Gerätetreiber, Prozessorsteuerung, ...

Transparenz für das Betriebssystem

- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent

- anders verhält es sich mit **Paravirtualisierung** $\leadsto$ intransparent
  - Grundidee dabei ist, dass das Betriebssystem gezielt mit dem VMM in Interaktion tritt und bewusst auf Transparenz verzichtet
  - Hintergrund ist die **Deduplikation** von Funktionen aber auch Daten, die sowohl im Betriebssystem als auch im VMM vorhanden sein müssen
    - Betriebsmittelverwaltung, Gerätetreiber, Prozessorsteuerung, …
  - weiterer Aspekt ist die damit einhergehende Reduktion von Gemeinkosten (*overhead*) durch Wegfall der Teilinterpretation des Betriebssystems

Transparenz für das Betriebssystem

- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent

- anders verhält es sich mit **Paravirtualisierung** $\leadsto$ intransparent
  - Grundidee dabei ist, dass das Betriebssystem gezielt mit dem VMM in Interaktion tritt und bewusst auf Transparenz verzichtet
  - Hintergrund ist die **Deduplikation** von Funktionen aber auch Daten, die sowohl im Betriebssystem als auch im VMM vorhanden sein müssen
    - Betriebsmittelverwaltung, Gerätetreiber, Prozessorsteuerung, ...
  - weiterer Aspekt ist die damit einhergehende Reduktion von Gemeinkosten (*overhead*) durch Wegfall der Teilinterpretation des Betriebssystems
  - in dem Zusammenhang werden im Betriebssystem ursprünglich enthaltene sensitive Befehle als Elementaroperationen des VMM repräsentiert

Transparenz für das Betriebssystem

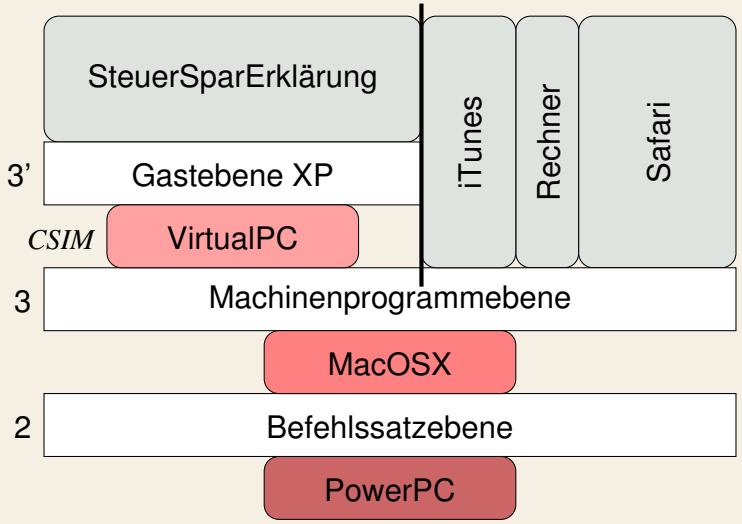
- **Vollvirtualisierung** (Selbstvirtualisierung) ist funktional transparent

- anders verhält es sich mit **Paravirtualisierung** $\rightsquigarrow$ intransparent
  - Grundidee dabei ist, dass das Betriebssystem gezielt mit dem VMM in Interaktion tritt und bewusst auf Transparenz verzichtet

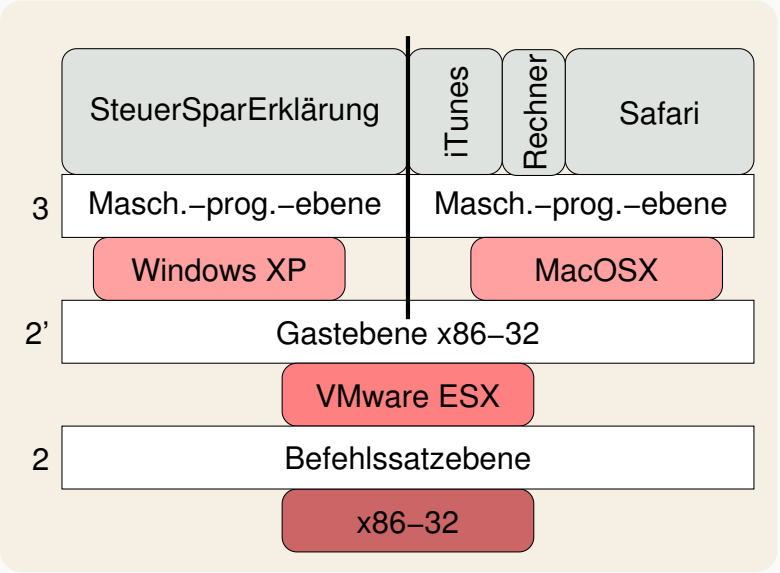
$\hookrightarrow$ Betriebssystem und VMM gehen eine Art **Symbiose** ein

Anwendungsbeispiele

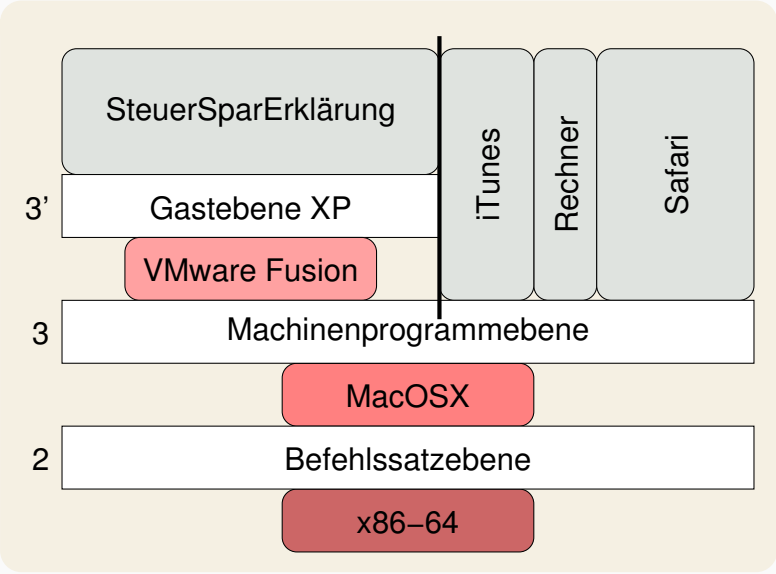
VirtualPC ≡ CSIM



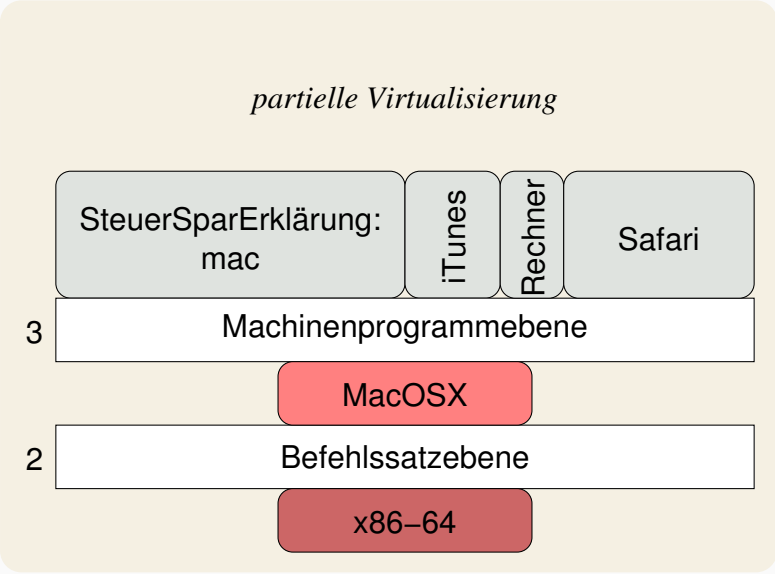
Anwendungsbeispiele



Anwendungsbeispiele

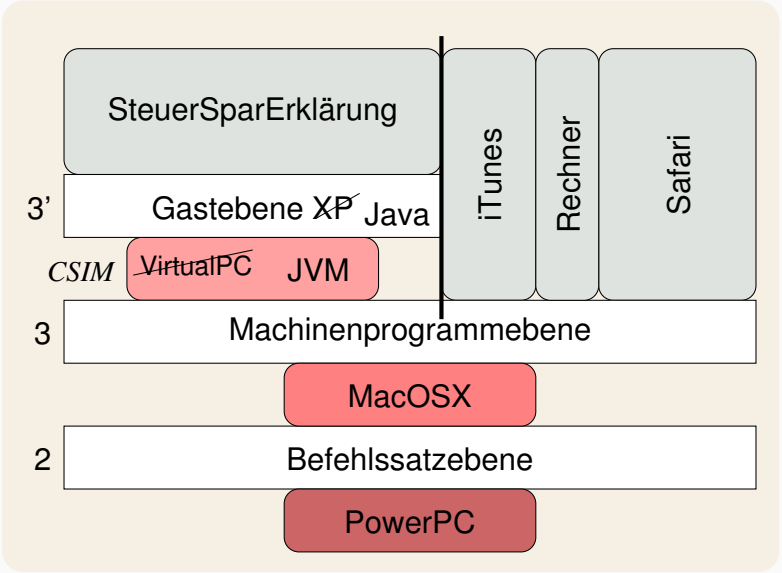


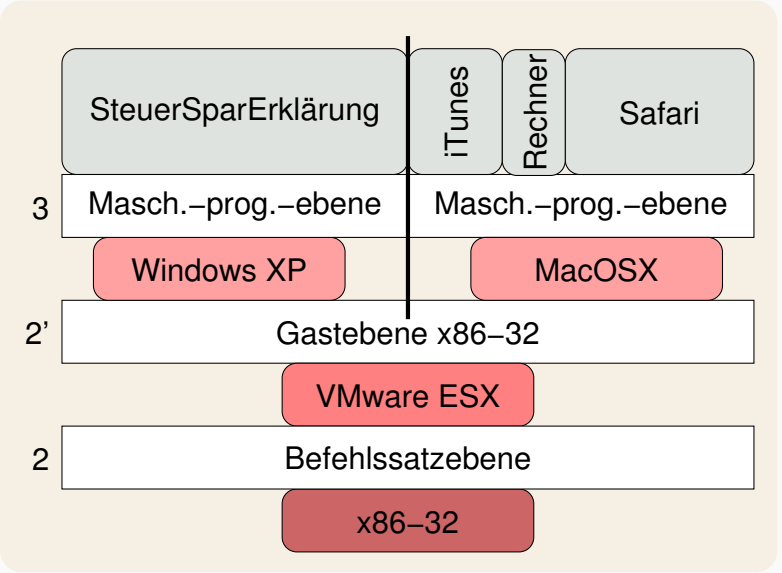
Anwendungsbeispiele

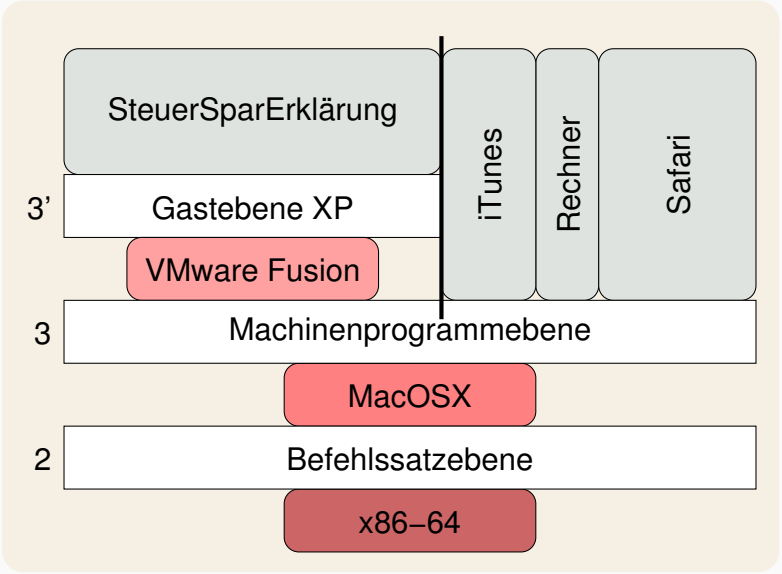


Anwendungsbeispiele

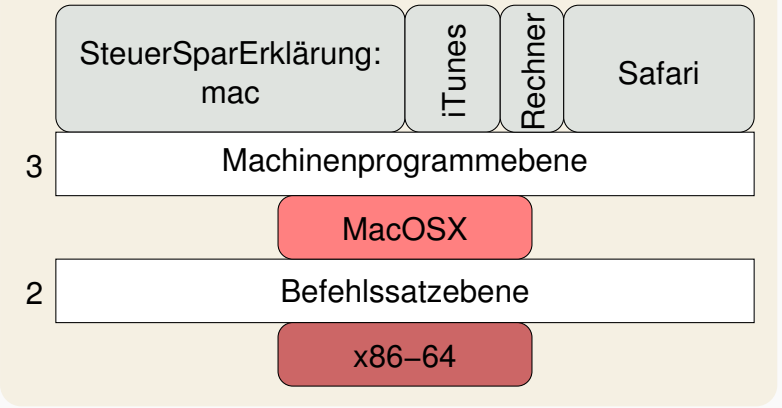
JVM ≡ VirtualPC ≡ CSIM







partielle Virtualisierung



Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – V.2 Rechnerorganisation: Maschinenprogramme

20. Juni 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

- Einführung
  - Hybrid
- Programmhierarchie
  - Hochsprachenkonstrukte
  - Assembliersprachenanweisungen
  - Betriebssystembefehle
- Organisationsprinzipien
  - Funktionen
  - Komponenten
- Zusammenfassung

Gliederung

Lehrstoff

- Maschinenprogramm als Entität einer **hybriden Schicht** verstehen
  - Instruktionen an die Befehlssatzebene, die direkt ausgeführt werden
  - Instruktionen an das Betriebssystem, die partiell interpretiert werden
- Ebene$_{[2,3]}$ als **Programmhierarchie** virtueller Maschine vertiefen
  - indem exemplarisch für x86 und Linux das Zusammenspiel dieser Maschinen zur Diskussion gestellt wird
  - dabei die prinzipielle Funktionsweise von Systemaufrufen erkennen
- **Grobstruktur** von Maschinenprogrammen im Ansatz kennenlernen
  - mit dem Laufzeitsystem und den Systemaufrufstümpfen als zwei zentrale Bestandteile der Systemsoftware
  - inklusive Anwendungsroutinen zusammengebunden zum **Lademodul**

Auch wenn wir die Programmbeispiele symbolisch dargestellt sehen, ist zu beachten, dass Maschinenprogramme letztlich numerischer Natur sind. (vgl. [3, S. 18])

Einführung

Hybrid

Hybride Schicht in einem Rechensystem

- Maschinenprogramme enthalten zwei Sorten von Befehlen:
  i **Maschinenbefehle** der Befehlssatzebene (ISA)
    - normalerweise direkt interpretiert durch die Zentraleinheit[1]
    - ausnahmsweise partiell interpretiert durch das Betriebssystem
  ii **Systemaufrufe** an das Betriebssystem
    - normalerweise partiell interpretiert durch das Betriebssystem

Hybrid (lat. *hybrida* Bastard, Mischling, Frevelkind)[a]

[a] gr. *hýbris* Übermut, Anmaßung

„etwas Gebündeltes, Gekreuztes oder Gemischtes“ [6]

- ein System das zwei Techniken miteinander kombiniert:
  i Interpretation von Programmen der Befehlssatzebene
  ii partielle Interpretation von Maschinenprogrammen
- ein Maschinenprogramm ist **Hybridsoftware**, die auf Ebene$_{[2,3]}$ läuft

[1] *central processing unit*, CPU

Betriebssystem ≡ Programm der Befehlssatzebene

- ein Betriebssystem implementiert die Maschinenprogrammebene
  - es zählt damit selbst nicht zur Klasse der Maschinenprogramme
  - es setzt normalerweise keine Systemaufrufe (an sich selbst) ab
  - es unterbricht sich normalerweise niemals von selbst
- gleichwohl sollten Betriebssysteme es zulassen, in der Ausführung eigener Programme unterbrochen werden zu können
  - nicht durch Systemaufrufe
  - aber durch *Traps* oder *Interrupts* — **Ausnahmen**

↪ sie interpretieren die eigenen Programme nur eingeschränkt partielle

Teilinterpretation von Betriebssystemprogrammen

*Bewirkt **indirekt rekursive Programmausführungen** im Betriebssystem[a] und erfordert daher die Fähigkeit zum **Wiedereintritt** (re-entrance). Je nach Operationsprinzip[b] des Betriebssystems ist dies zulässig oder (temporär) unzulässig.*

[a] ausgelöst durch synchrone/asynchrone Unterbrechungen
[b] nichtblockierende/blockierende Synchronisation

Gliederung

Maschinensprache(n)

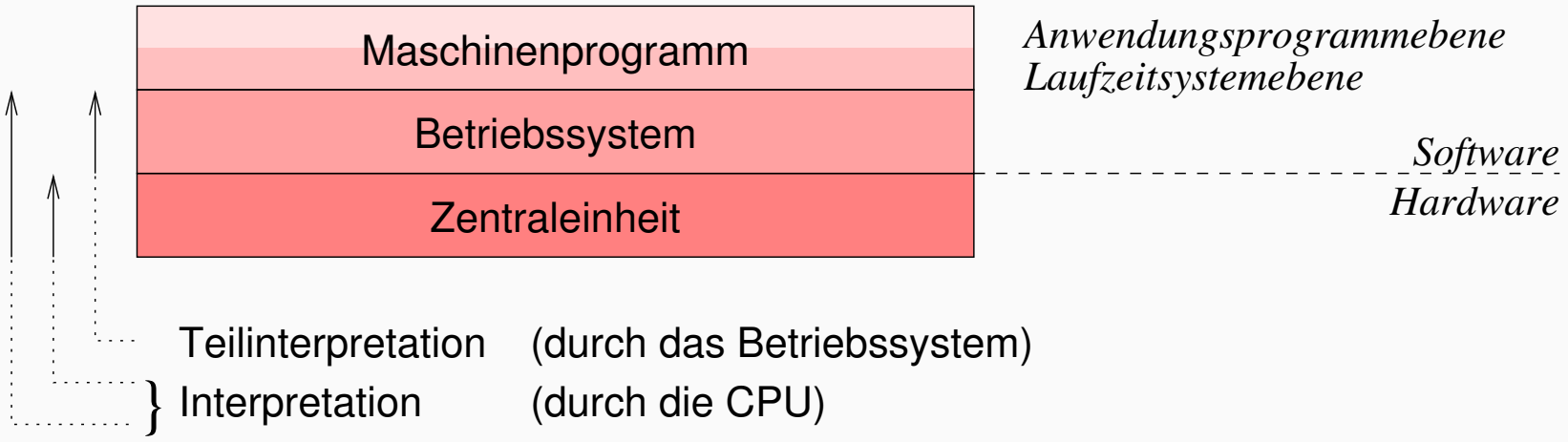
- Maschinenprogramme setzen sich aus Anweisungen zusammen, die ohne Übersetzung von einem Prozessor ausführbar sind
  - gleichwohl werden sie (normalerweise) durch Übersetzung generiert
    - nahezu ausschließlich automatisch: Kompilierer, Assemblierer, Binder
    - in seltenen Fällen manuell: **nativer Kode** (*native code*)[2]
  - sie repräsentieren sich technisch als **Lademodul** (*load module*)
    - erzeugt durch Dienstprogramme (*utilities*): `gcc(1)`, `as(1)`, `ld(1)`
    - geladen, verarbeitet und entsorgt durch Betriebssysteme
  - d.h., als **ausführbares Programm** und in numerischer Form

- Grundlage für die Entwicklung von Maschinenprogrammen bilden Hoch- und Assembliersprachen, und zwar für jede Art Software:
  - Anwendungsprogramme, Laufzeitsysteme und Betriebssysteme
  - symbolisch repräsentiert auf Ebene$_{[4,5]}$, numerisch auf Ebene$_3$

[2] Binärkode des realen Prozessors, auch: Maschinenkode.

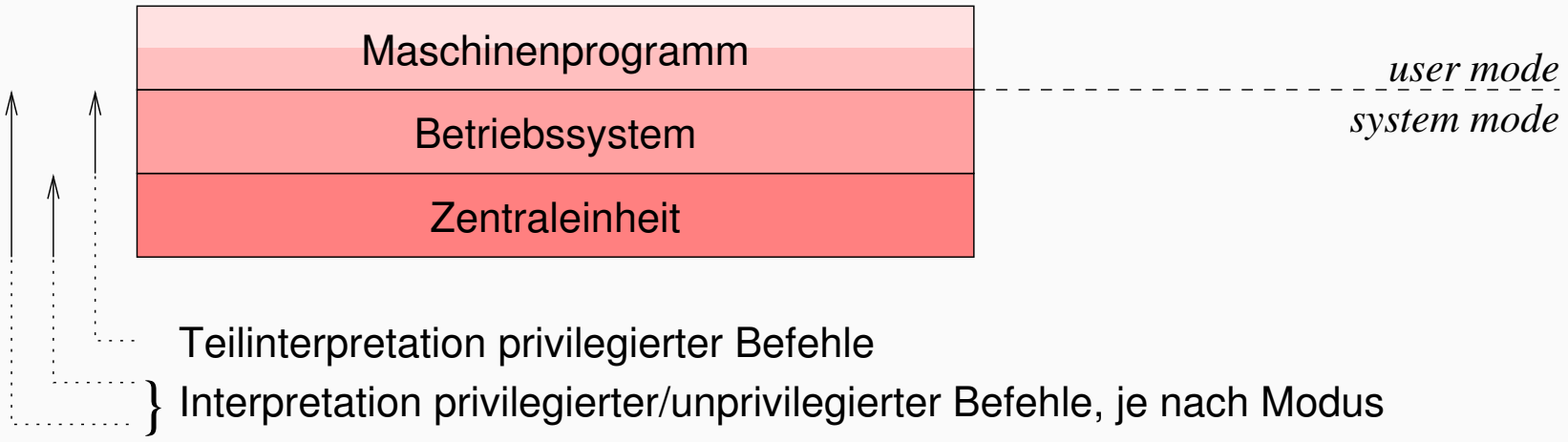
„Triumvirat“

…zur Ausführung von Anwendungsprogrammen



- Maschinenprogramm = Anwendungsprogramm + Laufzeitsystem
  - beide Teilebenen liegen im selben **Adressraum**, der zudem (logisch) per **Speicherschutz** von anderen Adressräumen isoliert ist
  - einfache Unterprogrammaufrufe aktivieren das Laufzeitsystem
- Ausführungsplattform = Betriebssystem + Zentraleinheit (CPU)
  - Verarbeitung eines Maschinenprogramms durch einen Prozessor, der in Hard- und Software implementiert vorliegt
  - komplexe **Systemaufrufe** (*system calls*) aktivieren das Betriebssystem

Benutzer-/Systemgrenze



- Maschinenprogramm = Benutzerebene (*user level*, *user space*)
  - eingeschränkter Umgang mit Merkmalen der Befehlssatzebene in Bezug auf Maschinenbefehle, Hardwarekomponenten und Peripheriegeräte
  - nur **unprivilegierte Operationen** werden direkt ausgeführt, privilegierte Operationen erfordern den **Moduswechsel** $\rightsquigarrow$ Systemaufruf
- Ausführungsplattform = Systemebene (*system level*, *kernel space*)
  - uneingeschränkter Umgang mit den Merkmalen der Befehlssatzebene
  - alle Maschinenbefehle werden direkt ausgeführt, alle Operationen gültig

Programmhierarchie

Hochsprachenkonstrukte

Anwendungsprogramm: Hochsprache

C

- ein auf Ebene$_5$ symbolisch repräsentiertes Programm der Ebene$_3$:

```
1  void echo() {
2      char c;
3      while (read(0, &c, 1) == 1) write(1, &c, 1);
4  }
```

echo.c

Funktion `read(2)` *überträgt ein Zeichen von Standardeingabe (*0*) an die Arbeitsspeicheradresse der lokalen Variablen* 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*.*

Programmhierarchie

Assembliersprachenanweisungen

Anwendungsprogramm: Assembliersprache

- dasselbe Programm (S. 13) symbolisch repräsentiert auf Ebene 4:[3]

```
.file "echo.c"
.text
.p2align 4,,15
.globl   echo
.type echo, @function
echo:
pushl %ebx
subl  $40, %esp
leal  28(%esp), %ebx
jmp   .L2
.p2align 4,,7
.p2align 3
.L3:
movl  $1, 8(%esp)
movl  %ebx, 4(%esp)
movl  $1, (%esp)
call  write
.L2:
movl  $1, 8(%esp)
movl  %ebx, 4(%esp)
movl  $0, (%esp)
call  read
cmpl  $1, %eax
je    .L3
addl  $40, %esp
popl  %ebx
ret
```

- **unaufgelöste Referenzen** der Systemfunktionen `read(2)` und `write(2)` werden vom Binder `ld(1)` aufgelöst $\mapsto$ `libc.a`

[3] Übersetzung von `echo.c` mit `-S` liefert `echo.s`

Programmhierarchie

Betriebssystembefehle

Laufzeitsystem: symbolischer Maschinenkode

- **Stümpfe** der Systemfunktionen auf Ebene 3, symbol. aufbereitet:

```
1  read:
2    push %ebx
3    movl 16(%esp),%edx
4    movl 12(%esp),%ecx
5    movl 8(%esp),%ebx
6    mov  $3,%eax
7    int  $0x80
8    pop  %ebx
9    cmp  $-4095,%eax
10   jae  __syscall_error
11   ret
```

```
12 write:
13   push %ebx
14   movl 16(%esp),%edx
15   movl 12(%esp),%ecx
16   movl 8(%esp),%ebx
17   mov  $4,%eax
18   int  $0x80
19   pop  %ebx
20   cmp  $-4095,%eax
21   jae  __syscall_error
22   ret
```

  - nach Kompilation[4] Verwendung der `disassemble`-Operation von `gdb(1)`

- **Systemaufruf** wird durch `int $0x80` (*software interrupt*) ausgelöst
  - Operationskode in `%eax`
  - Parameter in `%ebx`, `%ecx` und `%edx`
  - Resultat in `%eax` zurück

```
23 __syscall_error:
24   neg %eax
25   mov %eax,errno
26   mov $-1,%eax
27   ret
28
29   .comm errno,16
```

[4] Übersetzung von `echo.c` inklusive abschließender Bindung.

Betriebssystem: Assembliersprache

- **Systemaufraufzuteiler** (*system call dispatcher*):
  - ein auf Ebene$_4$ symbolisch repräsentiertes Programm der Ebene$_2$
  - `kernel-source-2.4.20/arch/i386/kernel/entry.S` (Auszug)

Prolog

```
1   system_call:
2     pushl %eax
3     cld
4     pushl %es
5     pushl %ds
6     pushl %eax
7     pushl %ebp
8     pushl %edi
9     pushl %esi
10    pushl %edx
11    pushl %ecx
12    pushl %ebx
13    ...
```

Abruf und Ausführung

```
14    ...
15    cmpl  $(NR_syscalls),%eax
16    jae   badsys
17    call  *sys_call_table(,%eax,4)
18    movl  %eax,24(%esp)
19  ret_from_sys_call:
20    ...
21    jmp   restore_all
22  badsys:
23    movl  $-ENOSYS,24(%esp)
24    jmp   ret_from_sys_call
```

Epilog

```
25  restore_all:
26    popl  %ebx
27    popl  %ecx
28    popl  %edx
29    popl  %esi
30    popl  %edi
31    popl  %ebp
32    popl  %eax
33    popl  %ds
34    popl  %es
35    addl  $4,%esp
36    iret
```

4–12 ▪ Sicherung des Prozessorzustands des Maschinenprogramms

7–12 ▪ Übernahme der aktuellen Parameter von Systemaufrufen

15–18 ▪ Überprüfung des Operationskodes und Aufruf der Systemfunktion

26–34 ▪ Wiederherstellung des gesicherten Prozessorzustands

36 ▪ Wiederaufnahme der Ausführung des Maschinenprogramms

Betriebssystem: Interpreter

- **Befehlsabruf- und -ausführungszyklus** (*fetch-execute cycle*) zur Ausführung von Systemaufrufen

1. Prozessorstatus des unterbrochenen Programms sichern . . . . . Prolog
   - Aufforderung der CPU zur Teilinterpretation nachkommen
2. Systemaufruf interpretieren . . . . . Abruf und Ausführung
   i. Systemaufrufnummer (Operationskode) abrufen
   ii. auf Gültigkeit überprüfen und ggf. Fehlerbehandlung auslösen
   iii. bei gültigem Operationskode, zugeordnete Systemfunktion ausführen
3. Prozessorstatus wiederherstellen und zurückspringen . . . . . Epilog
   - Beendigung der Teilinterpretation der CPU „mitteilen“
   - Ausführung des unterbrochenen Programms wieder aufnehmen

- mangels „echter“ **Systemimplementierungssprache**[5] ist hier in dem Kontext der Einsatz von Assembliersprache erforderlich
  - Teilinterpretation erfordert kompletten Zugriff auf den Prozessorstatus
  - dieser ist nicht mehr Teil des Programmiermodells einer Hochsprache

[5] Höhere Programmiersprache mit hardwarenahen Sprachelementen.

Betriebssystem: Hochsprache C

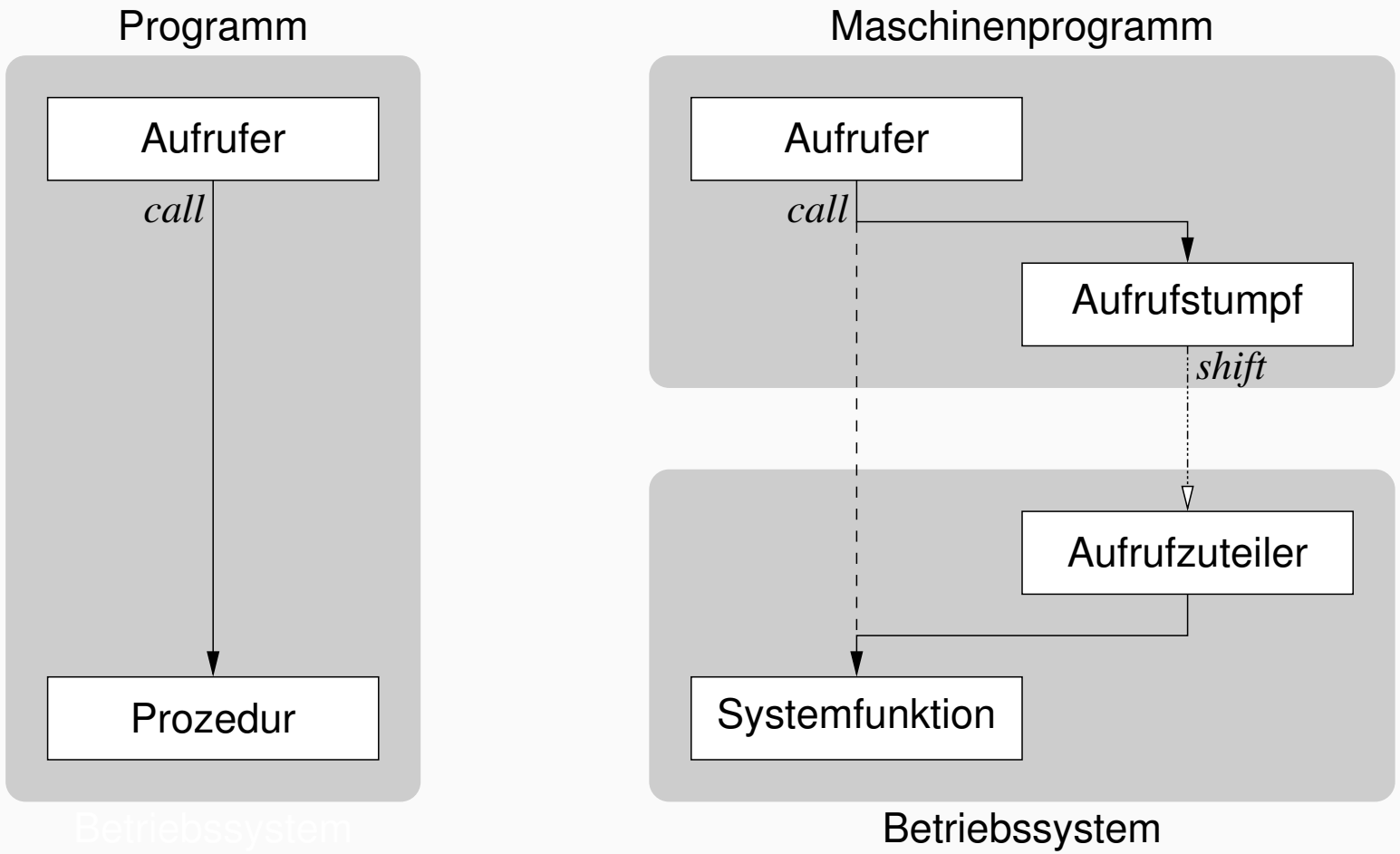
- ein auf Ebene$_5$ symbolisch repräsentiertes Programm der Ebene$_2$:
  - `kernel-source-2.4.20/fs/read_write.c` (Auszug)

```
1  asmlinkage
2  ssize_t sys_read(unsigned int fd, char *buf, size_t count) {
3      ssize_t ret;
4      struct file *file;
5
6      ret = -EBADF;
7      file = fget(fd);
8      if (file) {
9          ...
10     }
11     return ret;
12 }
13
14 asmlinkage ssize_t sys_write ...
```

- **Systemfunktion** (Implementierung) innerhalb des Betriebssystems
  - aktiviert durch `call *sys_call_table(,%eax,4)` (S. 18, Zeile 17)
  - **1** weist den Kompilierer an, Parameter auf dem Stapel zu übergeben[6]

[6] Standardmäßig werden die ersten Parameter der Systemfunktionen von Linux in Registern übergeben, für x86-32: `eax`, `ecx` und `edx`.

Prozedur- vs. Systemaufruf



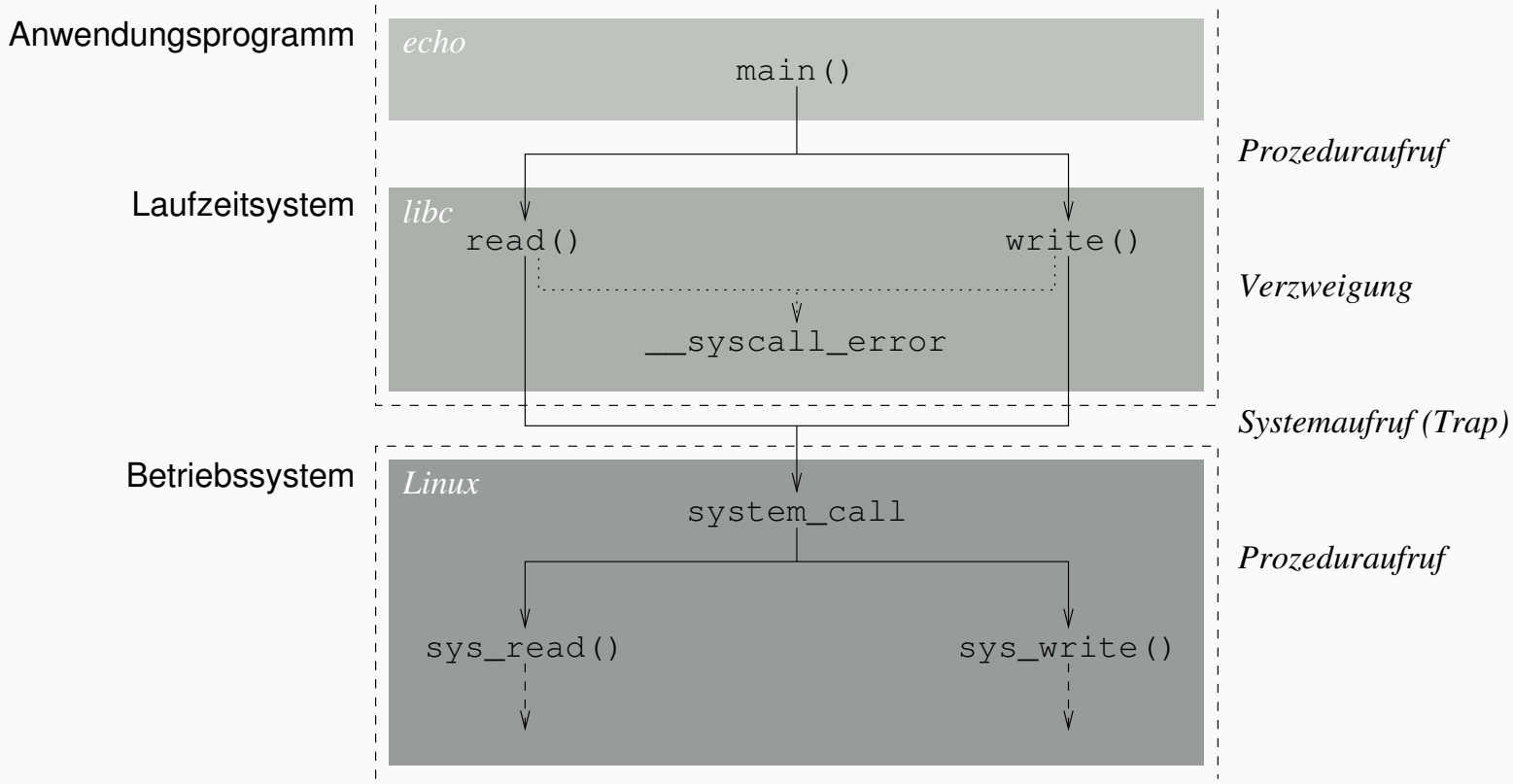
- Systemaufruf als adressraumübergreifender Prozeduraufruf
  - verlagert (*shift*) die weitere Prozedurausführung ins Betriebssystem

Gliederung

Organisationsprinzipien

Funktionen

Domänenübergreifende Aufrufhierarchie



- „obere“ Domäne (Ebene$_3$, $\sqcup$)
  - Anwendungsmodus
  - unprivilegiert (graduell)
  - räumlich isoliert (total)
  - transient (logisch)
- „untere“ Domäne (Ebene$_2$, $\sqcap$)
  - Systemmodus
  - privilegiert (graduell)
  - räumlich isoliert (partiell)
  - resident (logisch)

Systemaufrufschnittstelle (*system call interface*)[7]

```
1  read:
2    push %ebx
3    movl 16(%esp),%edx
4    movl 12(%esp),%ecx
5    movl 8(%esp),%ebx
6    mov  $3,%eax
7    int  $0x80
8    pop  %ebx
9    cmp  $-4095,%eax
10   jae  __syscall_error
11   ret
```

- „Grenzübergangsstelle“ **Aufrufstumpf**
  - einerseits erscheint ein Systemaufruf als normaler Prozeduraufruf
  - andererseits bewirkt der Systemaufruf einen Moduswechsel
- sorgt für **Ortstransparenz** (funktional)
  - die Lokalität der aufgerufenen Systemfunktion muss nicht bekannt sein

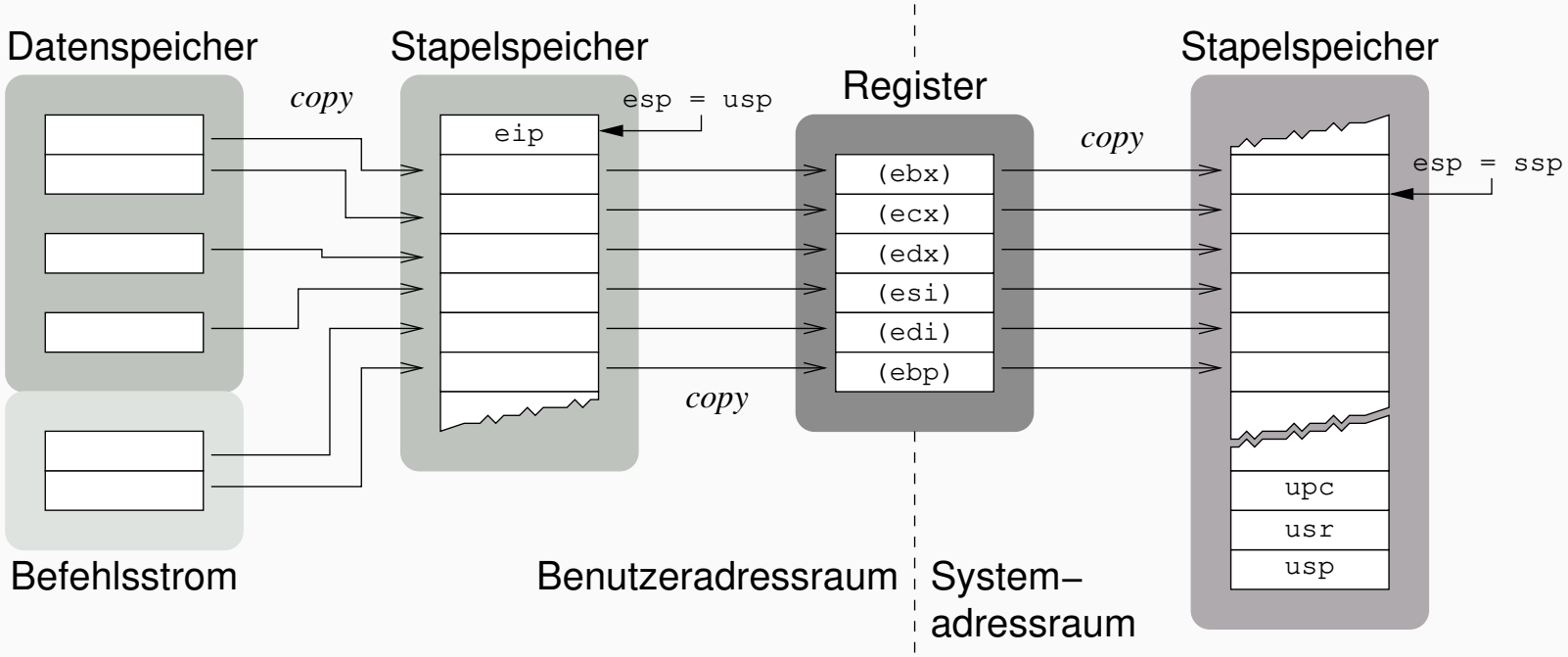
- Systemaufrufe sind **Prozedurfernaufrufe**, um **Prozessdomänen** in kontrollierter Weise zu überwinden

3–5	▪ tatsächliche Parameter (Argumente) in Registern übergeben
6	▪ Systemaufrufnummer (Operationskode) in Register übergeben
7	▪ Domänenwechsel ($\text{Ebene}_3 \mapsto \text{Ebene}_2$) auslösen
	▪ Aufruf abfangen (*trap*) und dem Betriebssystem zustellen
9–10	▪ Status überprüfen und ggf. Fehlerbehandlung durchführen

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

Parametertransfer: Linux

IA-32 passend



- Werteübergabe (*call by value*) für alle Parameter
  - Variable: Befehlsoperand ist Adresse im Datenspeicher inkl. Register
  - Direktwert: Bestandteil des Befehls im Befehlsstrom
- stark abhängig vom **Programmiermodell** der Befehlssatzebene[8]
  - die Registeranzahl bestimmt die Anzahl direkter Parameter
  - ggf. sind weitere Parameter indirekt über den Stapelzeiger zu laden

[8] …und der problemorientierten Programmiersprachenebene, des Kompilierers.

Laufzeitumgebung (*runtime environment*)[9]

- **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)

		`printf`	`scanf`			`perror`
`man(3)`		↓	↓			↓
		`putchar`	`getchar`	`malloc`	`free`	`errno`
		↓	↓	↓		
`man(2)`	`socket`	`write`	`read`	`sbreak`		

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

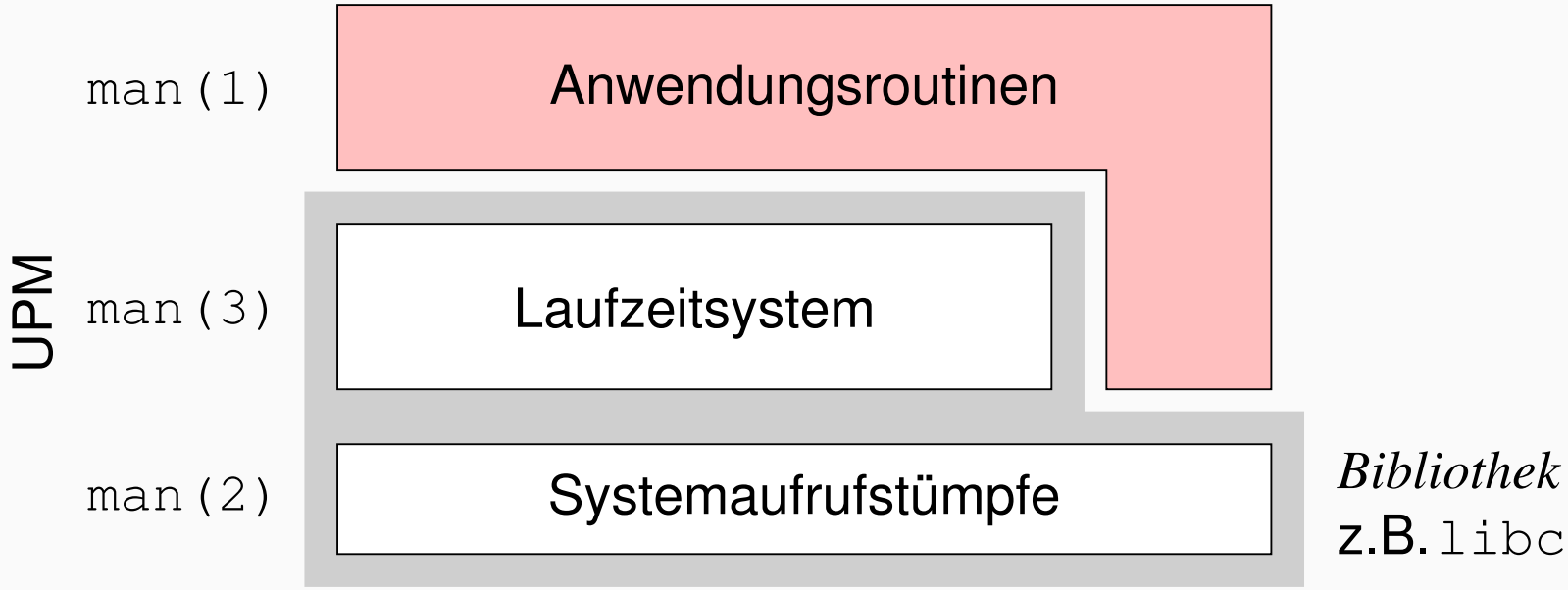
Ensemble problemspezifischer Prozeduren

- **Anwendungsroutinen** (des Rechners)
  - bei C/C++ die Funktion `main()` und anderes Selbstgebautes
  - setzen u.a. Betriebssystem- oder Laufzeitsystemaufrufe ab
- **Laufzeitsystemfunktionen** (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 `read(2)` und `write(2)`
  - setzen Aufrufe an das Betriebssystem ab
    - Systemaufruf $\mapsto$ Abfangstelle im Betriebssystem $\sim$ *Trap*
- bilden zusammengebunden das **Maschinenprogramm** (Lademodul)

Organisationsprinzipien

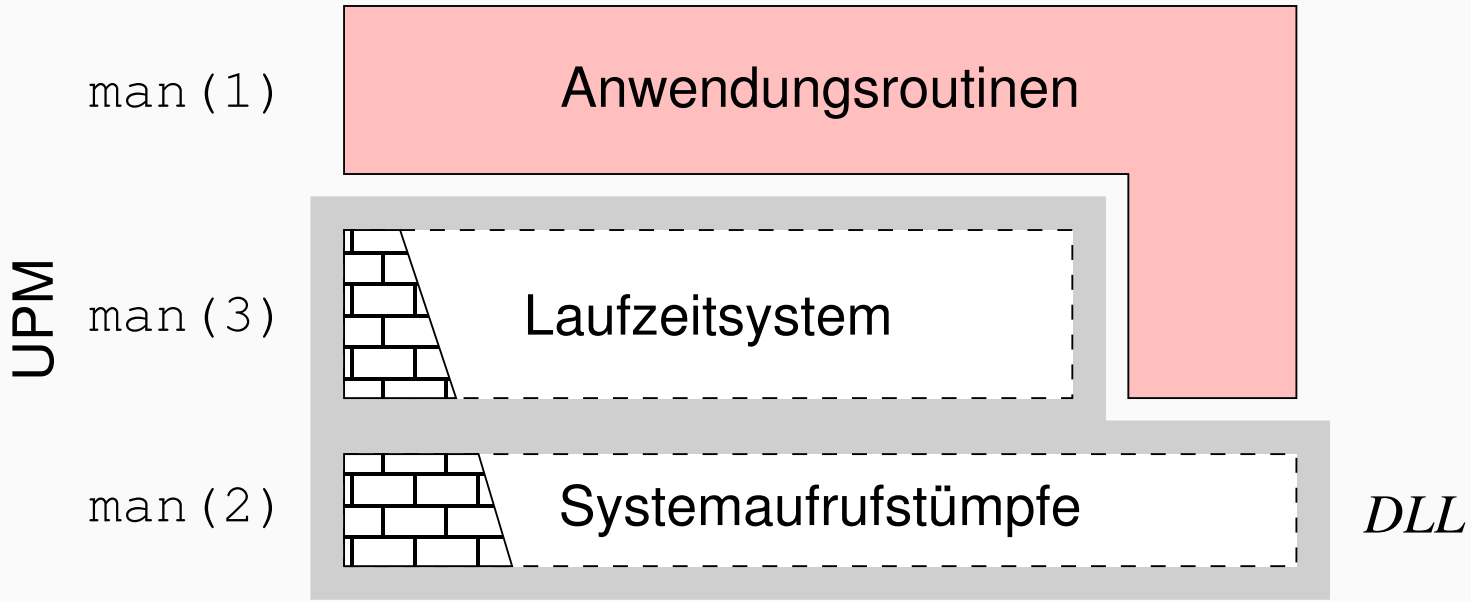
Komponenten

Grobstruktur von Maschinenprogrammen I



- statisch gebundenes Programm
  - zum Ladezeitpunkt des Programms sind alle Referenzen aufgelöst
    - Kompilierer und Assemblierer lösen lokale (interne) Referenzen auf
    - der Binder löst globale (`extern`, `.globl`) Referenzen auf
  - Schalter `-static` bei `gcc(1)` oder `ld(1)`
- Laufzeitüberprüfung von Bibliotheksreferenzen entfällt

Grobstruktur von Maschinenprogrammen II



- dynamisch gebundenes Programm
  - Bibliotheksfunktionen erst bei Bedarf (vom Betriebssystem) einbinden
    - Ebene$_{[2,3]}$ erkennt einen **Bindungsfehler** (*link trap*, Multics [4])
    - den ein **bindender Lader** (*linking loader*) im Betriebssystem behandelt
  - dynamische Bibliothek (*shared library*, *dynamic link library* (DLL))
- Laufzeitüberprüfung von Bibliotheksref. $\rightsquigarrow$ **Teilinterpretation**

Gliederung

Resümee

…Maschinenprogramme gibt es nicht ohne Betriebssystem

- Bedeutung der Maschinenprogrammebene als **Hybrid** skizziert
  - **Maschinenbefehle** der Befehlssatzebene und **Betriebssystembefehle**
    - letztere als **Systemaufrufe** abgesetzt und partiell interpretiert
  - Betriebssysteme als Programme der Befehlssatzebene eingeordnet
- $\text{Ebene}_{[2,3]}$ als **Programmhierarchie** virtueller Maschinen erklärt
  - Repräsentation einer **Systemfunktion** in Hochsprache, Assembliersprache und symbolischen Maschinenkode behandelt
  - in dem Zusammenhang die Implementierung von Systemaufrufen erörtert: **Systemaufrufstumpf** und **Systemaufrufzuteiler**
  - Befehlsabruf- und ausführungszyklus eines Betriebssystems und damit die Funktion als **Interpreter** (von Betriebssystembefehlen) verdeutlicht
- **Organisationsprinzipien** von Maschinenprogrammen präsentiert
  - domänenübergreifende **Aufrufhierarchie** von Funktionen verschiedener Abstraktionsebenen im Zuge der Ausführung eines Systemaufrufs
  - Ebene_3-Programme sind ein Ensemble von (a) Anwendungsroutinen und (b) Laufzeitsystem und Systemaufrufstümpfen
  - Komplex (b) ist Teil einer statischen/dynamischen **Programmbibliothek**

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Fog, A. :
Optimization Manuals.
4. Instruction Tables.
Technical University of Denmark, Dez. 2014

[2] Intel Corporation (Hrsg.):
Addendum—Intel Architecture Software Developer's Manual.
2: Instruction Set Reference.
Intel Corporation, 1997.
(243689-001)

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.1

Literaturverzeichnis (2)

[4] Organick, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0–262–15012–3

[5] Vasudevan, A. ; Yerraballi, R. ; Chawla, A. :
A High Performance Kernel-Less Operating System Architecture.
In: Estivill-Castro, V. (Hrsg.) ; Australian Computer Society (Veranst.): *Proceedings of the Twenty-Eighth Australasian Computer Science Conference (ACSC2005)* Bd. 38 Australian Computer Society, CRPIT, 2005. –
ISBN 1–920682–20–1, S. 287–296

[6] Wikipedia:
`http://de.wikipedia.org/wiki/Hybrid`.
2015

Anhang

Betriebssystembefehle

Laufzeitkontext

- Kontext eines Programmablaufs
  - der für einen bestimmten Programmablauf relevante Prozessorstatus
  - vorgegeben durch die im Programm festgelegte Berechnungsvorschrift
  - je nach Art und Mächtigkeit der Maschinenbefehle unterschiedlich groß

Prozessorstatus

Der im Programmiermodell der CPU für einen (abstrakten/realen) Prozessor definierte Zustand, manifestiert in den im **Registersatz** dieser CPU gespeicherten Daten.

- **Kontextwechsel** müssen **Konsistenz** des Prozessorstatus wahren
  - hier: Unterprogrammaufrufe, Systemaufrufe, …, Koroutinenaufrufe
  - vorgegeben durch die **Aufrufkonventionen** des jeweiligen Prozessors
    - des Kompilierers einerseits und des Betriebssystems andererseits

flüchtige Register
- Inhalt gilt als unbeständig, darf verändert werden
- bei Aufrufender gespeichert (*caller saved*)[10]

nichtflüchtige Register
- Inhalt gilt als beständig, muss unverändert bleiben
- bei Aufgerufener gespeichert (*callee saved*)

[10] x86: `eax`, `ecx`, `edx`

Programmbeispiel: Speicherzelleninhalte austauschen

- Ebene$_5$

```
1  void swap(long *one, long *other) {
2      long aux = *one;
3      *one = *other;
4      *other = aux;
5  }
```

```
6  extern long foo, bar;
7
8  swap(&foo, &bar);
```

- Ebene$_4$ beziehungsweise Ebene$_{[3,2]}$ im symbolischen Maschinenkode

```
10 swap:
11     pushl   %ebp
12     movl    %esp, %ebp
13     pushl   %esi
14     movl    12(%ebp), %eax
15     movl    8(%ebp), %ecx
16     movl    (%ecx), %edx
17     movl    (%eax), %esi
18     movl    %esi, (%ecx)
19     movl    %edx, (%eax)
20     popl    %esi
21     popl    %ebp
22     retl
```

```
23 pushl   $_bar
24 pushl   $_foo
25 calll   swap
```

23–24 Parameterübergabe
25 Unterprogrammaufruf

11–12 lokale Basis einrichten
13 Register sichern
14–15 Parameterübernahme
16 lokale Variable definieren
17–19 Tausch bewerkstelligen
20–22 Epilog und Rücksprung

Varianten von Aktivierungsblöcken

- funktional gleich auf allen Ebenen, aber nichtfunktional ist Ebene_5 ungleich gegenüber $\text{Ebene}_{[4,3,2]}$ in räum- und zeitlicher Hinsicht

mit lokaler Basis

```
1  swap:
2      pushl   %ebp
3      movl    %esp, %ebp
4      pushl   %esi
5      movl    12(%ebp), %eax
6      movl    8(%ebp), %ecx
7      movl    (%ecx), %edx
8      movl    (%eax), %esi
9      movl    %esi, (%ecx)
10     movl    %edx, (%eax)
11     popl    %esi
12     popl    %ebp
13     retl
```

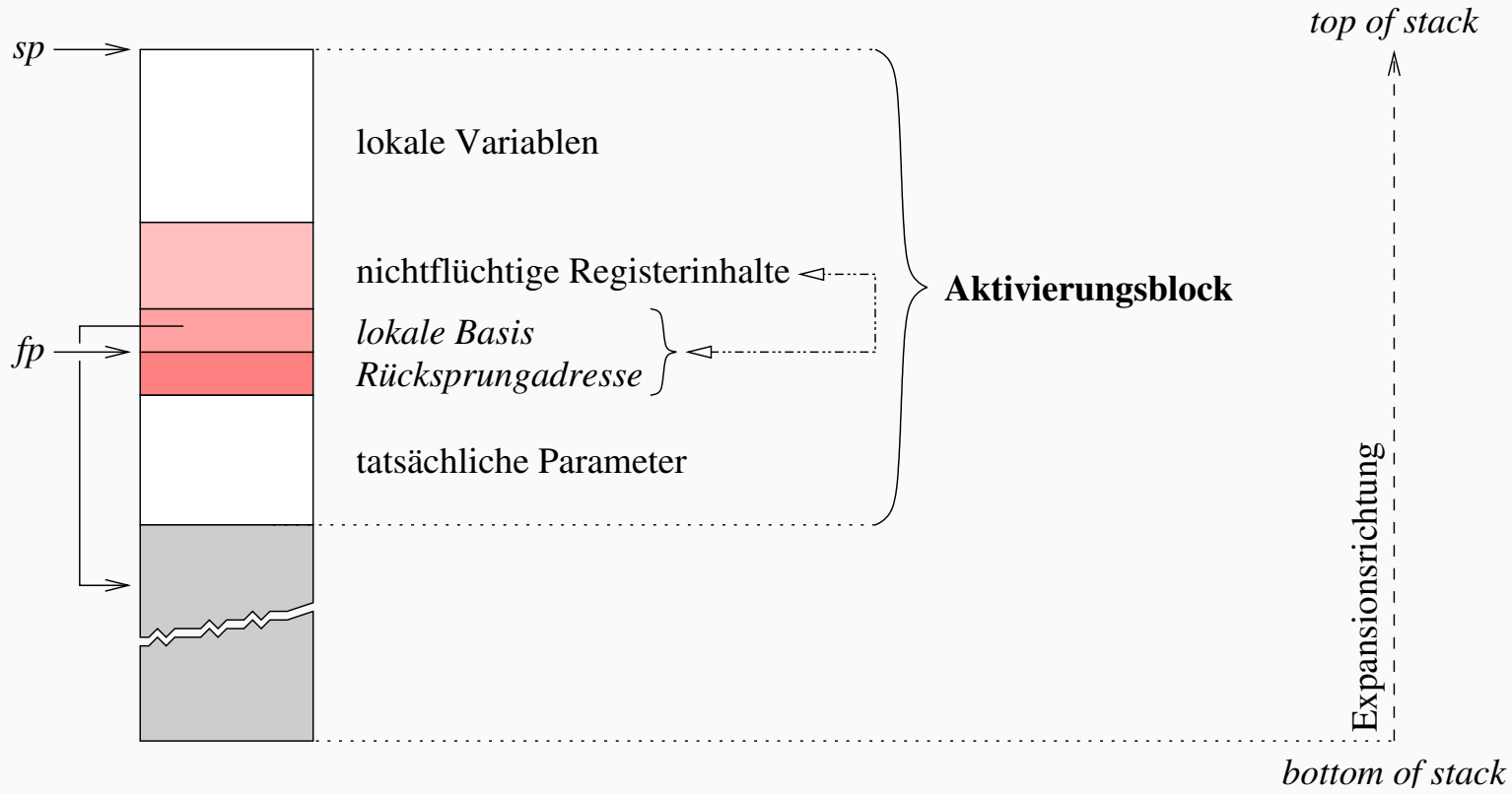
ohne lokaler Basis (`-fomit-frame-pointer`)

```
21 swap:
22 #
23 #
24     pushl   %esi
25     movl    12(%esp), %eax
26     movl    8(%esp), %ecx
27     movl    (%ecx), %edx
28     movl    (%eax), %esi
29     movl    %esi, (%ecx)
30     movl    %edx, (%eax)
31     popl    %esi
32 #
33     retl
```

- Art der Lokalisierung der Argumente, aber auch lokaler Variablen
  - relativ zum Basiszeiger (*base pointer*), ein **fester Bezugspunkt** oder
  - relativ zum Stapelzeiger (*stack pointer*), logisch **variabler Bezugspunkt**

Aktivierungsblock auf dem Stapel

activation record



- Prozessorregister der Befehlssatzebene zur Unterprogrammverwaltung
  - ***sp*** ▪ *stack pointer*, markiert die Oberseite des Stapels
  - ***fp*** ▪ *frame pointer* (optional)[11], die lokale Basis eines Unterprogramms
  - ▪ Zeiger auf die lokale Basis des umgebenden Unterprogramms

[11] `gcc -fomit-frame-pointer` speichert/verwaltet keine lokale Basis (S. 40).

Relevante Merkmale der Befehlssatzebene

- die Expansionsrichtung des Stapels verläuft…
  - von hohen zu niedrigen Adressen (*top-down stack*, x86) oder
  - von niedrigen zu hohen Adressen (*bottom-up stack*)
- der Stapelzeiger adressiert…
  - das zuletzt auf dem Stapel abgelegte Datum (x86) oder
  - den nächsten freien Platz an der Oberseite des Stapels
- eine Adresse auf eine Speicherzelle im Stapel ist…
  - repräsentiert durch eine beliebige Bytenummer (x86) oder
  - ausgerichtet passend zur Operandengröße der nächsten Stapeloperation

Jenseits von Assembliersprache oder nativem Kode

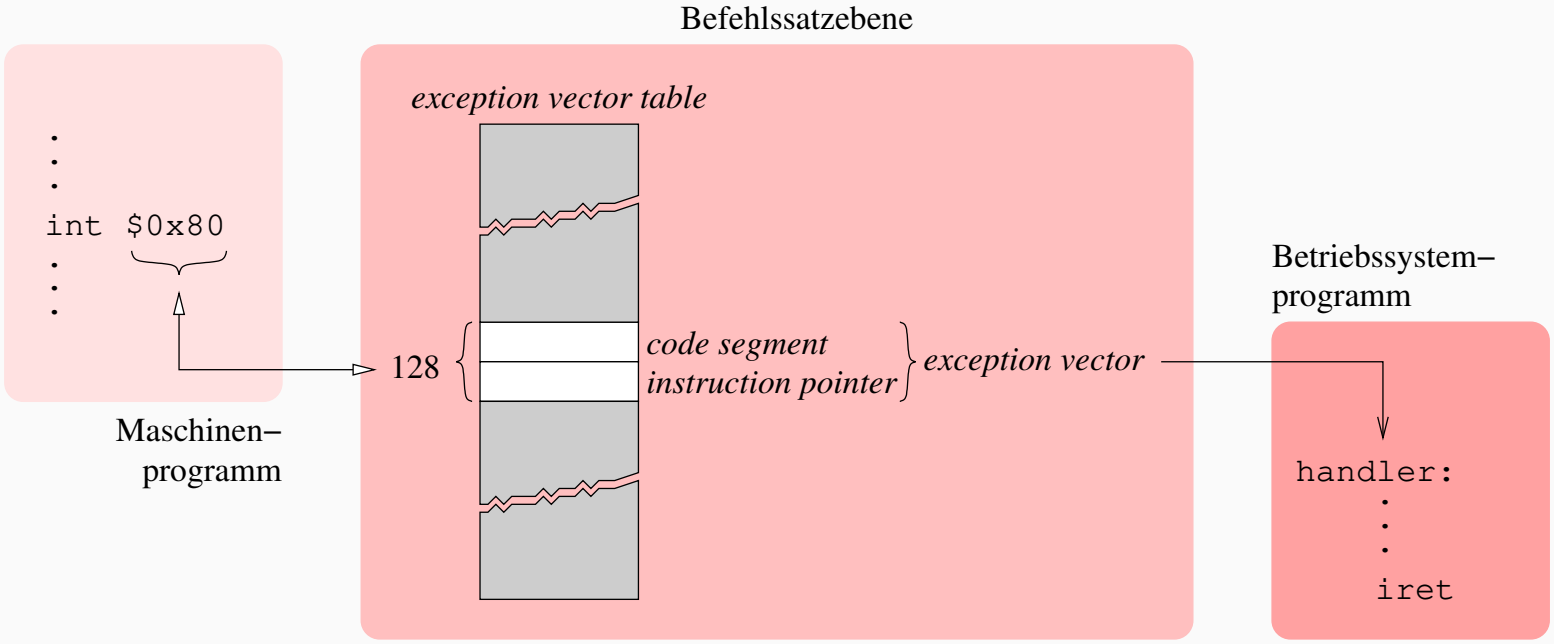
Jedes einzelne dieser Merkmale ist eine prozessorabhängige Größe, die die Software, um den Kontext von Programmabläufen zu speichern, zu verwalten oder zu wechseln nicht übertragbar macht.

↪ Aspekte, die insbesondere für Systemsoftware bedeutsam sind

Anhang

Systemaufrufe

Systemaufruf mittels Unterbrechungsbefehl[12]



- die CPU durchläuft ihren gewöhnlichen **Unterbrechungszyklus**

`int`
- (minimalen) Prozessorstatus sichern
- Befehlszählerregister vom Ausnahmevektor laden
- privilegierten Betriebsmodus aktivieren

`iret`
- (minimalen) Prozessorstatus wieder herstellen
- nichtprivilegierten Betriebsmodus reaktivieren, zurückspringen

[12] `int` (x86), `trap` (m68k, PDP11), `sc` (PowerPC), ..., `svc` (System/370)

Ausnahme ohne wirkliche Ausnahmesituation

- den Systemaufruf konventionell über eine **Abfangstelle** (*trap*) laufen zu lassen, ist vergleichsweise „schwergewichtig“
  - Systemaufruf (`int n`/`iret`) in Relation zu Prozeduraufruf (`call`/`ret`)
  - je nach x86-Modell, Faktor 3–30 mehr an Latenz (Prozessortakte, [1])
- im Zusammenhang mit der Funktionsweise gängiger Betriebssysteme (z.B. Linux) ist dies zudem unzweckmäßig
  - der im Rahmen der Unterbrechungsbehandlung gesicherte Prozessorstatus entspricht nicht der Wirklichkeit des unterbrochenen Prozesses
  - vielmehr geschieht diese Statussicherung, bevor die Prozessorregister zur Argumentenübergabe verwendet werden (vgl. S. 25, Zeile 2)
  - die Statussicherung durch das Betriebssystem bleibt **inkonsistent** (S. 18)
- der eigentlich bedeutsame Aspekt eines Systemaufrufs ist jedoch der **Domänenwechsel**, der „leichtgewichtig“ bewirkt werden kann
  - für x86-Prozessoren wurden hierfür dedizierte Ebene$_2$-Befehle eingeführt
    - `sysenter`/`sysexit` (Intel, [2]) und `syscall`/`sysret` (AMD)
  - diese ändern lediglich den **Betriebsmodus** des Ebene$_2$-Prozessors (CPU)

Systemaufrufbeschleunigung

fast system call interface [2]

`sysenter`/`syscall` unprivilegiert $\mapsto$ privilegiert (d.h., Ebene$_{3\mapsto 2}$)

`sysexit`/`sysret` privilegiert $\mapsto$ unprivilegiert (d.h., Ebene$_{2\mapsto 3}$)

- Verwendung im Maschinenprogramm (Ebene$_3$) für Linux:

Umschaltung hin zur Ebene$_2$

```
1  __kernel_vsyscall:
2    pushl %ecx
3    pushl %edx
4    pushl %ebp
5    movl  %esp,%ebp
6    sysenter
```

- Aufruf ersetzt `int $0x80` im Systemaufrufstumpf
- `sysenter` bewirkt Sprung zu `sysenter_entry` im Kern

Fortsetzung auf Ebene$_3$

```
7  SYSENTER_RETURN:
8    popl %ebp
9    popl %edx
10   popl %ecx
11   ret
```

`Sysexit` erwartet den PC in `%edx` und den SP in `%ecx`, Werte die der Kern definiert:
► `%ecx` ← `%ebp` und
► `%edx` ← &Zeile 7.
Die Registerinhalte müssen daher auf Ebene$_3$ gesichert und wiederhergestellt werden.

- Ausführung von `sysexit` auf Ebene$_2$ bewirkt Rücksprung an Zeile 7
- der Wert von `SYSENTER_RETURN` ist eine „Betriebssystemkonstante“

- der Mechanismus kann die Systemaufruflatenz des Ebene$_2$-Prozessors signifikant verringern (z.B. von 181 auf 92 Taktzyklen [5])

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – V.3 Rechnerorganisation: Betriebssystemmaschine

---

27. Juni 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

Gliederung

Lehrstoff

- den Prozessor der Maschinenprogrammebene als **hybride Maschine** kennenlernen und sein **Operationsprinzip** begreifen
  - Ablauf der Teilinterpretation von Maschinenprogrammen verinnerlichen
  - den Aspekt der Ablaufinvarianz eines Betriebssystems nachvollziehen
- Gemeinsamkeiten und Unterschiede von synchronen und asynchronen **Unterbrechungen** verstehen
  - Konzepte „*Trap*“ und „*Interrupt*“ differenzieren, voneinander abgrenzen
  - beide als Ausnahme von der normalen Programmausführung sehen
  - den Prozessorstatus bzw. -zustand eines Programmablaufs identifizieren
  - daraus Implikationen für die Unterbrechungsbehandlung ableiten
- ein Betriebssystem als **nichtsequentielles Programm** erkennen und in die „Untiefe“ solcher Programme einführen
  - durch Wettlaufsituationen verursachte Laufgefahren erläutern
  - eine erste Einführung zum zentralen Begriff „kritischer Abschnitt“ geben

virtuelle Maschine $\leftrightarrow$ **Betriebssystem** $\leftrightarrow$ hybride Maschine

Einführung

Hybride Maschine

Elementaroperationen der Maschinenprogrammebene

- Maschinenprogramme umfassen zwei Sorten von Befehlen [2, S. 5]:

  i Anweisungen an das Betriebssystem, das Ebene_3 implementiert
  - explizit als **Systemaufruf** (*system call*) kodiert
  - implizit als **Unterbrechung** (*trap*, *interrupt*) ausgelöst

  ii Anweisungen an die CPU, die $\text{Ebene}_{[2,3]}$ implementiert
  - Ebene_2 direkt, nur dort ist die Ausführung aller Befehle der CPU gültig
  - Ebene_3 indirekt, in enger Kooperation mit dem Betriebssystem

- wirklich ausführende Instanz im Rechensystem ist immer die CPU
  - reine Ebene_3-Befehle { werden „wahrgenommen“, nicht ausgeführt, signalisieren jew. eine Ausnahme (*exception*), die ans Betriebssystem „hochgereicht“ wird um dort behandelt zur werden.

- Betriebssysteme fangen Ebene_3-Befehle ab, behandeln Ausnahmen
  - sie bilden jeweils eine (logisch) eigenständige **Maschine**
  - die die von ihr ausführenden Befehle von der CPU zugestellt bekommt

Einführung

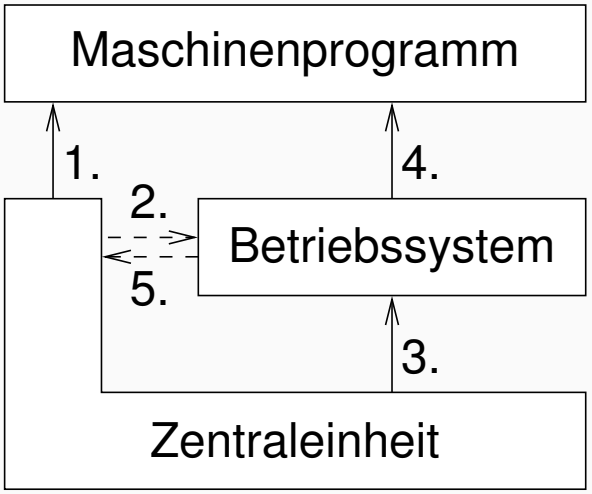
Teilinterpretation

Partielle Interpretation

- die CPU (Zentraleinheit):

1. interpretiert das Maschinenprogramm eingeschränkt befehlsweise,
2. setzt dessen Ausführung aus,
   - Ausnahmesituation
   - **Unterbrechung**

   startet das Betriebssystem und
3. interpretiert die Programme des Betriebssystems befehlsweise.



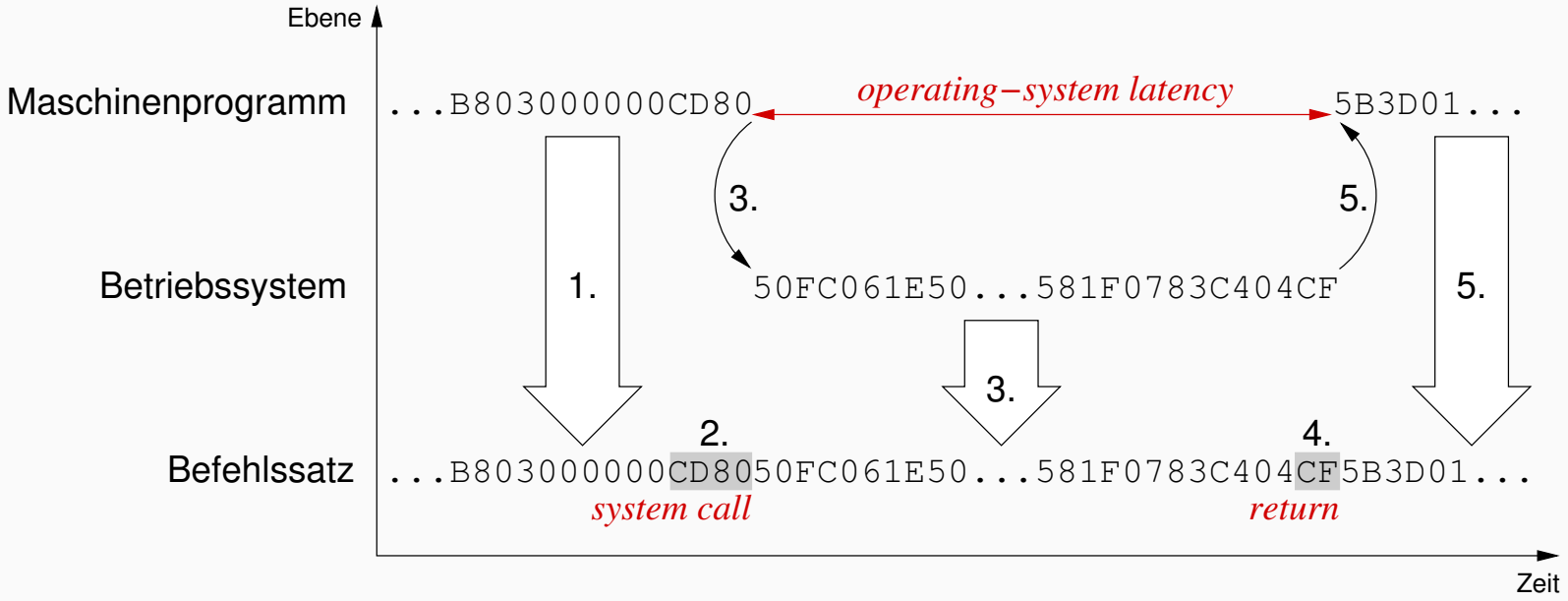
- Folge von 3., der Ausführung von Betriebssystemprogrammen:

4. das Betriebssystem interpretiert das in seiner Ausführung unterbrochene Maschinenprogramm befehlsweise und
5. instruiert die CPU (Zentraleinheit), die Ausführung des/eines zuvor unterbrochenen Maschinenprogramms wieder aufzunehmen.

In Phase 1. ist nur eine Teilmenge von Ebene$_2$-Befehlen direkt von der CPU ausführbar, in Phase 3. dagegen alle.

Ausführung von Maschinenprogrammen

- logischer Aufbau des Befehlsstroms für die Zentraleinheit, in Analogie zu den umseitig (S. 8) genannten fünf Phasen:



1. Ausführung eines Maschinenprogramms durch die Zentraleinheit (CPU)
2. Wahrnehmung einer synchronen/asynchronen Ausnahme (hier: synchron)
3. Teilinterpretation durch das Betriebssystem, Ausnahmebehandlung
4. Beendigung der Ausnahmebehandlung/Teilinterpretation
5. Wiederaufnahme der Ausführung des Maschinenprogramms

Unterbrechung von Betriebssystemabläufen

Die in Phase 3. (S. 8) erfolgende Ausführung von Programmen des Betriebssystems kann ebenfalls ausgesetzt werden. Jede Programmausführung kann eine Unterbrechung erfahren, wenn der ausführende Prozessor dazu befähigt ist.

- **ablaufinvariant** (*re-entrant*) ausgelegte Betriebssysteme ermöglichen den **Wiedereintritt** während der eigenen Ausführung
  - auch wenn Betriebssysteme normalerweise keine Systemaufrufe absetzen, können sie sehr wohl von Unterbrechungen betroffen sein:
    - i im Kontext der Ausführung eines Systemaufrufs ☺
    - ii hervorgerufen durch Peripheriegeräte (Ein-/Ausgabe, Zeitgeber) ☺
    - iii bedingt durch einen Programm(ier)fehler $\rightsquigarrow$ **Panik** ☹
  - die in der Folge notwendige Unterbrechungsbehandlung gestaltet sich wie eine **indirekte Rekursion**
    - das Betriebssystem wird in seiner Definition selbst nochmals aufgerufen
    - nämlich indirekt durch die CPU im Rahmen der partiellen Interpretation
- der Wiedereintritt kann **asynchron** erfolgen, was das Betriebssystem als **nichtsequentielles Programm** darstellt

Zwischenzusammenfassung

- Befehle der Maschinenprogrammbene, also Ebene_3-Befehle sind…
  - „normale“ Befehle der Ebene_2, die die CPU direkt ausführen kann
    - **unprivilegierte Befehle**, die in jedem Arbeitsmodus ausführbar sind
  - „unnormale“ Befehle der Ebene_2, die das Betriebssystem ausführt
    - **privilegierte Befehle**, die nur im privilegierten Arbeitsmodus ausführbar sind
- die „aus der Reihe fallenden“ Befehle stellen Adressräume, Prozesse, Speicher, Dateien und Wege zur Ein-/Ausgabe bereit
  - Interpreter dieser Befehle ist das Betriebssystem
  - der dadurch definierte Prozessor ist die **Betriebssystemmaschine**
- demzufolge ist ein Betriebssystem immer nur **ausnahmsweise** aktiv
  - es muss von außerhalb aktiviert werden
    - programmiert im Falle eines Systemaufrufs (`CD80`: Linux/x86) oder einer sonstigen synchronen Programmunterbrechung (*trap*)
    - nicht programmiert, also nicht vorhergesehen, im Falle einer asynchronen Programmunterbrechung (*interrupt*)
  - es deaktiviert sich immer selbst, in beiden Fällen programmiert (`CF`: x86)

Gliederung

Unterbrechungsarten und Ausnahmesituationen

- die Ausnahmesituationen der Ebene$_2$ fallen in zwei Kategorien:

  trap — **Abfangung** für Ausnahmen von interner Ursache

  interrupt — **Unterbrechung** durch Ausnahmen von externer Ursache

- **Unterschiede** ergeben sich hinsichtlich…
  - Quelle, Synchronität, Vorhersagbarkeit und Reproduzierbarbeit

- ihre **Behandlung ist zwingend** und grundsätzlich prozessorabhängig
  - aufwerfen (*raising*) einer Ausnahme kommt entweder einem realen (CPU) oder einem abstrakten (Betriebssystem) Prozessor zu
    - die CPU wirft eine Ausnahme der Hardware (IRQ, NMI, Fehler)
    - das Betriebssystem wirft eine Ausnahme der Software (POSIX: SIG∗)
  - wogegen die Behandlung (*handling*) einem abstrakten Prozessor obliegt
    - Hardwareausnahmen behandelt das Betriebssystem (auf Ebene$_2$)
    - Betriebssystemausnahmen behandelt das Maschinenprogramm (auf Ebene$_3$)

Ausnahmen

Trap

Synchrone Ausnahme

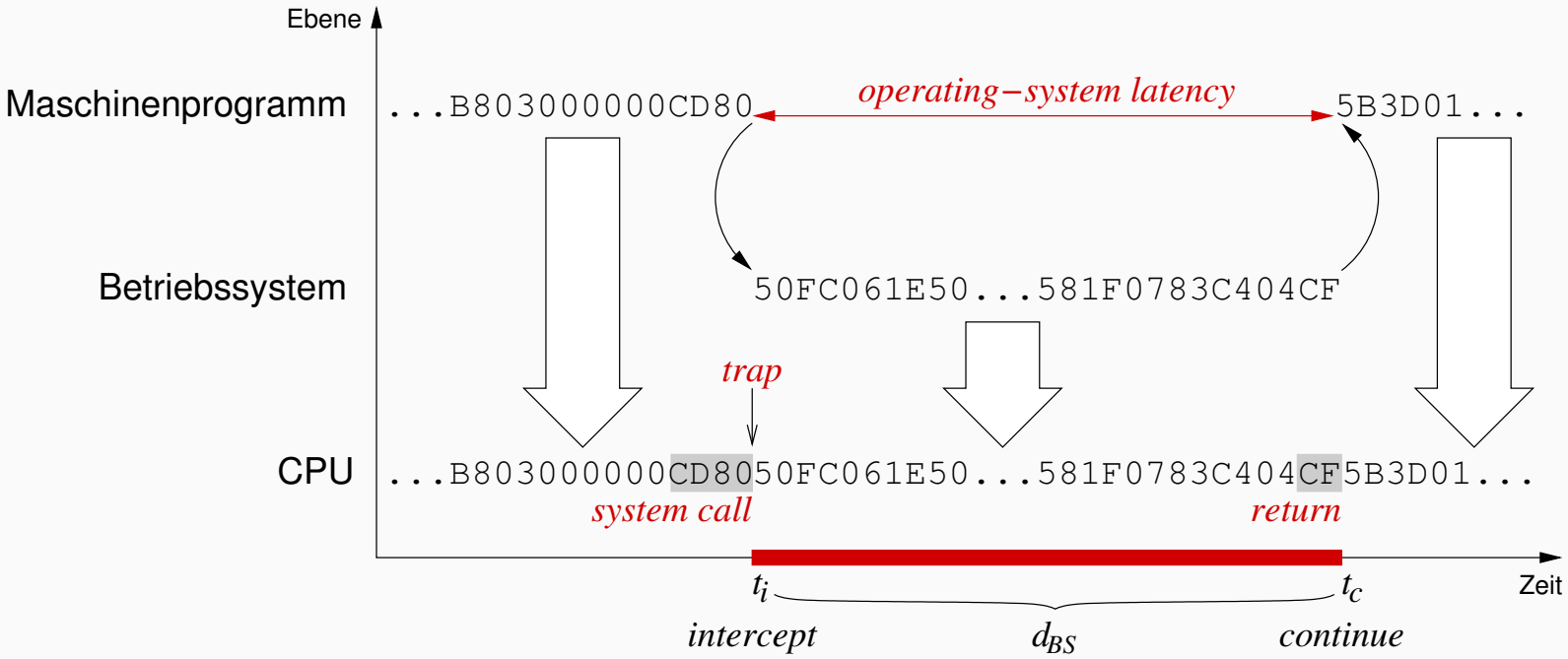
- unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- Seitenfehler im Falle lokaler Ersetzungsstrategien (vgl. [3, S. 16])

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.

  - durch das Programm in Ausführung (≡ Prozess) selbst ausgelöst
  - als Folge der Interpretation eines Befehls des ausführenden Prozessors
- im **Fehlerfall** ist die Behebung der Ausnahmebedingung zwingend

Synchrone Ausnahme — *Trap*



Betriebssystemlatenz

Verzögerung zwischen dem Abfang-/Unterbrechungszeitpunkt und dem Zeitpunkt der Wiederaufnahme der Programmausführung.

d_{BS}

- muss begrenzt sein für ein echtzeitfähiges Betriebssystem
- **maximale Ausführungszeit** (*worst-case execution time*, WCET)

Ausnahmen

Interrupt

Asynchrone Ausnahme

- Signalisierung „externer“ Ereignisse
- Beendigung einer DMA- bzw. E/A-Operation
- Seitenfehler im Falle globaler Ersetzungsstrategien (vgl. [3, S. 16])

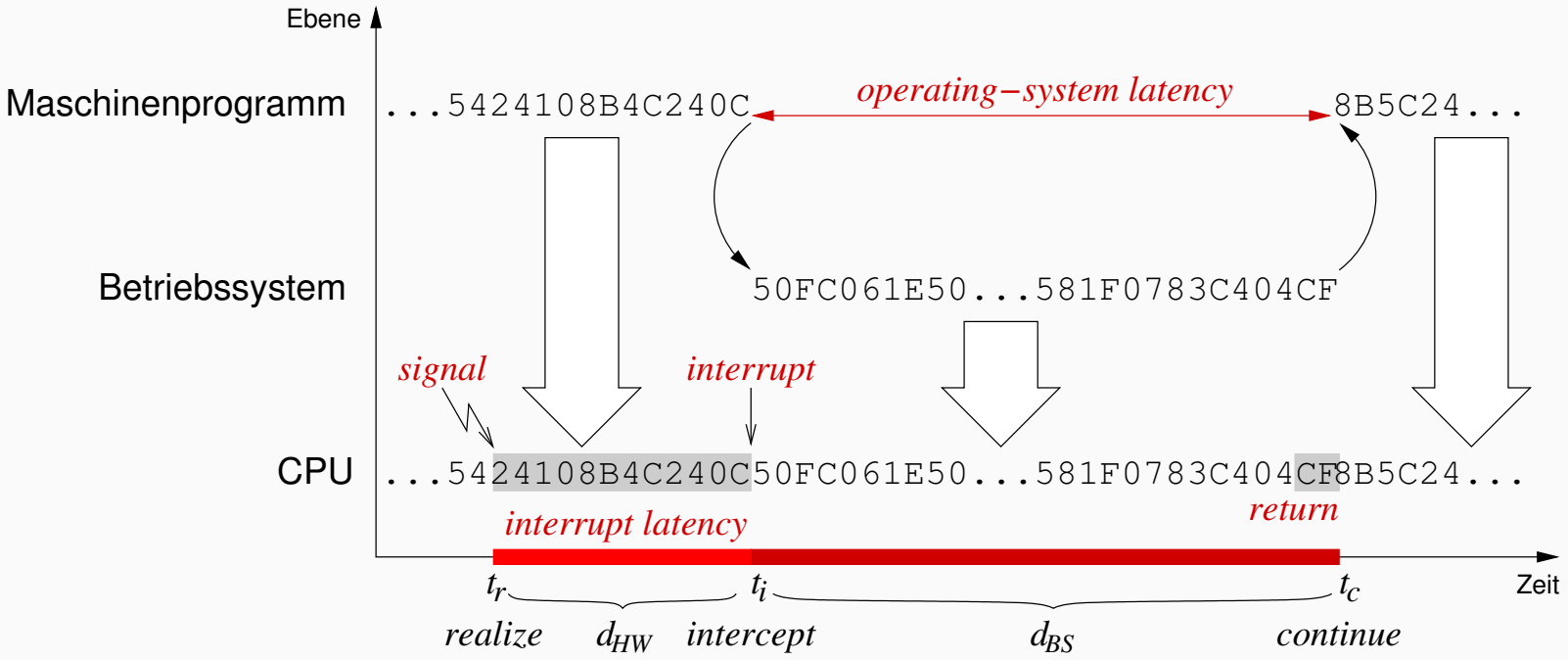
Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

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 die Ausführung des betreffenden Programms unterbrochen wird, ist nicht vorhersehbar.

- durch einen anderen, externen (Soft-/Hardware-) Prozess ausgelöst
- unabhängig von der Befehlsinterpretation des ausführenden Prozessors

- **Nebeneffektfreiheit** der Unterbrechungsbehandlung ist zwingend

Asynchrone Ausnahme — *Interrupt*



Unterbrechungslatenz

Verzögerung zwischen Wahrnehmung (durch die CPU) und Annahme (im Betriebssystem) der Unterbrechung.

d_{HW}

- Restausführungszeit des laufenden Befehls der CPU plus
- restliche Dauer einer **Unterbrechungssperre** im Betriebssystem

Gliederung

Programmunterbrechung

Ausnahmen

Ausnahmen von der normalen Programmausführung

Ausführungsunterbrechungen sind Ereignisse, die den unterbrochenen Programmablauf unerwünscht verzögern und nicht immer durch ihn selbst auch verursacht sind.

- 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)

*Im Betriebssystem sind Maßnahmen zur **Ereignisbehandlung** unabdingbar, im Maschinenprogramm dagegen nicht.*

- sie sind in beiden Fällen jedoch immer problemspezifisch auszulegen

Abrupter Wechsel des Laufzeitkontextes

vgl. [2, S. 30]

- Unterbrechungen implizieren **nicht-lokale Sprünge**:

vom { unterbrochenen / behandelnden } Prozess zum { behandelnden / unterbrochenen } Prozess

- der **Unterbrechungshandhaber**[1] wird plötzlich aktiviert, sein exakter Aktivierungszeitpunkt ist nicht vorhersehbar
  - der **Prozessorstatus** des unterbrochenen Programms ist daher während der Unterbrechungsbehandlung **invariant** zu halten
    - Sicherung vor Ansprung bzw. Start der Behandlungsroutine
    - Wiederherstellung vor Rücksprung zum unterbrochenen Programm
  - Mechanismen dazu liefert die Befehlssatzebene (CPU) bzw. das jeweils behandelnde Programm (Betriebssystem) selbst
- führt zu **Betriebslast**, deren Höhe die Programmierebene des Betriebssystems und die Befehlssatzebene bestimmt

[1] Dem deutschen Patentwesen entnommen, das die englische Bezeichnung „*handler*“ fachbegrifflich als „Handhaber“ übersetzt. Dort wird „*trap*“ sachlich und fachlich korrekt auch als „Falle“ verstanden (vgl. auch S. 16).

Programmunterbrechung

Sicherung/Wiederherstellung

Prozessorstatus invariant halten

- die CPU führt eine **totale oder partielle Zustandssicherung** durch
  - **minimal**
    - Statusregister (SR) und Befehlszähler (*program counter*, PC)
  - **maximal**
    - den kompletten Registersatz
  - Maßnahme, die im **Unterbrechungszyklus** der CPU stattfindet
    - je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt
- das Betriebssystem führt eine **partielle Zustandssicherung** durch

alle { dann noch ungesicherten / flüchtigen[a] / im weiteren Verlauf verwendeten } CPU-Register

[a] Register, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in der **Aufrufkonvention** des Kompilierers.

- die erste Option betrifft ein Betriebssystem, das mit Sprachkonzepten der Ebene_4 (d.h., in Assembliersprache) programmiert wurde
- demgegenüber betreffen die letzten beiden Optionen ein in Hochsprache (Ebene_5) programmiertes Betriebssystem

Option 1: Alle dann noch ungesicherten…

- **Mantelprozedur** zur Statussicherung auf Ebene$_2$:
  - Behandlungsroutine (`handler`) evtl. in Assembliersprache programmiert

```
1  train:
2    pushal
3    call handler
4    popal
5    iret
```

m68k

```
1  train:
2    moveml d0-d7/a0-a6,a7@-
3    jsr handler
4    moveml a7@+,d0-d7/a0-a6
5    rte
```

- `train` (trap/interrupt):
  - **2** alle Arbeitsregisterinhalte im RAM (Stapelspeicher) sichern
  - **3** Unterbrechungsbehandlung durchführen
  - **4** im RAM gesicherten Arbeitsregisterinhalte wiederherstellen
  - **5** unterbrochene Programmausführung wieder aufnehmen
- beteiligte Prozessoren:
  - CPU (Ebene$_2$), Betriebssystem (Ebene$_3$)

Option 2: Alle flüchtigen… vgl. [2, S. 30]

- **Mantelprozedur** zur Statussicherung in Bezug auf Ebene$_5$:
  - Behandlungsroutine (`handler`) in Hochsprache programmiert

```
1  train:
2    pushl %edx; pushl %ecx; pushl %eax
3    call  handler
4    popl  %eax; popl  %ecx; popl  %edx
5    iret
```

m68k

```
1  train:
2    moveml d0-d1/a0-a1,a7@-
3    jsr handler
4    moveml a7@+,d0-d1/a0-a1
5    rte
```

- `train` (trap/interrupt):
  - **2** ▪ Inhalte flüchtiger Arbeitsregister im RAM (Stapelspeicher) sichern
  - **3** ▪ Unterbrechungsbehandlung durchführen
  - **4** ▪ im RAM gesicherten Arbeitsregisterinhalte wiederherstellen
  - **5** ▪ unterbrochene Programmausführung wieder aufnehmen
- beteiligte Prozessoren:
  - CPU (Ebene$_2$), Betriebssystem (Ebene$_3$), Kompilierer (Ebene$_5$)

Option 3: Alle im weiteren Verlauf verwendeten…

- **Mantelprozedur** zur Statussicherung auf Ebene$_5$:
  - Behandlungsroutine (`handler`) in Hochsprache programmiert

```
1  inline void __attribute__ ((interrupt)) train () {
2      handler();
3  }
```

- `__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
- beteiligte Prozessoren:
  - CPU (Ebene$_2$), Kompilierer (Ebene$_5$)

Gliederung

Nichtsequentialität

Wettlaufsituation

Nichtsequentielles Programm

vgl. [8, S. 9]

```
1  int wheel = 0;
```

Unterbrechungen *bewirken, dass nicht zu jedem Zeitpunkt bestimmt ist, wie es weiter geht. (vgl. auch S. 45)*

- welche `wheel`-Werte gibt `main()` aus?

```
2  main () {
3      for (;;)
4          printf("%u\n", wheel++);
5  }
```

  - normalerweise fortlaufende Werte im Bereich[2] $[0, 2^{32} - 1]$, Schrittweite 1

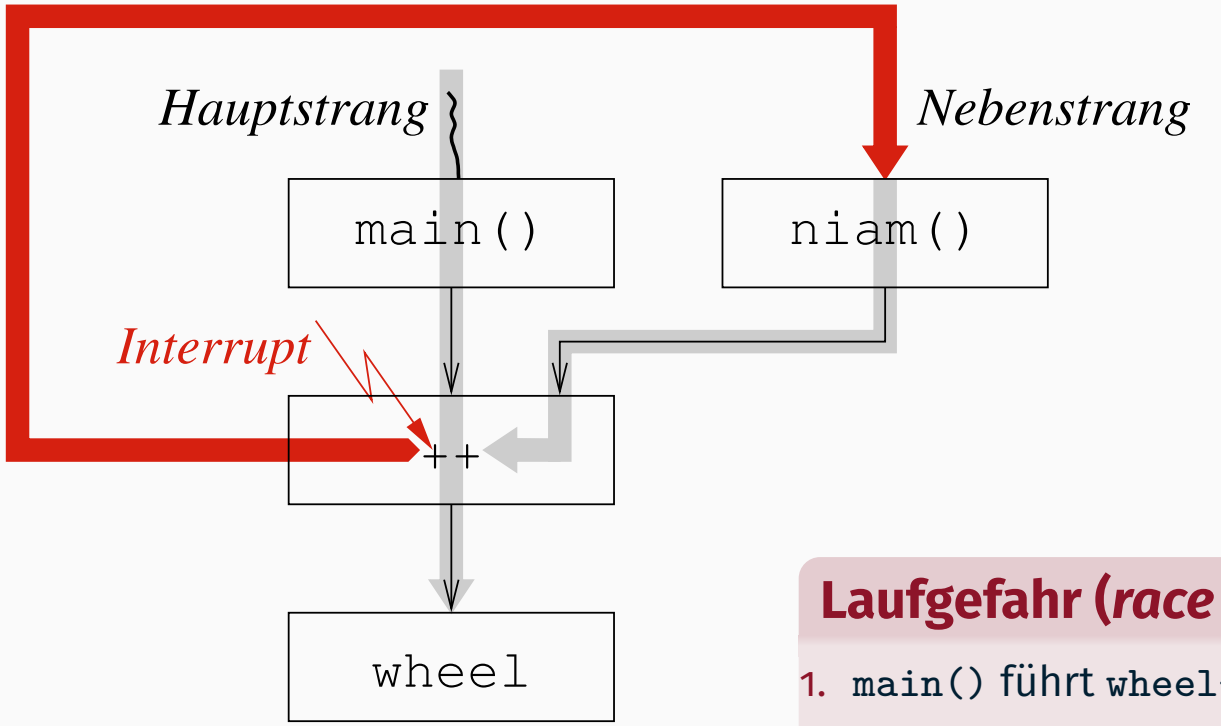
- falls `niam()` `main()` unterbricht : welche Ausgabewerte nun?

```
6  void __attribute__ ((interrupt)) niam () {
7      wheel++;
8  }
```

  - mit Schrittweite n, $0 \leq n \leq 2^{32} - 1$, jenachdem…
    - wie `wheel++` vom Kompilierer für die zugrunde liegende CPU übersetzt und
    - wie oft und wo `main()` dann von `niam()` unterbrochen wurde
  - $n = 1$ impliziert nicht, dass keine Unterbrechung stattgefunden hat

[2] Annahme: `sizeof(unsigned int)` $= 4$ Bytes je acht Bits, d.h. 32 Bits.

Asynchronität von Unterbrechungen



Laufgefahr (*race hazard*)

1. `main()` führt `wheel++` aus
2. `wheel++` wird unterbrochen
3. der *Interrupt* führt zu `niam()`
4. `niam()` führt `wheel++` aus
5. `wheel++` überlappt sich selbst

Teilbarkeit von Aktionen[3] (vgl. auch S. 47)

- `wheel++` ist eine **Elementaroperation** (kurz: Elop) der Ebene$_5$
  - in Hochsprache formuliert ist diese Aktion scheinbar **atomar**, **unteilbar**
- nicht zwingend ist `wheel++` auch eine Elop der Ebene$_4$ (und tiefer)
  - in Assemblier-/Maschinensprache formuliert ist diese Aktion **teilbar**

	`main()`	`niam()`
Ebene$_5$	`wheel++`	

Ebene$_4$	`movl wheel,%edx`
	`leal 1(%edx),%eax`
	`movl %eax,wheel`
		`incl wheel`

# Elop	3	1

  - dies trifft insbesondere auch zu auf die dreiphasige Aktion `incl wheel`:
    - den Wert (1) von `wheel` laden, (2) verändern und (3) an `wheel` speichern
  - ein ***read-modify-write*-Zyklus**, teilbar bei Mehr-/Viel(kern)prozessoren
- im **Überlappungsfall** ist die gleichzeitige Ausführung von `wheel++` möglich, was falsche Berechnungsergebnisse liefern kann

[3] Aktion ist die Ausführung einer Anweisung einer (virtuellen/realen) Maschine.

Nichtsequentialität

Kritischer Abschnitt

Unterbrechungsbedingte Überlappungseffekte

- `niam()`-Ausführung überlappt `main()`-Ausführung:

`wheel`	Befehls-folge	`main()`			`niam()`
		x86-Befehl	%edx	%eax	*x86-Befehl*
42	1	`movl wheel,%edx`	42	?	
43	2				`incl wheel`
43	3	`leal 1(%edx),%eax`	42	43	
43	4	`movl %eax,wheel`	42	43	

  - zweimal `wheel++` durchlaufen (nämlich je einmal in `main()` und `niam()`)
  - zweimal gezählt, den Wert von `wheel` aber nur um eins erhöht

- in nichtsequentiellen Programmen ist die Implementierung des Inkrementoperators[4] (++) als **kritischer Abschnitt** aufzufassen

> *critical in the sense, that the processes have to be constructed in such a way, that at any moment at most one of the two is engaged in its **critical section**. [1, S. 11]*

[4] Gleiches gilt für den Dekrementoperator, egal ob Prä- oder Postfix.

Semantikkonforme Elementaroperation

- der **Postfix-Inkrementoperator** (`wheel++`) hat folgende Semantik:
  - ***fetch*** den Operandenwert (`wheel`) als Ausdruckswert bereitstellen
  - ***add*** dann dem Operanden den um eins erhöhten (++) Wert zuweisen
- ein „***fetch and add***“ (FAA), geschieht logisch in einem Schritt
  - um der Laufgefahr vorzubeugen, muss diese Aktion physisch unteilbar sein
- dies leistet `xadd` (x86):
  - i $tmp \leftarrow dst$
  - ii $dst \leftarrow tmp + src$
  - iii $src \leftarrow tmp$
  - $src = 1$, $dst = wheel$
- Befehl durchsetzen: 4–7
  - *inline assembler* (`gcc`)
- die nunmehr unteilbare Aktion für `printf()` sicherstellen:

```
11  printf("%u\n", FAA(&wheel, 1));
```

```
1   inline int FAA(int *ref, int val) {
2       int aux = val;
3
4       asm volatile ("xaddl %0, %1"
5           : "=g" (aux), "=m" (*ref)
6           : "0" (aux), "m" (*ref)
7           : "memory", "cc");
8
9       return aux;
10  }
```

```
12  ...
13  movl  $1, %eax
14  xaddl %eax, wheel
15  ...
```

Unteilbarer Grundblock

Definition (Grundblock (*basic block*))

Ein aus einer Anweisungsfolge bestehender Programmabschnitt mit genau einem Eintrittspunkt und einem Austrittspunkt.

Gelingt die Abbildung einer in Hochsprache ausformulierten kritischen Operation auf einen elementaren Maschinenbefehl nicht — weil sie zu komplex ist oder ein äquivalenter Maschinenbefehl nicht existiert, gefunden werden kann oder gesucht werden will —, muss die Operation als kritischer Abschnitt ausformuliert werden.

- **Unterbrechungssperre**
  - **4** IRQ abwehren
  - **5–6** unteilbar, atomar
  - **7** IRQ zulassen

  - **IRQ** *interrupt request*
- **Holzhammermethode**
  - Kollateraleffekte
  - Alternative [7, S. 5–15]

```
1  inline int FAA(int *ref, int val) {
2      int aux;
3
4      enter(INTERRUPT_LOCK);
5      aux = *ref;
6      *ref += val;
7      leave(INTERRUPT_LOCK);
8
9      return aux;
10 }
```

vgl. auch S. 50*ff*

Gliederung

Resümee

…Unterbrechungen sind nicht einfach

- zur Einführung wurden **virtuelle Maschinen** erneut aufgegriffen
  - um zu verdeutlichen, dass ein Betriebssystem eine **hybride Maschine** ist
  - die die **Teilinterpretation** von Maschinenprogrammen bewerkstelligt
    - was den Wiedereintritt ins Betriebssystem einschließt: **Ablaufinvarianz**
- das Mittel zur Teilinterpretation ist die **Ausnahme**

  trap
  - die synchron, vorhersagbar und reproduzierbar ist

  interrupt
  - sich asynchron, unvorhersagbar und nicht reproduzierbar zeigt

  - wodurch in ihrer Ausführung unterbrochene Programme verzögert werden
- dabei ist der aktuelle **Laufzeitkontext** invariant zu halten
  - was **Sicherung/Wiederherstellung** des Prozessorstatus zur Folge hat
    - geleistet durch die Befehlssatzebene (CPU) und dem Betriebssystem
- überlappende Programmausführung bringt **Nichtsequentialität**
  - die eine **Wettlaufsituation** bei der Programmausführung bewirken kann
  - der diesbezügliche Programmbereich ist ein **kritischer Abschnitt**
    - in dem sich zu jedem Zeitpunkt nur ein einziger Prozess befinden darf

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[2] Kleinöder, J. ; Schröder-Preikschat, W. :
Maschinenprogramme.
In: [4], Kapitel 5.2

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Speichervirtualisierungung.
In: [4], Kapitel 12.3

Literaturverzeichnis (2)

[4] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

[5] Kopetz, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997. –
ISBN 0–7923–9894–7

[6] Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Concurrent Systems — Nebenläufige Systeme.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)

Literaturverzeichnis (3)

[7] Schröder-Preikschat, W. :
Guarded Sections.
In: [6], Kapitel 10

[8] Schröder-Preikschat, W. :
Processes.
In: [6], Kapitel 3

Anhang

Interrupt

Interferenz

- Unterbrechungen verursachen **Zittern** (*jitter*) im Ablaufverhalten, machen Programme **nicht-deterministisch**
  - nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
- je nach „Räumlichkeit“ des unterbrochenen (P_i) und behandelnden (P_h) Programms ergeben sich verschiedene Ausprägungen

  getrennt
  - P_i auf Ebene$_3$, P_h auf Ebene$_2$
  - in räumlicher Hinsicht hat P_h keinen Einfluss auf P_i

  gemeinsam
  - P_i und P_h zusammen auf Ebene$_3$ oder Ebene$_2$
  - in räumlicher Hinsicht kann P_h einen Einfluss auf P_i haben
  - P_i und P_h bilden ein **nichtsequentielles Programm**
  - aber in beiden Fällen wird P_i um die jeweilige Dauer von P_h verzögert
- in zeitlicher Hinsicht beeinflusst die Unterbrechungsart „*interrupt*“ jedes Programm, dessen Ausführung dadurch ausgesetzt wird
  - dies ist kritisch für **echtzeitabhängige Programme**[5]

[5] Deren korrektes Verhalten hängt nicht nur von den logischen Ergebnissen von Berechnungen ab, sondern auch von dem physikalischen Zeitpunkt der Erzeugung und Verwendung der Berechnungsergebnisse. [5]

Anhang

Wettlaufsituation

Unteilbarkeit

Definition (in Anlehnung an den Duden)

Das Unteilbarsein, um etwas als Einheit oder Ganzheit in Erscheinung treten zu lassen.

- eine Frage der „Distanz“ des Betrachters (Subjekts) auf ein Objekt
  - **Aktion** auf höherer, **Aktionsfolge** auf tieferer Abstraktionsebene

Ebene	Aktion	Aktionsfolge
5	`i++`	

4–3	`incl i`*	`movl i,%r`
		`addl $1,%r`*
		`movl %r,i`

	`addl $1,i`*	
2–1		* ***read*** *from memory into accumulator* ***modify*** *contents of accumulator* ***write*** *from accumulator into memory*

  - typisch für den Komplexbefehl eines „abstrakten Prozessors“ (C, CISC)

Unteilbarkeit komplexer Operationen

Ganzheit oder Einheit einer Aktionsfolge, deren Einzelaktionen alle scheinbar gleichzeitig stattfinden (d.h., synchronisiert sind)

- wesentliche Eigenschaft für eine atomare Operation[6]
  - die logische Zusammengehörigkeit von Aktionen in zeitlicher Hinsicht
  - wodurch die Aktionsfolge als **Elementaroperation** (ELOP) erscheint

Beispiele von Aktionen zum Inkrementieren eines Zählers:

- Ebene $5 \mapsto 3$

C/C++

```
1 i++;
```

ASM

```
1 movl i, %eax
2 addl $1, %eax
3 movl %eax, i
```

- Ebene $3 \mapsto 2$

ASM

```
1 incl i
```

ISA

```
1 read   A from <i>
2 modify A by 1
3 write  A to <i>
```

- die Inkrementierungsaktionen (`i++`, `incl`) sind nur bedingt unteilbar
  - unterbrechungsfreier Betrieb (Ebene $5 \mapsto 3$), Uniprozessor (Ebene $3 \mapsto 2$)
  - Problem: zeitliche Überlappung von Aktionsfolgen hier gezeigter Art

[6] von (gr.) *átomo* „unteilbar“.

Anhang

Kritischer Abschnitt

Ein-/Austrittsprotokolle


```
1  typedef enum safeguard {INTERRUPT_LOCK} safeguard_t;
2
3  extern void panic(char *);
4
5  inline void enter(safeguard_t type, ...) {
6      if (type == INTERRUPT_LOCK)
7          asm volatile ("cli" : : : "cc");
8      /* more safeguard variants... */
9      else
10         panic("bad safeguard");
11 }
12
13 inline void leave(safeguard_t type, ...) {
14     if (type == INTERRUPT_LOCK)
15         asm volatile ("sti" : : : "cc");
16     /* more safeguard variants... */
17     else
18         panic("bad safeguard");
19 }
```

Unteilbarer Grundblock Ebene$_4$

- Übersetzung von `faa.c` (vgl. S. 37) mit `-S`

```
1  FAA:
2    movl 4(%esp), %edx
3    cli
4    movl (%edx), %eax
5    movl 8(%esp), %ecx
6    addl %eax, %ecx
7    movl %ecx, (%edx)
8    sti
9    ret
```

oben `-D"int_t=long int"`

3–8 unteilbare Sequenz

rechts `-D"int_t=long long int"`

8–15 unteilbare Sequenz

`cli` *clear interrupt flag*, abschalten

`sti` *set interrupt flag*, einschalten

```
1  FAA:
2    subl $8, %esp
3    movl %ebx, (%esp)
4    movl 16(%esp), %ecx
5    movl %esi, 4(%esp)
6    movl 20(%esp), %ebx
7    movl 12(%esp), %esi
8    cli
9    movl (%esi), %eax
10   movl 4(%esi), %edx
11   addl %eax, %ecx
12   adcl %edx, %ebx
13   movl %ecx, (%esi)
14   movl %ebx, 4(%esi)
15   sti
16   movl (%esp), %ebx
17   movl 4(%esp), %esi
18   addl $8, %esp
19   ret
```

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VI.1 Betriebssystemkonzepte: Prozesse

23. Mai 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

Gliederung

Lehrstoff

- **Prozess** als das zentrale Konzept von Betriebssys. kennenlernen
  - wobei von der **Verkörperung** (Inkarnation) dieses Konzepts getrennt wird
    - ob also ein Prozess z.B. als *Thread* oder *Task* implementiert ist bzw.
    - ob er allein oder mit anderen zusammen im selben Adressraum verweilt und
    - ob sein Adressraum eine physisch durchzusetzende Schutzdomäne darstellt
  - auf das Wesentliche konzentrieren: Prozess als „*program in execution*“ [7]
- auf (die Art der) **Betriebsmittel** eingehen, die ein Prozess benötigt
  - wiederverwendbare und konsumierbare Betriebsmittel unterscheiden
  - implizite und explizite Koordinierung von Prozessen verdeutlichen, d.h.,
  - geplante und programmierte **Synchronisation** von Prozessen erklären
  - mehr- und einseitige Synchronisation beispielhaft zeigen: *bounded buffer*
- **Prozessausprägungen** und zugehörige Systemfunktionen beleuchten
  - typische (logische) Verarbeitungszustände von Prozessen einführen
  - Einplanung (*scheduling*) und Einlastung (*dispatching*) differenzieren
  - Verortung von Prozessen auf Benutzer- und Systemebene skizzieren
  - Prozesskontrollblock, -zeiger und -identifikation begrifflich erfassen

Einführung

Begriff

Rezipiert als Informatikbegriff

Definition (Prozess $\equiv$ Programmablauf)

Ein Programm in Ausführung durch einen Prozessor.

- das Programm spezifiziert eine Folge von Aktionen des Prozessors
  - die Art einer Aktion hängt von der betrachteten Abstraktionsebene ab

    Ebene $_5$ $\mapsto$ Programmanweisung $\geq$ 1 Assembliermnemoniks

    Ebene $_4$ $\mapsto$ Assembliermnemonik $\geq$ 1 Maschinenbefehle

    Ebene $_3$ $\mapsto$ Maschinenbefehl $\geq$ 1 Mikroprogramminstruktionen

    Ebene $_2$ $\mapsto$ Mikroprogramminstruktion
  - die Aktion eines Prozessors ist damit nicht zwingend unteilbar (atomar)
    - sowohl für den abstrakten (virtuellen) als auch den realen Prozessor
- das Programm ist statisch (passiv), ein Proze. ist dynamisch (aktiv)

Hinweis (Prozess $\neq$ Prozessinkarnation, Prozessexemplar)

*Eine Prozessinkarnation ist **Exemplar** eines Programms als **Bautyp** für einen Prozess, wie ein Objekt Exemplar eines Datentyps ist.*

Gliederung

Grundlagen

Virtualität

Partielle Virtualisierung

- Prozess bezeichnet sowohl den Ablauf eines Programms als auch die **Abstraktion** von einem solchen Programmablauf
  - der physisch durch seinen gegenwärtigen **Laufzeitkontext** definiert ist, insbesondere manifestiert im **Programmiermodell** des Prozessors
- diese Abstraktion ermöglicht es, simultan mehrere Programmabläufe im **Multiplexverfahren** auf einem Prozessor stattfinden zu lassen
  - dabei sind die Abläufe Teil eines einzelnen oder mehrerer Programme
    - Mehrfädigkeit (*multithreading*)/Mehrprogrammbetrieb (*multiprogramming*)
  - für den Ablauf lastet das Betriebssystem einen Prozess ein (*dispatching*)
    - Laufzeitkontext umschalten aktiviert dann einen anderen Programmablauf
  - hierzu plant das Betriebssystem Prozesse entsprechend ein (*scheduling*)
- geläufig ist das **Zeitteilverfahren** (*time-sharing*; CTSS [5]), von dem es verschiedene Ausführungen gibt
  - je nachdem, wie viel und wie oft den Prozessen Rechenzeit innerhalb einer bestimmten Zeitspanne zugeordnet werden kann, soll oder muss
  - pro **Zeitschlitz** laufen im Prozess meist mehrere Aktionen (S. 16) ab

Simultanverarbeitung

multiprocessing

- Prozesse sind das Mittel zum Zweck, (pseudo/quasi) **gleichzeitige Programmabläufe** stattfinden zu lassen $\rightsquigarrow$ **Parallelität**
  - ***multiprogramming*** — mehrere Programme
  - ***multitasking*** — mehrere Aufgaben mehrerer Programme
  - ***multithreading*** — mehrere Fäden eines oder mehrerer Progra.
  - pseudo/quasi gleichzeitig, wenn weniger reale Prozessoren zur Verfügung stehen als zu einem Zeitpunkt Programmabläufe möglich sind
- ein Programmablauf ist möglich, wenn:
  - i er dem Betriebssystem explizit gemacht worden ist und
  - ii alle von ihm benötigten **Betriebsmittel** (real/virtuell) verfügbar sind
- ist eine gemeinsame Benutzung (*sharing*) oder logische Abhängigkeit von Betriebsmitteln gegeben, wird **Synchronisation** erforderlich
  - die Fäden/Aufgaben/Programme teilen sich dieselben (realen) Daten
  - formuliert in dem Programm daselbst, das damit als **nichtsequentielles Programm** in Erscheinung tritt

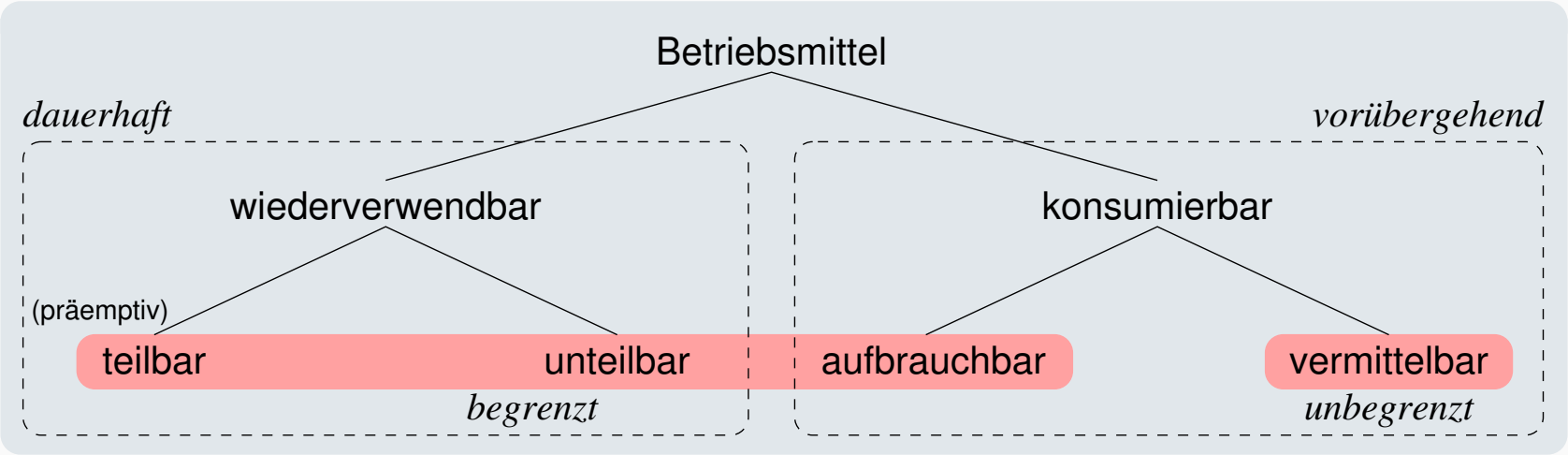
$\hookrightarrow$ die Maßnahmen dazu gestalten sich recht unterschiedlich, je nach Art des Betriebsmittels und Zweck des Prozesszugriffs

Grundlagen

Betriebsmittel

Betriebsmittelklassifikation

nach [15, 16]



- dauerhafte[1] Betriebsmittel sind von Prozessen **wiederverwendbar**
  - sie werden angefordert, belegt, benutzt und danach wieder freigegeben
  - in Benutzung befindliche Betriebsmittel sind ggf. **zeitlich teilbar**
    - je nachdem, ob der **Betriebsmitteltyp** eine gleichzeitige Benutzung zulässt
  - falls unteilbar, sind sie einem Prozess **zeitweise exklusiv** zugeordnet
- vorübergehende Betriebsmittel sind von Prozesse **konsumierbar**
  - sie werden produziert, zugeteilt/vermittelt, benutzt und aufgebraucht
- wiederverwendbare (gegenständliche) und aufbrauchbare (messbare) Betriebsmittel stehen nur begrenzt zur Verfügung

[1] auch: persistente

Eigentümlichkeiten von Betriebsmitteln

- vom Betriebssystem zu verwaltende Betriebsmittel:

wiederverwendbar (Hardware)

- **Prozessor**: CPU, FPU, GPU; MMU
- **Speicher**: RAM, *scratch pad*, *flash*
- **Peripherie**: Ein-/Ausgabe, *storage*

konsumierbar

- **Signal**: IRQ, NMI, *trap*
- **Größe**: Zeit, Energie

- von jedem Programm verwaltete Softwarebetriebsmittel:

wiederverwendbar

- **Text**: kritischer Abschnitt
- **Daten**: Variable, Platzhalter

konsumierbar

- **Signal**: Meldung
- **Nachricht**: Datenstrom

  - wiederverwendbare Betriebsmittel sind Behälter für vermittelbare
    - zur Verarbeitung müssen letztere in Variablen/Platzhaltern vorliegen
  - Verfügbarkeit ersterer beschränkt Erzeugung/Verbrauch letzterer

- gleichzeitige Zugriffe auf unteilbare und Übernahme vermittelbarer Betriebsmittel erfordern die **Synchronisation** involvierter Prozesse

Grundlagen

Programme

Gerichteter Ablauf eines Geschehens [25]

Betriebssysteme bringen Programme zur Ausführung, in dem dazu Prozesse erzeugt, bereitgestellt und begleitet werden

- im Informatikkontext ist ein Prozess ohne Programm nicht möglich
  - die als Programm kodierte Berechnungsvorschrift definiert den Prozess
  - das Programm legt damit den Prozess fest, gibt ihn vor
  - gegebenenfalls bewirkt, steuert, terminiert es gar andere Prozesse
    - wenn das Betriebssystem die dazu nötigen Befehle anbietet!
- ein Programm beschreibt die Art des Ablaufs eines Prozesses

  sequentiell
  - eine Folge von zeitlich nicht überlappenden Aktionen
  - verläuft deterministisch, das Ergebnis ist determiniert

  parallel
  - nicht sequentiell
  - in beiden Arten besteht ein Programmablauf aus **Aktionen**

Beachte: Programmablauf und Abstraktionsebene (vgl. S. 6)

Ein und derselbe Programmablauf kann auf einer Abstraktionsebene sequentiell, auf einer anderen parallel sein. [21]

Definition (Programm)

Die für eine Maschine konkretisierte Form eines Algorithmus.

- virtuelle Maschine C
  - nach der Editierung und
  - vor der Kompilierung

```
1  #include <stdint.h>
2
3  void inc64(int64_t *i) {
4    (*i)++;
5  }
```

- eine Aktion (Zeile 4)

- virtuelle Maschine ASM (x86)
  - nach der Kompilierung[2] und
  - vor der Assemblierung

```
11 inc64:
12   movl 4(%esp), %eax
13   addl $1, (%eax)
14   adcl $0, 4(%eax)
15   ret
```

- drei Aktionen (Zeilen 12–14)

Definition (Aktion)

Die Ausführung einer Anweisung einer (virtuellen/realen) Maschine.

[2] Übersetzung des Unterprogramms (Z. 1–5) mit -S.

Nichtsequentielles Maschinenprogramm

Definition

Ein Programm P, das Aktionen spezifiziert, die parallele Abläufe in P selbst zulassen.

- ein Ausschnitt von P am Beispiel von *POSIX Threads* [17]:

```
1  pthread_t tid;
2
3  if (!pthread_create(&tid, NULL, thread, NULL)) {
4    /* ... */
5    pthread_join(tid, NULL);
6  }
```

- der in P selbst zugelassene parallele Ablauf:

```
7  void *thread(void *null) {
8    /* ... */
9    pthread_exit(NULL);
10 }
```

~~Nicht~~sequentielles Maschinenprogramm

- Aktionen für Parallelität, aber weiterhin **sequentielle Abläufe in P**

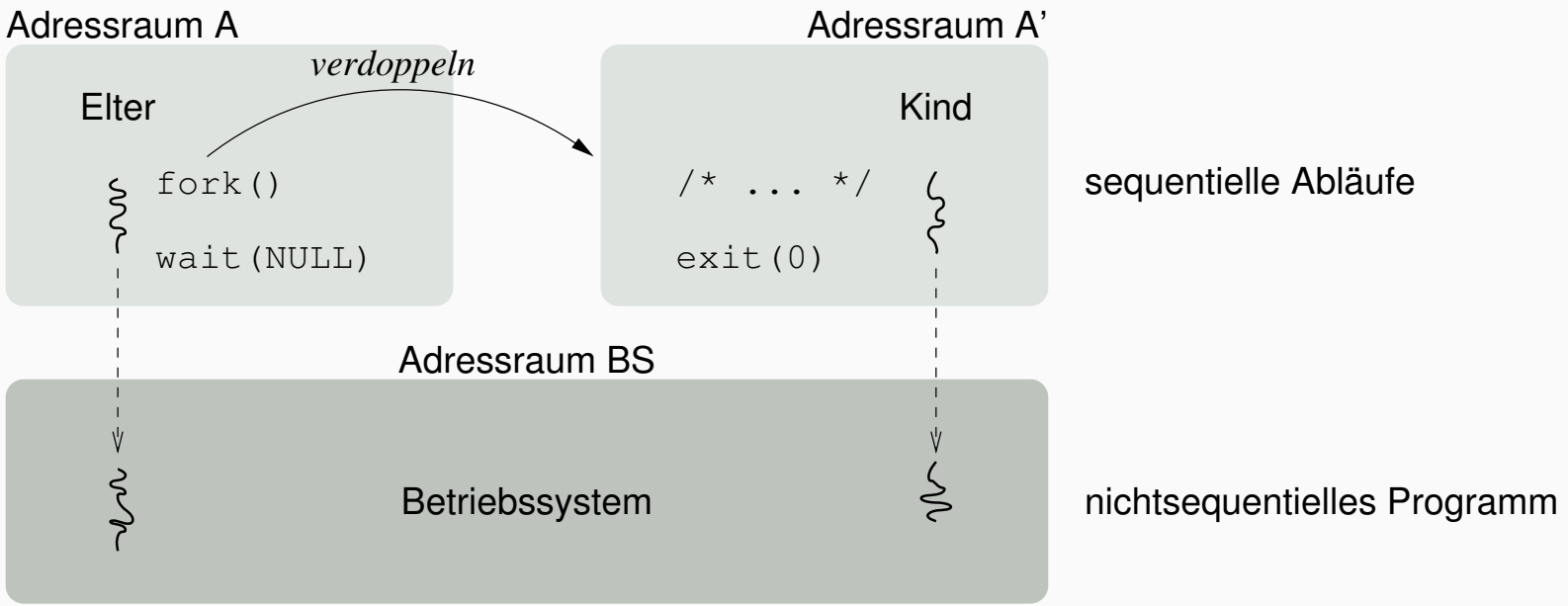
```
1  pid_t pid;
2
3  if (!(pid = fork())) {
4    /* ... */
5    exit(0);
6  }
7
8  wait(NULL);
```

- `fork` dupliziert den Adressraum A von P, erzeugt A' als Duplikat von A
- in A als Ursprungsadressraum entsteht damit jedoch kein paralleler Ablauf
- unabhängig vom Parallelitätsgrad in P, setzt `fork` diesen für A' immer auf 1

  - Programm P spezifiziert zwar Aktionen, die Parallelität zulassen, diese kommt jedoch nur allein durch `fork` nicht in P selbst zur Wirkung
- die Aktionen bedingen parallele Abläufe innerhalb des Betriebssys.
  - Simultanbetrieb (*multiprocessing*) sequentieller Abläufe benötigt das Betriebssystem in Form eines nichtsequentiellen Programms
  - hilfreiches Merkmal: Mehrfädigkeit (*multithreading*) im Betriebssystem
- ein Betriebssystem ist **Inbegriff** des nichtsequen. Programms[3]

[3] Ausnahmen (strikt kooperative Systeme) bestätigen die Regel.

Simultanverarbeitung sequentieller Abläufe



- dabei ist die **Parallelität** in dem System unterschiedlich ausgeprägt:
  - **pseudo**
    - durch *Multiplexen* eines realen/virtuellen Prozessors (S. 9)[4]
  - **echte**
    - durch *Vervielfachung* eines realen Prozessors
- Folge der Operationen sind **parallele Prozesse** im Betriebssystem
  - auch als **nichtsequentielle Prozesse** bezeichnet
  - nämlich Prozesse, deren Aktionen sich zeitlich überlappen können

[4] (gr.) *pseúdein* belügen, täuschen

Gliederung

Verwaltung

Planung

Planung der Einlastung

„scheduling“ von *„dispatching“*

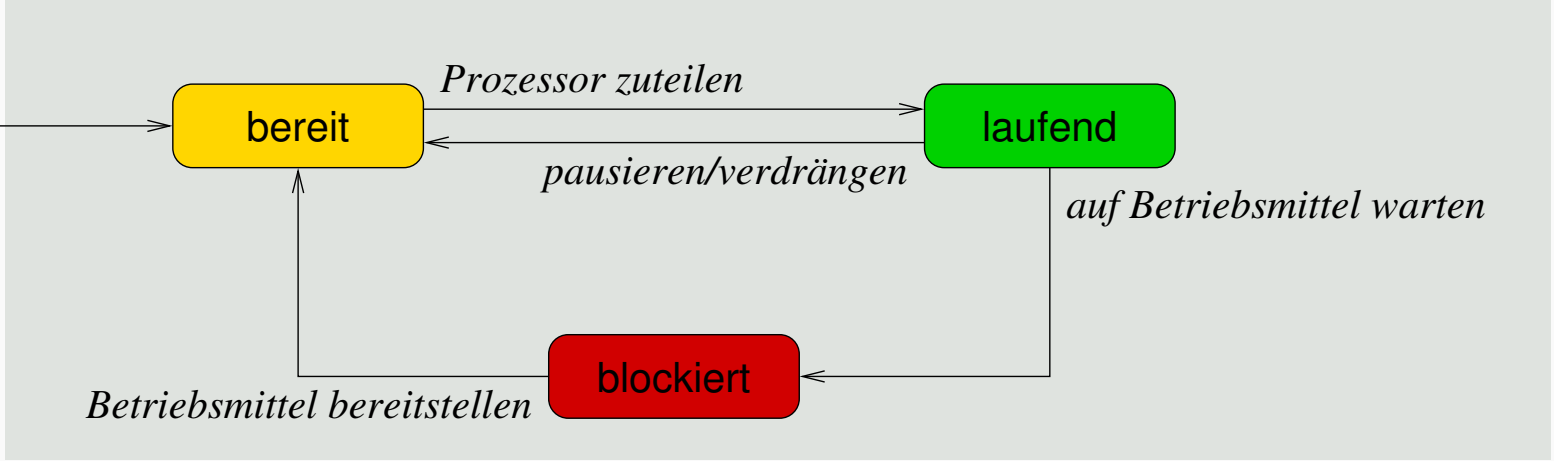
Prozesse werden gestartet, unterbrochen, fortgesetzt und beendet

- zentrale Funktion dabei die **Prozesseinplanung** (*process scheduling*), die allgemein zwei grundsätzliche Fragestellungen zu lösen hat:
  i. Zu welchem (logischen/physikalischen) Zeitpunkt sollen Prozesse in den Kreislauf der Programmverarbeitung eingespeist werden?
  ii. In welcher Reihenfolge sollen die eingespeisten Prozesse stattfinden?
- Zweck aller hierzu erforderlichen Verfahren ist es, die **Zuteilung von Betriebsmitteln** an konkurrierende Prozesse zu kontrollieren

Einplanungsalgorithmus (*scheduling algorithm*)

Beschreibt und formuliert die **Strategie**, nach der ein von einem Rechensystem zu leistender Ablaufplan zur Erfüllung der jeweiligen **Anwendungsanforderungen** entsprechend der gewählten **Rechnerbetriebsart** aufzustellen, abzuarbeiten und fortzuschreiben ist.

- ein Prozess kann angeordnet werden und stattfinden, wenn alle dazu benötigten Betriebsmittel verfügbar sind



- die Zustandsübergänge bewirkt der **Planer** (*scheduler*), sie definieren verschiedene Phasen der Prozessverarbeitung

scheduling
- beim Übergang in die Zustände „bereit“ oder „blockiert“

dispatching
- beim Übergang in den Zustand „laufend“

- je **Rechenkern** kann es zu einem Zeitpunkt stets nur einen laufenden, jedoch mehr als einen blockierten oder bereiten Prozess geben

Reihenfolgebestimmung

Prozessen das Betriebsmittel { **Prozessor**, **Speicher**, **Gerät**, ⋮, **Datei**, **Nachricht**, **Puffer** } koordiniert zuteilen

- Betriebsmittel, die in **Hardware** oder **Software** ausgeprägt vorliegen

- fehlen Prozessen nur noch ein Prozessor als Betriebsmit. definiert die **Bereitliste** (*ready list*) den **Ablaufplan** zur Prozessorzuteilung
  - Listenelemente sind **Prozesskontrollblöcke** (siehe S. 36), geordnet[5] nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, …
  - die Liste repräsentiert sich als statische oder dynamische Datenstruktur

[5] Gemäß Einplanungsstrategie für eine bestimmte Rechnerbetriebsart (Stapel-, Mehrzugangs-, Echtzeitbetrieb).

Verwaltung

Synchronisation

Koordinierung

explizit — die Ausnahme bestätigt die Regel

- Prozesseinplanung profitiert von **Vorwissen** zu Kontrollfluss- und Datenabhängigkeiten, die vorhersehbar sind
  - dann ist ein Ablaufplan möglich, der die Prozesse impliziert koordiniert
- ohne Abhängigkeitsvorwissen sind Prozesse explizit zu koordinieren, per **Programmanweisung** $\rightsquigarrow$ nichtsequentielles Programm
  - der Ablaufplan reiht zwar Prozesse, koordiniert diese jedoch nicht

Definition (Synchronisation [14])

Koordination der Kooperation und Konkurrenz zwischen Prozessen.

- verläuft unterschiedlich, je nach Betriebsmittel- und Prozesszugriffsart

Beachte: Auch vorhergesagte Prozesse finden unvorhersehbar statt, wenn nämlich der Ablaufplan sich als nicht durchsetzbar erweist.

- weil das Vorwissen unvollständig, durch **Ungewissheit** geprägt ist
- weil die **Berechnungskomplexität** den engen Zeitrahmen sprengt
- weil plötzlichem **Hintergrundrauschen** nicht vorgebeugt werden kann
  - Unterbrechungen, Zugriffsfehler auf Zwischen- oder Arbeitsspeicher
  - Befehlsverknüpfung (*pipelining*), Arbitrationslogik (Bus)

Synchronisationsarten

mehr-/einseitige Synchronisation

- bei unteilbaren Betriebsmitteln greift Synchronisation **multilateral**
  - vorausgesetzt die folgenden beiden Bedingungen treffen zu:
    i Betriebsmittelzugriffe durch Prozesse geschehen (quasi) **gleichzeitig** und
    ii bewirken **widerstreitende Zustandsänderungen** des Betriebsmittels
  - Zugriffe auf gemeinsam benutzte Betriebsmittel sind zu koordinieren
    - was sich blockierend oder nichtblockierend auf die Prozesse auswirken kann
    - im blockierenden Fall wird das Betriebsmittel von einem Prozess exklusiv belegt, im nichtblockierenden Fall kann die Zustandsänderung scheitern
- bei konsumierbaren Betriebsmittel wirkt Synchronistaion **unilateral**
  - allgemein auch als logische oder bedingte Synchronisation bezeichnet:
    logisch – wie durch das Rollenspiel der involvierten Prozesse vorgegeben
    bedingt – wie durch eine Fallunterscheidung für eine Berechnung bestimmt
  - Benutzung eines vorübergehenden Betriebsmittels folgt einer Kausalität
    - nichtblockierend für Produzenten und blockierend für Konsumenten
- Prozesse, die gleichzeitig auftreten, überlappen einander zeitweilig
  - sie interagieren zwingend, wenn sie sich dann auch räumlich überlappen
  - dies bedeutet **Interferenz** (*interference*: Störung, Behinderung)…

Koordinationsmittel: Semaphor

- fundamentale Primitiven [9] für Erwerb/Abgabe von Betriebsmitteln, wobei die Operationen folgende **intrinsische Eigenschaften** haben:

  P Abk. für (Hol.) ***prolaag***; alias *down*, *wait* oder *acquire*
  - verringert[6] den Wert des Semaphors s um 1:
    i genau dann, wenn der resultierende Wert nichtnegativ wäre [10, p. 29]
    ii logisch uneingeschränkt [11, p. 345]
  - ist oder war der Wert vor dieser Aktion 0, blockiert der Prozess
    - er kommt auf eine mit dem Semaphor assoziierte Warteliste

  V Abk. für (Hol.) ***verhoog***; alias *up*, *signal* oder *release*
  - erhöht[6] den Wert des Semaphors s um 1
  - ein ggf. am Semaphor blockierter Prozess wird wieder bereitgestellt
    - welcher Prozess von der Warteliste genommen wird, ist nicht spezifiziert
- beide Primitiven sind logisch oder physisch **unteilbare Operationen**, je nachdem, wie dies technisch sichergestellt ist [24]
- ursprünglich definiert als **binärer Semaphor** ($s = [0, 1]$), generalisiert als **allgemeiner Semaphor** ($s = [n, m], m > 0$ und $n \leq m$)

Fallbeispiel

- als Tabelle implementierte **Universalzeigerliste** begrenzter Länge:

```
1  typedef struct table {
2      size_t get, put;
3      void *bay[TABLE_SIZE];
4  } table_t;
5
6  #define PUT(list,item) list.bay[list.put++ % TABLE_SIZE] = item
7  #define GET(list,item) item = list.bay[list.get++ % TABLE_SIZE]
```

- angenommen, mehrere Prozesse agieren mit `GET` oder `PUT` gleichzeitig auf derselben Datenstruktur `list` ⇝ **kritischer Wettlauf**
  - `++` ▪ läuft Gefahr, falsch zu zählen (vgl. [18, S. 28])
  - `PUT` ▪ läuft Gefahr, Listeneinträge zu überschreiben[7]
  - `GET` ▪ läuft Gefahr, denselben Listeneintrag mehrfach zu liefern[7]
- **Simultanverarbeitung** lässt die beliebige zeitliche Überlappung von Prozessen zu, so dass **explizite Koordinierung** erforderlich wird

[7] Mehrere sich zeitlich überlappende Prozesse könnten denselben Wert aus der Indexvariablen (`put` bzw. `get`) lesen, bevor diese verändert wird.

Multilaterale Synchronisation

- **wechselseitiger Ausschluss** (*mutual exclusion*) sich sonst womöglich überlappender Ausführungen von `PUT` & `GET`: **binärer Semaphor**

Vorbelegung des Semaphors

```
/* critical section is free */
buffer_t buffer = {{1}};
```

```
1  typedef struct buffer {
2      semaphore_t lock;
3      table_t data;
4  } buffer_t;
5
6  inline void store(buffer_t *pool, void *item) {
7      P(&pool->lock);          /* enter critical section */
8      PUT(pool->data, item);   /* only one process at a time */
9      V(&pool->lock);          /* leave critical section */
10 }
11
12 inline void *fetch(buffer_t *pool) {
13     void *item;
14     P(&pool->lock);          /* enter critical section */
15     GET(pool->data, item);   /* only one process at a time */
16     V(&pool->lock);          /* leave critical section */
17     return item;
18 }
```

- ein **Unter-/Überlauf** der Universalzeigerliste bzw. des Puffers kann nicht ausgeschlossen werden $\rightsquigarrow$ **Programmierfehler**

Unilaterale Synchronisation

- **Reihenfolgenbildung** von Prozessen, die als Produzent (`stuff`) oder Konsument (`drain`) agieren: **allgemeiner Semaphor**

```
1  typedef struct stream {
2      semaphore_t free, full;
3      buffer_t data;
4  } stream_t;
5
6  void stuff(stream_t *pipe, void *item) {
7      P(&pipe->free);          /* prevent overflow */
8      store(&pipe->data, item);
9      V(&pipe->full);          /* signal consumable */
10 }
11
12 void *drain(stream_t *pipe) {
13     void *item;
14     P(&pipe->full);          /* prevent underflow */
15     item = fetch(&pipe->data);
16     V(&pipe->free);          /* signal space */
17     return item;
18 }
```

Vorbelegung der Semaphore

```
/* all table items available, no consumable
 * critical section is free */
stream_t stream = {{TABLE_SIZE}, {0}, {{1}}};
```

  - typisches Muster der Implementierung eines Klassikers — nicht nur in der Systemprogrammierung: **begrenzter Puffer** (*bounded buffer*)

Verwaltung

Implementierungsaspekte

Prozess ↔ Faden

process ↔ thread

- Gemeinsamkeit besteht darin, einen **Vorgang** auszudrücken, Unterschiede ergeben sich in der technischen Auslegung

Prozess
- Ausführung eines Programms (*locus of control* [8])
- seit Multics eng verknüpft mit „eigenem Adressraum“ [6]
- seit Thoth, als ***Team*** im selben Adressraum teilend [2]

Faden
- konkreter Strang, roter Faden (*thread*) in einem Programm
- **sequentieller Prozess** [1, S. 78] — ein „Thoth-Prozess“

andere
- Aufgabe (*task*), Arbeit (*job*) ∼ Handlung (wie Prozess)
- Faser (*fiber*), Fäserchen (*fibril*) ∼ Gewichtsklasse (wie Faden)

separation of concerns [12, S. 1]

Steht das „Was“ (ein ohne Zweifel bestehender Programmablauf) oder das „Wie“ (Art und Grad der Isolation) im Vordergrund der Diskussion?

- Informatikfolklore vermischt Programmablauf und Adressraumschutz
  - ein Prozess bewegt sich in dem Adressraum, den ein Programm definiert
  - das tut er aber unabhängig davon, ob dieser Adressraum geschützt ist
    - wenn überhaupt, dann ist daher sein Programmspeicher zu schützen…

Verortung

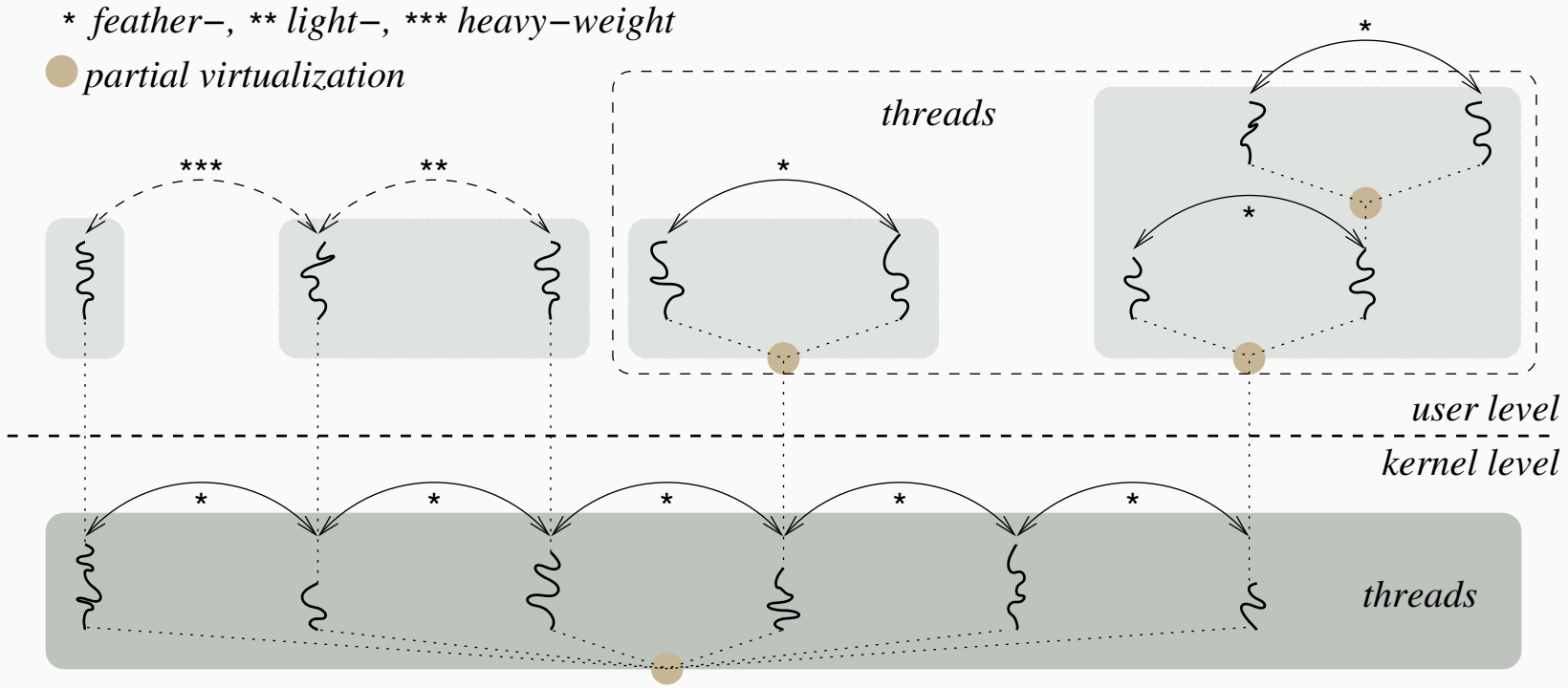
Betriebssystem- vs. Anwendungsdomäne

Prozesse sind in einem Rechensystem verschiedenartig verankert

- unter oder auf der Maschinenprogrammebene
  - **unter**
    - ursprünglich, im Betriebssystem bzw. Kern (*kernel*)
    - Prozessinkarnation als Wurzel
    - partielle Virtualisierung des realen Prozessor(kern)s
    - ↪ „*kernel-level thread*“ in der Informatikfolklore
  - **auf**
    - optional, im Laufzeit- oder gar Anwendungssystem
    - Prozessinkarnation als Blatt oder innerer Knoten
    - partielle Virtualisierung eines abstrakten Prozessor(kern)s
    - ↪ „*user-level thread*“ in der Informatikfolklore
- der jew. Prozessor weiß nicht, dass er ggf. (part.) virtualisiert wird
  - ein „*user-level thread*“ ist ein in Zeit gemultiplexter „*kernel-level thread*“
  - einem „*kernel-level thread*“ sind seine „*user-level threads*“ unbewusst
- Betriebssysteme kennen nur ihre eigenen Prozessinkarnationen
  - ein „*kernel-level thread*“ entsteht durch Raum-/Zeitmultiplexen der CPU
  - hält ein „*kernel-level thread*“ inne, setzen seine „*user-level threads*“ aus

Gewichtsklasse

Unterbrechungs-/Wiederaufnahmeaufwand



- Arten von **Prozesswechsel** zur partiellen Prozessorvirtualisierung:
  - * im selben (Anwendungs-/Kern-) Adressraum, ebenda fortsetzend
  - ** im Kernadressraum, denselben Anwendungsadressraum teilend
  - *** im Kernadressraum, im anderen Anwendungsadressraum landend

Verkörperung

- der **Prozesskontrollblock**[8] (*process control block*, PCB) bündelt alle zur partiellen Virtualisierung relevanten Attribute eines Prozesses
  - in dem (pro Prozess) typischerweise folgende Daten verbucht sind:
    - Adressraum, Speicherbelegung, Laufzeitkontext, …, Ressourcen allgemein
    - Verarbeitungszustand, Blockierungsgrund, Dringlichkeit, Termin
    - Name, Domäne, Zugehörigkeit, Befähigung, Zugriffsrechte, Identifikationen
  - als die zentrale **Informations- und Kontrollstruktur** im Betriebssystem
- pro Prozessor verwaltet das Betriebssystem einen **Prozesszeiger**, der die jeweils laufende Prozessinkarnation identifiziert
  - so, wie der Befehlszähler der CPU den laufenden Befehl adressiert, zeigt der Prozesszeiger des Betriebssystems auf den gegenwärtigen Prozess
  - beim Prozesswechsel (*dispatch*) wird der Prozesszeiger weitergeschaltet
- nach außen wird eine so beschriebene Prozessinkarnation systemweit eindeutig durch eine **Prozessidentifikation** (PID) repräsentiert
  - wobei „systemweit“ recht dehnbar ist und sich je nach Auslegung auf ein Betriebssystem, ein vernetztes System oder verteiltes System bezieht

[8] auch: **Prozessdeskriptor** (*process descriptor*, PD).

Gliederung

Resümee

…Trennung von Belangen trägt zum Verständnis bei

- in der Einführung zunächst prinzipielle **Begrifflichkeiten** erklärt
  - einen **Prozess** als „Programm in Ausführung“ definiert und damit die originale (klassische) Definition [7] übernommen
  - den Unterschied zur **Prozessinkarnation**/-verkörperung hervorgehoben
- darauf aufbauend wichtige **Grundlagen** zum Thema behandelt
  - partielle Virtualisierung und **Simultanverarbeitung**
  - **Betriebsmittel**, deren Klassifikation und Eigentümlichkeiten
  - Programm als Verarbeitungsvorschrift für eine Folge von **Aktionen**
  - **nichtsequentielles Programm**, das Aktionen für Parallelität spezifiziert
- verschiedene Aspekte der **Ausprägung** von Prozessen beleuchtet:

Planung
- implizite Koordinierung, **Einplanung** von Prozessen
- logische Verarbeitungszustände, **Einlastung**

Synchronisation
- explizite **Koordinierung** durch Programmanweisung
- binärer/allgemeiner **Semaphor**, Abgrenzung *Mutex*

Repräsentation
- **Verortung** der Prozesse im Rechensystem
- **Fäden** inner-/oberhalb der Maschinenprogrammebene

↪ Ressource: Prozesskontrollblock, -zeiger, -identifikation

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Bauer, F. L. ; Goos, G. :
Betriebssysteme.
In: *Informatik: Eine einführende Übersicht* Bd. 90.
Springer-Verlag, 1971, Kapitel 6.3, S. 76–92

[2] Cheriton, D. R. ; Malcolm, M. A. ; Melen, L. S.:
Thoth, a Portable Real-Time Operating System.
In: *Communications of the ACM* 22 (1979), Febr., Nr. 2, S. 105–115

[3] Coffman, E. G. ; Denning, P. J.:
***Operating System Theory*.**
Prentice Hall, Inc., 1973

[4] Conway, R. W. ; Maxwell, L. W. ; Millner, L. W.:
***Theory of Scheduling*.**
Addison-Wesley, 1967

Literaturverzeichnis (2)

[5] Corbató, F. J. ; Merwin-Daggett, M. ; Dalex, R. C.:
An Experimental Time-Sharing System.
In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*, ACM, 1962, S. 335–344

[6] Daley, R. C. ; Dennis, J. B.:
Virtual Memory, Processes, and Sharing in MULTICS.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 306–312

[7] Denning, P. J.:
Third Generation Computer Systems.
In: *Computing Surveys* 3 (1971), Dez., Nr. 4, S. 175–216

[8] Dennis, J. B. ; Horn, E. C. V.:
Programming Semantics for Multiprogrammed Computations.
In: *Communications of the ACM* 9 (1966), März, Nr. 3, S. 143–155

Literaturverzeichnis (3)

[9] Dijkstra, E. W.:
Over seinpalen / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –
Manuskript. –
(dt.) Über Signalmasten

[10] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[11] Dijkstra, E. W.:
The Structure of the "THE"-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

Literaturverzeichnis (4)

[12] Dijkstra, E. W.:
On the Role of Scientific Thought.
`http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`,
Aug. 1974

[13] Drescher, G. ; Schröder-Preikschat, W. :
An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections / Friedrich-Alexander-Universität Erlangen-Nürnberg, Department of Computer Science.
Erlangen, Germany, Jan. 2015 (CS-2015-01). –
Technical Reports

Literaturverzeichnis (5)

[14] Herrtwich, R. G. ; Hommel, G. :
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989. –
ISBN 3–540–51701–4

[15] Holt, R. C.:
On Deadlock in Computer Systems.
Ithaca, NY, USA, Cornell University, Diss., 1971

[16] Holt, R. C.:
Some Deadlock Properties of Computer Systems.
In: *ACM Computing Surveys* 4 (1972), Sept., Nr. 3, S. 179–196

Literaturverzeichnis (6)

[17] IEEE:
POSIX.1c Threads Extensions / Institute of Electrical and Electronics Engineers.
New York, NY, USA, 1995 (IEEE Std 1003.1c-1995). – Standarddokument

[18] Kleinöder, J. ; Schröder-Preikschat, W. :
Betriebssystemmaschine.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*. FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.3

[19] Kleinrock, L. :
***Queuing Systems*. Bd. I: Theory.**
John Wiley & Sons, 1975

Literaturverzeichnis (7)

[20] Lister, A. M. ; Eager, R. D.:
Fundamentals of Operating Systems.
The Macmillan Press Ltd., 1993. –
ISBN 0–333–59848–2

[21] Löhr, K.-P. :
Nichtsequentielle Programmierung.
In: Institut für Informatik (Hrsg.): *Algorithmen und Programmierung IV*.
Freie Universität Berlin, 2006 (Vorlesungsfolien)

[22] Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Concurrent Systems — Nebenläufige Systeme.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)

[23] Schröder-Preikschat, W. :
Guarded Sections.
In: [22], Kapitel 10

Literaturverzeichnis (8)

[24] Schröder-Preikschat, W. :
Semaphore.
In: [22], Kapitel 7

[25] Wikipedia:
Prozess.
http://de.wikipedia.org/wiki/Prozess, Nov. 2013

Anhang

Prozessbegriff

Ursprünglich als Rechtsbegriff

Prozess bedeutet „streitiges Verfahren vor Gericht, mit dem Ziel, den Streit durch eine verbindliche Entscheidung zu klären“ [25, Recht]

- Analogie in der Informatik bzw. zu Betriebssystemkonzepten:

Streit	▪ Rivalität[9] bei Inanspruchnahme von Betriebsmitteln ▪ Konkurrenz (lat. *concurrere* zusammenlaufen)
Verfahren	▪ Vorgehensweise zur planmäßigen Problemlösung ▪ Strategie (*policy*) oder Methode der Problemlösung
Gericht	▪ Funktion zur Einplanung (*scheduling*), Koordinierung ▪ Synchronisationspunkt in einem Programm
Verbindlichkeit	▪ Konsequenz, mit der die Einplanungszusagen gelten ▪ Einhaltung zugesagter Eigenschaften, Verlässlichkeit

- in der Regel folgen die Verfahren einer hierarchischen Gerichtsbarkeit
  - Betriebssysteme verfügen oft über eine mehrstufige Prozessverarbeitung
  - was aber kein Verfahrensabschnitt, keine Instanz (*instance*) impliziert
    - Übernahme von „Instanz“ in die Informatik war eher ungeschickt !!!

[9] lat. *rivalis* „an der Nutzung eines Wasserlaufs mitberechtigter Nachbar“

Anhang

Programme

Programme formulieren Prozesse

vgl. S. 16

- Adressbereich und virtuelle Maschine SMC[10]
  - Textsegment
  - Linux
  - nach dem Binden und
  - vor dem Laden
- reale Maschine
  - nach dem Laden
  - ablauffähig

```
1  0x080482f0:   mov  0x4(%esp),%eax     8b 44 24 04
2  0x080482f4:   add  $0x1,(%eax)        83 00 01
3  0x080482f7:   adc  $0x0,0x4(%eax)     83 50 04 00
4  0x080482fb:   ret                     c3
```

- gleiche Anzahl von Aktionen (Zeilen 1–3, jew.), aber verschiedene Darstellungsformen

Hinweis (`ret` bzw. `c3`)

Die Aktion zum Unterprogrammrücksprung korrespondiert zur Aktion des Unterprogrammaufrufs (gdb, `disas /rm main`):

```
1  0x080481c9: c7 04 24 b0 37 0d 08 movl $0x80d37b0,(%esp)
2  0x080481d0: e8 1b 01 00 00       call 0x80482f0 <inc64>
```

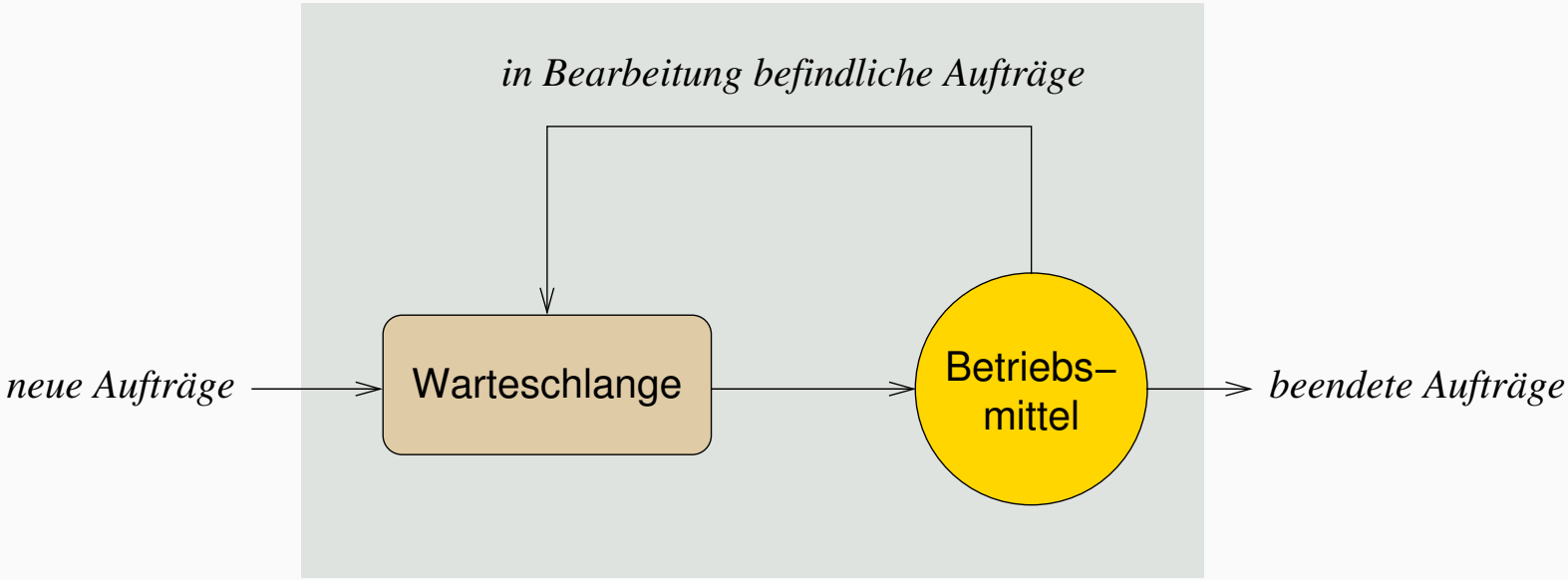
[10] symbolischer Maschinenkode (*symbolic machine code*): x86 + Linux.

Anhang

Planung

Einplanungsalgorithmen

- Verwaltung von (betriebsmittelgebundenen) **Warteschlangen**



Ein einzelner Einplanungsalgorithmus ist charakterisiert durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse in die Warteschlange eingereiht werden. [20]

Warteschlangentheorie und -praxis

- die Charakterisierung von **Einplanungsalgorithmen** macht glauben, Betriebssysteme fokussiert „mathematisch“ studieren zu müssen:
  - R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
  - E. G. Coffman, P. J. Denning. *Operating System Theory*.
  - L. Kleinrock. *Queuing Systems*, *Volume I: Theory*.
- praktische Umsetzung offenbart jedoch einen **Querschnittsbelang** (*cross-cutting concern*), der sich kaum modularisieren lässt
  - spezifische Betriebsmittelmerkmale stehen ggf. Bedürfnissen der Prozesse, die Aufträge zur Betriebsmittelnutzung abgesetzt haben, gegenüber
  - dabei ist die Prozessreihenfolge in Warteschlangen (bereit, blockiert) ein Aspekt, die Auftragsreihenfolge dagegen ein anderer Aspekt
  - **Interferenz** bei der Durchsetzung der Strategien kann die Folge sein
- Einplanungsverfahren stehen und fallen mit den Vorgaben, die für die jeweilige **Zieldomäne** zu treffen sind
  - die „Eier-legende Wollmilchsau“ kann es nicht geben
  - Kompromisslösungen sind geläufig — aber nicht in allen Fällen tragfähig

Anhang

Synchronisation

Wechselseitiger Ausschluss

mutual exclusion

- ein **Synchronisationsverfahren**, das die Formulierung unteilbarer Aktionsfolgen eines nichtsequentiellen Programms unterstützt
  - wobei eine solche Aktionsfolge einem kritischen Abschnitt entspricht

Definition (Kritischer Abschnitt)

Ein Programmabschnitt, der bei nichtsequentieller Ausführung durch **gleichzeitige Prozesse** einen **kritischen Wettlauf** impliziert:

*critical in the sense, that the processes have to be constructed in such a way, that at any moment at most one of [them] is engaged in its **critical section**. [10, S. 11]*

- S. 30 zeigt solche Abschnitte — jedoch ist wechselseitiger Ausschluss zur Vorbeugung eines kritischen Wettlaufs (dort) nicht zwingend

Axiom (s. [23, 13])

Jede „laufgefährliche Aktionsfolge“ lässt sich ohne wechselseitigen Ausschluss absichern $\rightsquigarrow$ **nichtblockierende Synchronisation**.

Mutex

mutual exclusion

- die Semaphorprimitiven P & V sind so definiert (S. 28), paarweise verwendet zu werden — nicht aber zwingend vom selben Prozess
  - sonst wäre einseitige (unilaterale, logische, bedingte) Synchronisation von Prozessen unmöglich $\rightsquigarrow$ **allgemeinerSemaphor**
  - sonst wäre mehrseitige (multilaterale) Synchronisation für einen kritischen Abschnitt, der den Prozess wechselt, falsch[11] $\rightsquigarrow$ **binärer Semaphor**
- in der Informatikfolklore wird dies jedoch verschiedentlich als Makel angesehen und damit ein alternatives „Konzept“ motiviert

Definition (Mutex)

Ein **spezialisierter binärer Semaphor** s, der Aktion $V(s)$ nur dem Prozess, der zuvor die Aktion $P(s)$ verantwortet hat, erlaubt.

- unautorisierte Verwendung von $V(s)$ gilt als **schwerwiegender Fehler**
- der fälschlicherweise $V(s)$ durchführende Prozess ist abzubrechen!

↪ allerdings fehlt dieses Merkmal, POSIX: *an error shall be returned* ☹

[11] Prozessumschaltung innerhalb von Betriebssystemen ist typischer Kandidat dafür: Ein anderer Prozess muss den kritischen Abschnitt verlassen!

Semaphor $\mapsto$ POSIX

vgl. S. 31

- die klassischen Semaphorprimitiven von Dijkstra sind direkt abbildbar auf **semantisch äquivalente Operationen** von POSIX:

```
1  #include <semaphore.h>
2
3  typedef struct semaphore {
4      sem_t sema;
5  } semaphore_t;
6
7  inline void P(semaphore_t *sema) {
8      sem_wait(&sema->sema);
9  }
10
11 inline void V(semaphore_t *sema) {
12     sem_post(&sema->sema);
13 }
```

42

Wert für *pshared*, der hier nur ungleich 0 sein sollte, damit der betreffende Semaphor auch von „Fäden“ anderer „Prozesse“ mitbenutzbar (*shared*) ist.

- nur die Initialisierung des POSIX-Semaphors gestaltet sich anders:

`lock`	`sem_init(&pipe->data.lock.sema, 42, 1)`
`free`	`sem_init(&pipe->free.sema, 42, TABLE_SIZE)`
`full`	`sem_init(&pipe->full.sema, 42, 0)`

  - **Laufzeitinitialisierung** ist die Regel, nicht Übersetzungs- oder Bindezeit

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VI.2 Betriebssystemkonzepte: Speicher

Wolfgang Schröder-Preikschat

23. Juni 2022

Agenda

Einführung

Grundlagen
- Speicherorganisation
- Adressraum

Speicherverwaltung
- Einleitung
- Speicherzuteilung
- Speichervirtualisierung

Zusammenfassung

Gliederung

Lehrstoff

- behandelt werden grundlegende Aspekte der **Speicherorganisation**
  - Dauerhaftigkeit von Datenhaltung und die Speicherpyramide
  - von Prozessen generierte Referenzfolge innerhalb ihrer Adressräume
  - Unterschied zwischen realen, logischen und virtuellen Adressraum
- die Grundkonzepte der **Speicherverwaltung** kennenlernen
  - Speicherzuteilung und -virtualisierung differenzieren
  - Auflösung gekachelter/segmentierter Speicherverwaltung erklären
  - wesentliche Merkmale von virtuellem Speicher im Ansatz vorstellen
- Bedeutung der sogenannten **Platzierungsstrategie** vermitteln
  - kurz segmentierte und gekachelte Verfahren vorstellen
  - Suchaufwand und Verschnitt gegenüberstellen
  - wichtige Verfahren (*best-*, *worst-*, *first-*, *next-fit*) benennen
- Vertiefung „dynamischer Speicher“, die **Halde**
  - Freispeicherverwaltung auf Maschinenprogrammebene
  - Interaktion zwischen Maschinenprogramm und Betriebssystem zeigen
  - zwischen lokaler und globaler Speicherverwaltung unterscheiden

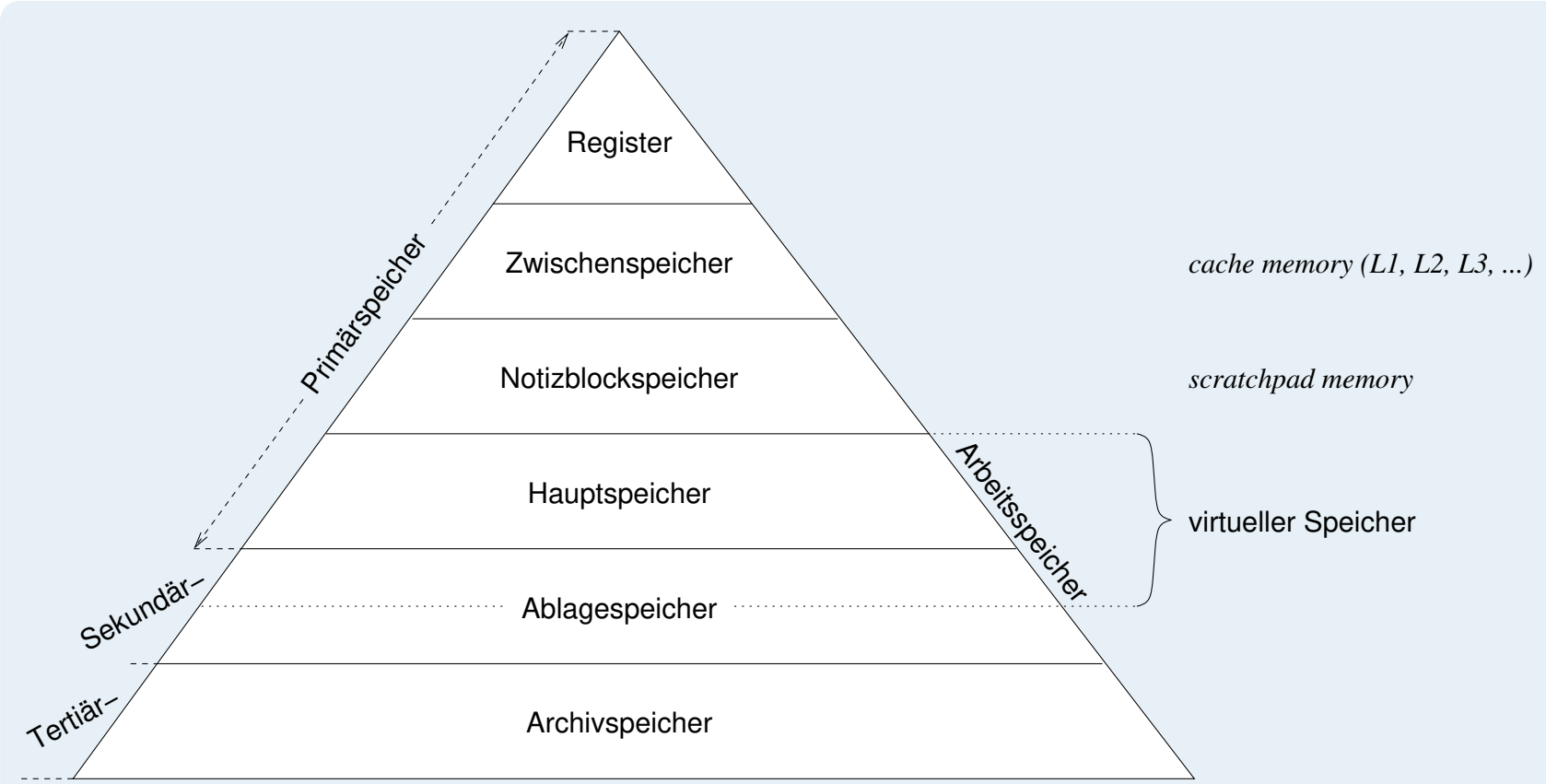
Gliederung

Definition (Speicher)

von (lat.): *spicarium* Getreidespeicher, ein Ort oder eine Einrichtung zum Einlagern von materiellen oder immateriellen Objekten.

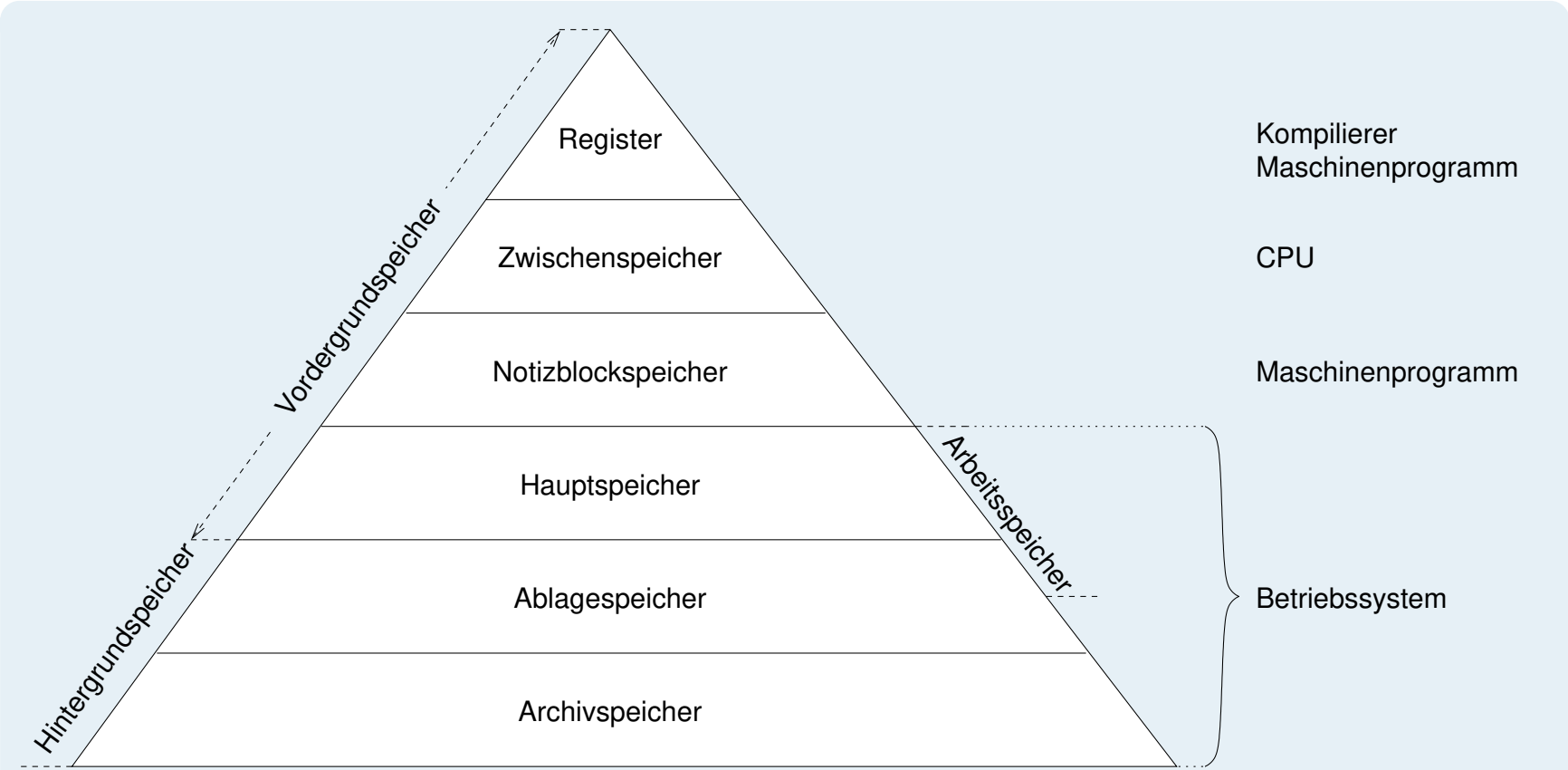
- Vorrichtung an elektronischen Rechenanlagen, die eine kurz-, mittel- oder langfristige Speicherung von Informationen ermöglicht
  - kurz
    - hunderte von ns $\leq$ *Ladungshaltung*(*RAM*) $\leq$ dutzende von s
    - **Primärspeicher**: Register-, Zwischen-, Haupt-/Arbeitsspeicher
  - mittel
    - *Flash*-/Festplattenspeicher 2–10, im Mittel 5 Jahre
    - **Sekundärspeicher**: Arbeitsspeicher, Ablagesystem (Dateien)
  - lang
    - Festplattenarchive $\leq$ 30 Jahre $\leq$ Magnetbandarchive
    - optische Speicher (DVD) vermutlich 100 Jahre
    - **Tertiärspeicher**: Archiv
  - je größer die Zeitspanne, desto größer Kapazität und Zugriffszeit
- dabei sind Haupt-/Arbeitsspeicher und bedingt auch die Ablage in den Maschinenprogrammen direkt adressierbar
  - Multics [2] bildete Dateien auf Segmente im virtuellen Adressraum ab

Speicherpyramide I



- falls **virtueller Speicher** Merkmal der Maschinenprogrammebene ist, erstreckt sich der Arbeitsspeicher auf Hauptspeicher und Ablage
  - nur die **Arbeitsmenge** an Text/Daten ist im Hauptspeicher eingelagert
  - alle anderen Bereiche sind in der Ablage (*swap area*) ausgelagert

Speicherpyramide II



- bei **Mehrprogrammbetrieb** $\mapsto$ **Adressraumisolation** verwaltet jedes Maschinenprogramm seinen eigenen **Haldenspeicher**
  - Verwaltungsfunktionen (`malloc`/`free`) stellt das Laufzeitsystem (`libc`)
  - diese interagieren mit der Speicherverwaltung des Betriebssystems

Referenzfolge

reference string

- ein **laufender Prozess** (vgl. [3, S. 20]) generiert Folgen von Adressen auf den Haupt-/Arbeitsspeicher, und zwar:
  - i nach Vorschrift des Programms, das diesen Prozess spezifiziert, wie auch
  - ii in Abhängigkeit von den Eingabedaten für den Programmablauf
- der zur Bildung dieser Adressen gegebene **Wertevorrat** hat anfangs eine feste Größe, dehnt sich gemeinhin dann aber weiter aus
  - dieser Vorrat ist initial statisch und gibt die zur Programmausführung mindestens erforderliche Menge an Haupt-/Arbeitsspeicher vor
  - zur Laufzeit ist diese **Menge** dynamisch, nimmt zu und kann dabei aber den „einem Prozess zugebilligten“ Wertevorrat nicht überschreiten
    - letzteres sichert entweder der Kompilierer oder das Betriebssystem zu
    - d.h., durch eine „typsichere Programmiersprache“ oder im Zusammenspiel mit der MMU der Befehlssatzebene
- der einem Prozess zugebillgte Adressvorrat gibt den **Adressraum** vor, in dem dieser Prozess (logisch/physisch) eingeschlossen ist
  - der Prozess kann aus seinem Adressraum normalerweise nicht ausbrechen und folglich nicht in fremde Adressräume eindringen
  - der **Prozessadressraum** hat eine durch (HW/BS) beschränkte Größe

Gedankenspiel I

- gegeben sei folgendes Programm in C:

```
1  #include <stdlib.h>
2
3  int main() {
4      while (1)
5          ((void(*)())random())();
6  }
```

- bzw. in ASM_{x86}:

```
12 main:
13     subl $12, %esp
14 .L2:
15     call random
16     call *%eax
17     jmp  .L2
```

- zu bestimmen sei die Referenzfolge eines diesbezüglichen Prozesses, unter folgender Annahme:

```
7  static void vain() { }
8
9  void (*(random)())() {
10     return vain;
11 }
```

```
18 vain:
19     rep
20     ret
21
22 random:
23     movl $vain, %eax
24     ret
```

- C: $\ldots, \overline{5, 10, 7}$
- ASM_{x86}: $\ldots, 13, \overline{15, 23, 24, 16, 19, 20, 17}$

Gedankenspiel II

- mit gdb einen Blick in den Prozessadressraum werfen, um so Einblick in den **statischen Wertevorrat** an Programmadressen zu erhalten:

```
1  (gdb) info line main
2  No line number information available for address 0x80481c0 <main>
3  (gdb) disas 0x80481c0
4  Dump of assembler code for function main:
5     0x080481c0 <+0>: push   %ebp
6     0x080481c1 <+1>: mov    %esp,%ebp
7     0x080481c3 <+3>: and    $0xfffffff0,%esp
8     0x080481c6 <+6>: xchg   %ax,%ax
9     0x080481c8 <+8>: call   0x8048300 <random>
10    0x080481cd <+13>:    call   *%eax
11    0x080481cf <+15>:    jmp    0x80481c8 <main+8>
12 End of assembler dump.
13 (gdb) disas 0x8048300
14 Dump of assembler code for function random:
15    0x08048300 <+0>: mov    $0x80482f0,%eax
16    0x08048305 <+5>: ret
17 End of assembler dump.
18 (gdb) disas 0x80482f0
19 Dump of assembler code for function vain:
20    0x080482f0 <+0>: repz ret
21 End of assembler dump.
22 (gdb)
```

Hier ist nur die aus dem Befehlsabruf resultierende, statisch leicht bestimmbare Referenzfolge gezeigt. Es fehlen Referenzen, die die Stapeloperationen (`push`, `call` und `ret`) zur Folge haben. Diese sind mangels Kenntnis der Vorgeschichte des Prozesses allein durch „Programmlektüre“ nicht ableitbar.

- für den **Befehlsabruf** ergibt sich als **Referenzfolge** des Prozesses:
  - $\ldots, 0x08048[1c0, 1c1, 1c3, 1c6, \overline{1c8, 300, 305, 1cd, 2f0, 1cf}]$

Adressraumlehre I

- Befehlssatzebene (Ebene$_2$)

Definition (realer Adressraum)

Der durch einen Prozessor definierte Wertevorrat $A_r = [0, 2^n - 1]$ von Adressen, mit $e \leq n \leq 64$ und (norm.) $e \geq 16$. Nicht jede Adresse in A_r ist jedoch gültig, d.h., A_r kann Lücken aufweisen.

- der **Hauptspeicher** ist adressierbar durch einen oder mehrere Bereiche in A_r, je nach Hardwarekonfiguration

- Maschinenprogrammebene (Ebene$_3$)

Definition (logischer Adressraum)

Der in Programm P definierte Wertevorrat $A_l = [n, m]$ von Adressen, mit $A_l \subset A_r$, der einem Prozess von P zugebilligt wird. Jede Adresse in A_l ist gültig, d.h., A_l enthält *konzeptionell* keine Lücken.

- führt **Arbeitsspeicher** ein, der linear adressierbar ausgelegt ist und durch das Betriebssystem vollständig auf den Hauptspeicher abgebildet wird

Adressraumlehre II

- Maschinenprogrammebene (Ebene$_3$)

Definition (virtueller Adressraum)

$A_v = A_l$: A_v übernimmt alle Eigenschaften von A_l. Jedoch nicht jede Adresse in A_v bildet ab auf ein im Hauptspeicher liegendes Datum.

  - Benutzung einer solchen nicht abgebildeten Adresse in A_v verursacht in dem betreffenden Prozess einen **Zugriffsfehler**
  - der Prozess erfährt eine **synchrone Programmunterbrechung** (*trap*), die vom Betriebssystem behandelt wird
  - das Betriebssystem sorgt für die **Einlagerung** des adressierten Datums in den Hauptspeicher und
  - der Prozess wird zur **Wiederholung** der gescheiterten Aktion gebracht
- der durch A_v für den jeweiligen Prozess benötigte Hauptspeicher ist „nicht in Wirklichkeit vorhanden, aber echt erscheinend“
  - jedoch steht jederzeit genügend Arbeitsspeicher für A_v zur Verfügung
    - einesteils im Hauptspeicher, anderenteils in der Ablage (*swap area*)
  - der Arbeitsspeicher ist eine virtuelle, der Hauptspeicher eine reale Größe

Gliederung

Aufgaben der Speicherverwaltung

- zentrale Aufgabe ist es, über die **Speicherzuteilung** an einen Prozess Buch zu führen und seine Adressraumgröße dazu passend auszulegen
  Platzierungsstrategie (*placement policy*)
  - wo im Hauptspeicher ist noch Platz?
- zusätzliche Aufgabe kann die **Speichervirtualisierung** sein, um trotz knappem Hauptspeicher Mehrprogrammbetrieb zu maximieren
  Ladestrategie (*fetch policy*)
  - wann muss ein Datum im Hauptspeicher liegen?

  Ersetzungsstrategie (*replacement policy*)
  - welches Datum im Hauptspeicher ist ersetzbar?
- die zur Durchführung dieser Aufgaben typischerweise zu verfolgenden Strategien profitieren voneinander — oder bedingen einander
  - ein Datum kann ggf. erst platziert werden, wenn Platz freigemacht wurde
  - etwa indem das Datum den Inhalt eines belegten Speicherplatzes ersetzt
  - ggf. aber ist das so ersetzte Datum später erneut zu laden
  - bevor ein Datum geladen werden kann, ist Platz dafür bereitzustellen

„Reviere“ einer Speicherverwaltung

- normalerweise sind die **Verantwortlichkeiten** auf mehrere Ebenen innerhalb eines Rechensystems verteilt

  Speicherzuteilung
  - Maschinenprogramm und Betriebssystem
  - Haldenspeicher, Hauptspeicher

  Speichervirtualisierung
  - ist allein Aufgabe des Betriebssystems
  - Haupt-/Arbeitsspeicher, Ablage

- das Maschinenprogramm verwaltet den seinem Prozess (-adressraum) jeweils zugeteilten Speicher **lokal** eigenständig
  - stellt dabei **sprachenorientierte Kriterien** in den Vordergrund
    - typisch für den Haldenspeicher $\rightsquigarrow$ `malloc`/`free`
- das Betriebssystem verwaltet den gesamten Haupt-/Arbeitsspeicher **global** für alle Prozessexemplare bzw. -adressräume
  - stellt dabei **systemorientierte Kriterien** in den Vordergrund
    - hilft, einen Haldenspeicher zu verwalten $\rightsquigarrow$ z.B. `mmap`/`sbrk`
- Maschinenprogramm und Betriebssystem gehen somit eine **Symbiose** ein, sie nehmen eine **Arbeitsteilung** vor
  - genauer gesagt: das Laufzeitsystem (`libc`) im Maschinenprogramm

Granularität/Auflösung der Speicherzuteilung

- je nach Haupt-/Arbeitsspeicher (inkl. Haldenspeicher) ergibt sich für die Verwaltung ein problemspezifischer Zerlegungsgrad wie folgt:
  - Wort
    - Platzierungseinheit für Hauptspeicher
      - abhängig von CPU: Vielfaches von **Byte**[1]
  - Zelle
    - Platzierungseinheit für Halden- und Hauptspeicher
      - abhängig von Laufzeit-/Betriebssystem: 8 B, 16 B, 16/32 B
  - Kachel
    - Platzierungs-, Lade- und Ersetzungseinheit für Arbeitsspeicher
      - abhängig von MMU: 512 B bis 1 GiB (typisch 4 KiB)
- diese Granulate werden zusammengefasst in **uniforme Segmente** mit Adresswerten, die einem Vielfachen der Granulatgröße entsprechen
  - wobei die Segmente die **Schutzeinheit** im logischen Adressraum bilden
  - bei entsprechender **Ausrichtung** (*alignment*) im Haupt-/Arbeitsspeicher

Platzierungsstrategie

Die Verfahren dazu sind je nach Speicherauflösung grob kategorisiert in **segmentierte** und **gekachelte Speicherverwaltung**.

[1] Das kleinste adressierbare Speicherelement: historisch 5–36 Bits breit.

Aspekte der Speicherzuteilung

segmentierte $\leftrightarrow$ gekachelte Speicherverwaltung

Segmente können verschieden groß sein, ihre jeweiligen Attribute sind Granularität, Adresse und Länge. Kacheln dagegen sind alle gleich groß, sie unterscheiden sich lediglich durch ihre Adressen.

- um die „Granulate“ im Hauptspeicher platzieren zu können, ist Buch über nicht zugeteilte Speicherbereiche zu führen
  - freie Speicherbereiche werden in einer **Löcherliste** (*hole list*) geführt
    - ein freier Platz ist Loch, Lücke, Hohlraum (*hole*) zwischen belegten Plätzen
    - wobei der für ein Listenelement benötigte Speicher das Loch selbst sein kann
      $\hookrightarrow$ Haldenspeicher: freie und belegte Plätze teilen denselben Adressraum ☺
  - die Liste ist sortiert nach Größe oder Adresse der vorhandenen Löcher
    - womit verschiedene Ziele bei der Zuteilungsstrategie verbunden sind (S. 19)
    - jedoch macht Sortierung nach Größe bei gekacheltem Speicher wenig Sinn
- wenn möglich, ist **Verschnitt** (*waste*) klein zu halten: *Zielkonflikt!*

  i einen Platz fester Größe zuteilen $\rightsquigarrow$ Segmente kacheln
    - vermeidet *externen* Verschnitt auf Kosten *internen* Verschnitts

  ii bei Freigabe zwei in A_r angrenzende Löcher zu einem Loch verschmelzen

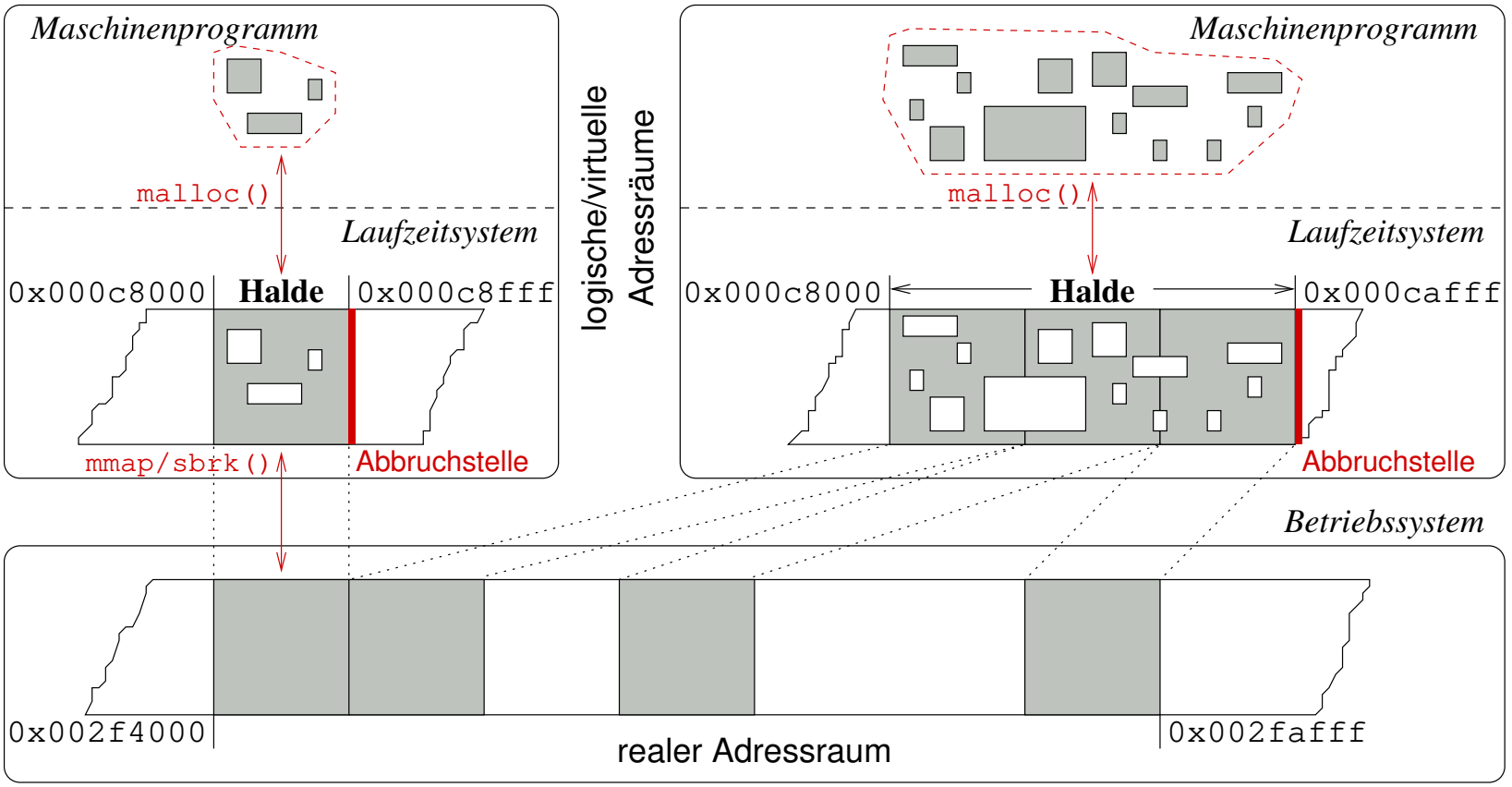
Strategien der Speicherzuteilung

- **Sortierkriterien** der Freispeicherliste und damit verbundene **Ziele**:
  - *best-fit*: zunehmende Größen, von vorne: **Verschnitt minimieren**
  - *worst-fit*: abnehmende Größen, von vorne: **Suchaufwand minimieren**
    - ↪ beide Strategien machen die Einsortierung eines anfallenden Rests und somit einen zweiten Listendurchlauf notwendig
    - ↪ aus gleichem Grund ist Verschmelzung angrenzender Löcher zu einem großen Loch in beiden Fällen aufwendig
  - *first-fit*: aufsteigende Adressen, von vorne: **Laufaufwand minimieren**
  - *next-fit*: wie zuvor, jedoch ab letzter Fundstelle: **Verschnitt nivellieren**
    - ↪ Verschmelzung angrenzender Löcher zu einem großen Loch ist in beiden Fällen unaufwendig
    - ↪ beide Strategien tendieren dazu, große Löcher zu zerschlagen und dadurch mehr Verschnitt zu generieren (FF mehr als NF)

Auch eine schwere Tür hat nur einen kleinen Schlüssel nötig. (Dickens)

- die „Einfachheit“ macht *first-* und *next-fit* zu guten Kompromissen. . .
  - vgl. Anhang, S. 32 – 36

Synergie bei der Speicherzuteilung



- das **Laufzeitsystem** verwaltet Speicher, der dem Adressraum eines einzelnen Maschinenprogramms vom Betriebssystem zugeteilt wurde
- das **Betriebssystem** verwaltet Speicher, der den Adressräumen aller Maschinenprogramme zugeteilt worden ist oder werden kann

Speicherrückgabe

. . . macht sehr wohl Sinn!

Speicherrückgabe durch `free` an das Betriebssystem ist nicht nötig, da bei virtuellem Speicher ungenutzter Speicher schon rückgewonnen wird — kolportiert die Informatikfolklore.

- jede Implementierung virtuellen Speichers basiert auf **Schätzungen**
  - Rückgewinnung ungenutzten Speichers leistet die Ersetzungsstrategie
  - alle dazu bekannten Verfahren greifen auf **Heuristiken** zurück
  - ob Speicher endgültig ungenutzt ist, weil er frei ist, bleibt daher ungewiss
  - nur das Maschinenprogramm selbst kann darüber Gewissheit haben
- eine Folge daraus ist, dass **Programmierfehler** unentdeckt bleiben
  - Adressen zu ungenutztem Haldenspeicher sind im Adressraum noch gültig
  - nur Rückgabe ans Betriebssystem kann diese Adressen ungültig machen
- zudem verursacht diese Heuristik **nichtdeterministische Prozesse**
  - eine kontraproduktive Eigenschaft für (festen/harten) Echtzeitbetrieb
  - deshalb implementiert nicht jedes Betriebssystem virtuellen Speicher. . .

Verschmelzung und Kompaktifizierung

Ist notwendig bzw. wünschenswert für große, rückgabefähige Löcher.

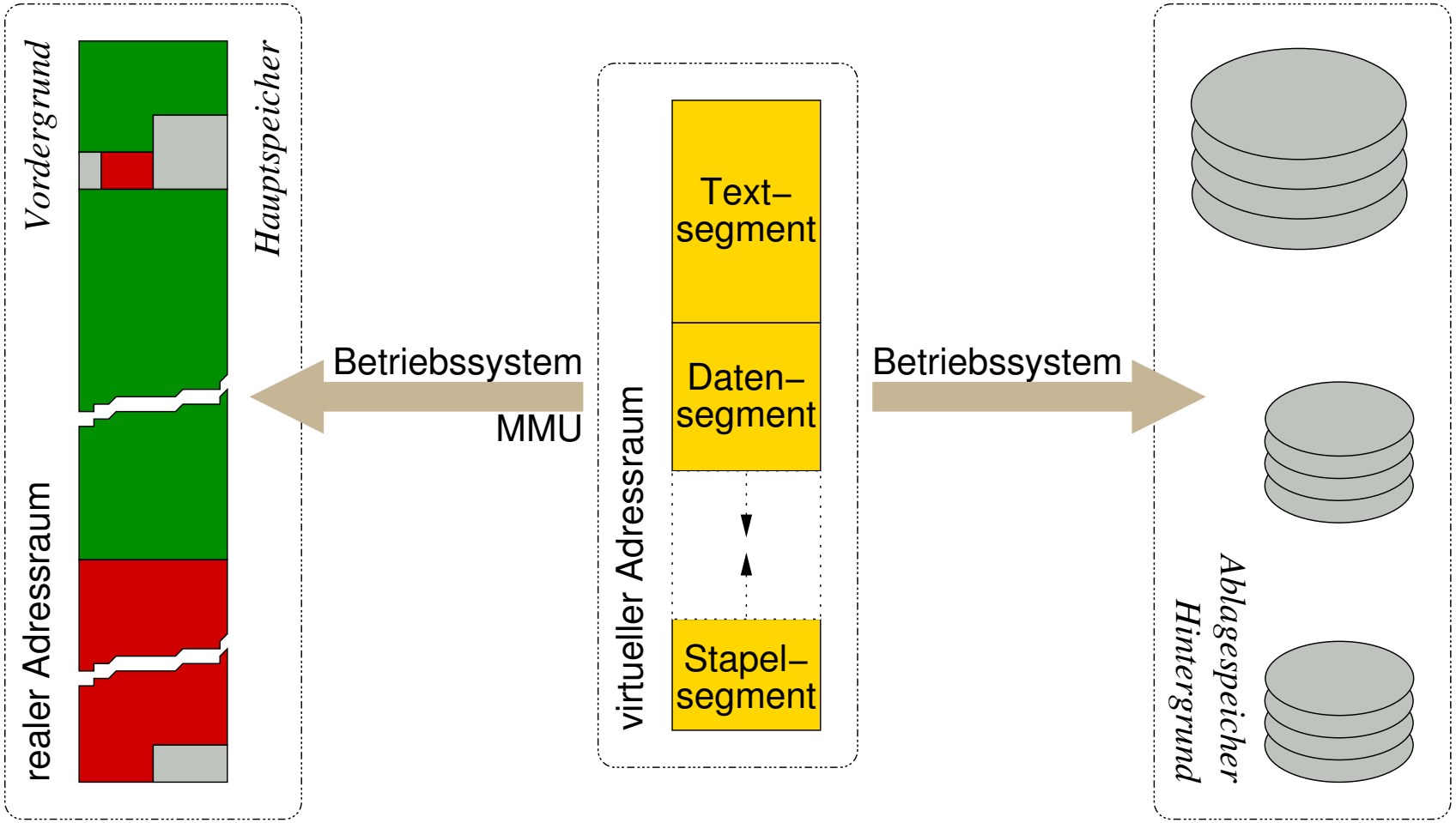
Virtueller Speicher

*Eine **Betriebssystemtechnik**, die laufende (ggf. nichtsequentielle) Prozesse ermöglicht, obwohl ihre Maschinenprogramme samt Daten nicht komplett im Hauptspeicher liegen.*

- sobald und solange Spannung anliegt, erfordert die Befehlssatzebene (d.h., der reale Prozessor) einen kontinuierlichen Befehlsstrom
  - anderenfalls gelingt der Befehlsabruf- und -ausführungszyklus nicht
- aus welchen Programmabläufen sich dieser Befehlsstrom ergibt, ist für die Befehlssatzebene jedoch nicht von Bedeutung
  - Abläufe innerhalb eines realen/logischen Adressraums erfordern, dass die betreffenden Programme vollständig im Hauptspeicher vorliegen
    - jede Adresse muss auf ein im Hauptspeicher vorliegendes Datum abbilden
  - im Gegensatz zu Abläufen innerhalb eines virtuellen Adressraums, der die **partielle Abbildung** einer Adresse auf ein Datum ermöglicht (vgl. S. 13)
- die durch die **Lokalität** eines Prozesses definierte **Referenzfolge** gibt die Adressen vor, die auf den Hauptspeicher abzubilden sind
  - alle anderen Adressen bilden ab auf die Ablage (*swap area*)

Partielle Abbildung virtueller Adressräume

(vgl. S. 13)



- Abbildungseinheit ist eine **Seite** (*page*), die Strukturierungselement sowohl des logischen als auch des virtuellen Adressraums ist
  - sie passt exakt auf eine Kachel, auch **Seitenrahmen** (*page frame*)

Abbildungstabelle

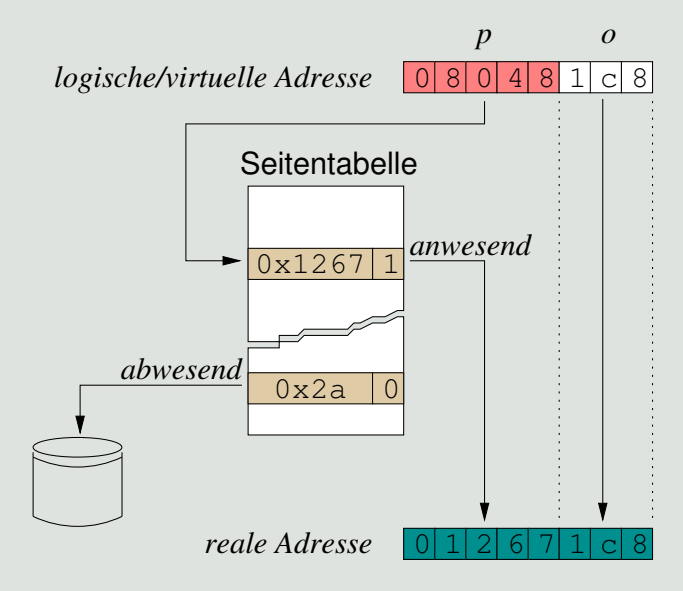
f : Seiten $\mapsto$ Seitenrahmen d.h. Kacheln

- eine ein- oder mehrstufig organisierte **Seitentabelle** (*page table*)
  - ein Feld (*array*), Datentypelement „**Seitendeskriptor**“ (*page descriptor*)
    - definiert durch das Betriebssystem, verarbeitet von der MMU
  - im Deutschen auch bezeichnet als „Seiten-Kachel-Tabelle“
- indiziert durch die **Seitennummer** (*page number*) einer Adresse
  - jede logische/virtuelle Adresse bildet ein Tupel $A = (p, o)$
  - mit Seitennummer p und Versatz (*offset*) o innerhalb dieser Seite
    - Wertevorrat $o = [0, 2^i - 1]$, mit $9 \leq i \leq 30$ (vgl. S. 17)
    - Wertevorrat $p = [0, 2^{n-i} - 1]$, mit $32 \leq n \leq 64$
    - mit p als Indexwert liest die MMU den A abbildenden Seitendeskriptor
- der Seitendeskriptor enthält die für die Abbildung nötigen **Attribute**
  - Informationen über den gegenwärtigen Zustand der Seite
    - anwesend, referenziert, modifiziert, schreibbar, ausführbar, zugänglich, ...
  - Kachelnummer/-adresse im Hauptspeicher (falls anwesend, eingelagert) oder Blocknummer in der Ablage (falls abwesend, ausgelagert)
- jeder Zugriff auf eine abwesende, ausgelagerte Seite verursacht einen **Seitenfehler** (*page fault*) $\rightsquigarrow$ Teilinterpretation des Zugriffs

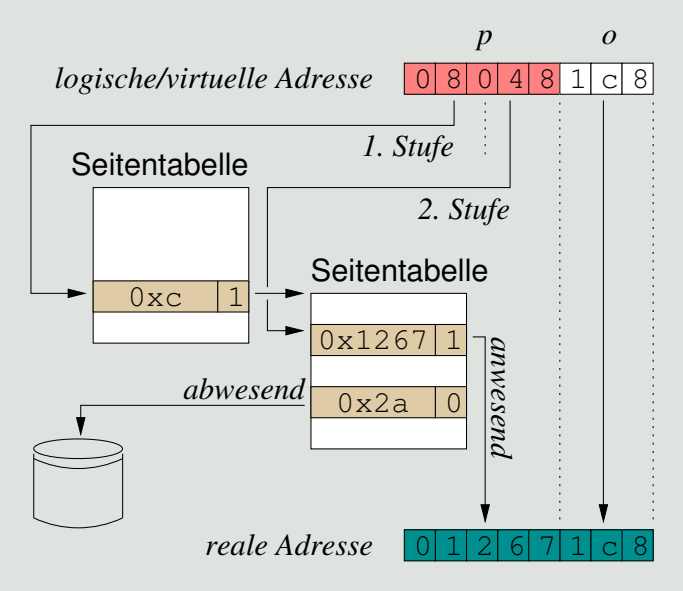
Abbildung (Prinzip)

- angenommen, die CPU ruft folgenden Befehl zur Ausführung ab:
  `0x080481c8 <+8>: call 0x8048300 <random>` (vgl. S. 11)

  - einstufige Abbildung



  - zweistufige Abbildung (x86)



  - *base/limit* Registerpaar (MMU) grenzt die Seitentabelle ein
  - verschieden große Seitentabellen

  - *base* Register (MMU) lokalisiert die Seitentabelle der 1. Stufe
  - gleich große Seitentabellen

↪ ***Trap***, falls $p \geq limit$ (li.) oder ungültiger/leerer Seitendeskriptor (beide)

Gliederung

Resümee

- behandelt wurde die **Speicherorganisation** von Rechensystemen
  - Primär-, Sekundär- und Tertiärspeicher als die drei Hauptkategorien
  - Verfeinerungen sind Haupt-, Arbeitsspeicher und Ablage
  - Bausteine von Vorder- und Hintergrundspeicher zur Programmausführung
  - bilden Ebenen einer **Speicherpyramide** (auch: Speicherhierarchie)
- die für Speicherverwaltung relevante **Adressraumlehre** präsentiert
  - **Referenzfolgen** sind wichtig, um virtuellen Speicher zu begreifen
  - Bezugspunkt dabei ist die **Adressraumart**: real, logisch, virtuell
  - logischer und virtueller Adressraum sind zwei verschiedene Konzepte:

    totale Abbildung $f : A_l \to A_r$ von logischen auf realen Adressen

    partielle Abbildung $f : A_v \rightsquigarrow A_r$ von virtuellen auf realen Adressen
- Aufgaben der Speicherverwaltung sind in **Politiken** untergliedert
  - Platzierungs-, Lade- und Ersetzungsstrategie
  - erstere meint Speicherzuteilung, letztere beiden Speichervirtualisierung
  - virtueller Speicher ermöglicht Prozesse unvollständiger Programme
- das **Ausmaß** eines virtuellen Adressraums kann riesig sein
  - der Hauptspeicher im realen Adressraum ist indes verschwindend klein. . .

Literaturverzeichnis I

[1] Chase, J. S. ; Levy, H. M. ; Freeley, M. J. ; Lazowska, E. D.:
Sharing and Protection in a Single-Address-Space Operating System.
In: *Transaction on Computer Systems* 12 (1994), Nov., Nr. 4, S. 271–307

[2] Daley, R. C. ; Dennis, J. B.:
Virtual Memory, Processes, and Sharing in MULTICS.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 306–312

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 6.1

Abbildungsfunktionen I

Seitendeskriptor und Bildung einer „eingerahmten“ realen Adresse:[2]

```
1  #define PAGE_SIZE (1<<12)    /* 4KB per page and page frame, resp. */
2
3  typedef struct page {
4      unsigned present:1;      /* true iff in mainstore, false otherwise */
5      unsigned reserved:11;    /* more page attributes (for further study in SP2) */
6      unsigned frame:20;       /* page-aligned mainstore address: page frame number */
7  } page_t;
8
9  inline void *match(page_t *page, void *addr) {
10     return (void *)((page->frame * PAGE_SIZE) | ((size_t)addr & (PAGE_SIZE - 1)));
11 }
```

Seitentabelle und Ableitung des Seitendeskriptors aus einer Adresse:

```
12 typedef struct map {
13     page_t *table;           /* page table base address */
14     size_t  limit;           /* last valid page table entry */
15 } map_t;
16
17 extern map_t map;           /* page table base/limit register pair */
18
19 inline page_t *probe(map_t *map, void *addr) {
20     unsigned pano = ((unsigned)addr) / PAGE_SIZE;
21     if (pano <= map->limit)          /* valid page number or address, resp. */
22         return &map->table[pano];    /* read page descriptor */
23     trap(SEGMENTATION_FAULT);        /* invalid page number: raise exception to OS */
24 }
```

[2] Mit möglichem Blocknummernwertevorrat $[0, 2^{31} - 1]$ für abwesende Seiten.

Abbildungsfunktionen II

- partielle Abbildung einer virtuellen Adresse, $f : A_v \rightsquigarrow A_r$ (vgl. S. 13)

```
1  void *v2r(void *addr) {
2      do {
3          page_t *page = probe(&map, addr);
4          if (page->present)  /* placed in mainstore */
5              return match(page, addr);
6          trap(PAGE_FAULT);   /* absent: raise excpetion to OS */
7      } while (1);            /* retry mapping */
8  }
```

  - gültige Seiten eines virtuellen Adressraums sind ein- oder ausgelagert
  - sind sie ausgelagert, wird ein **Seitenfehler** (*page fault*) angezeigt
  - Folge ist die **Seitenumlagerung** (*paging*) aus der Ablage (*swap area*) heraus und hinein in den Hauptspeicher

- totale Abbildung einer logischen Adresse, $f : A_l \rightarrow A_r$ (vgl. S. 12)

```
9  void *l2r(void *addr) {
10     return match(probe(&map, addr), addr);
11 }
```

  - im Unterschied zur partiellen Abbildung müssen die gültigen Seiten eines logischen Adressraums immer eingelagert sein
  - Seitenfehler gibt es hier nicht, jedoch bleibt der **Speicherzugriffsfehler** (*segmentation fault*: vgl. S. 29, Zeile 23)

Umfang eines virtuellen Adressraums

- mit N für die **Adressbreite** (einer virtuellen Adresse) in Bits:

N	Adressraumgröße (2^N Bytes)	Dimension			
16	65 536	64 kibi	(2^{10})	kilo	(10^3)
20	1 048 576	1 mebi	(2^{20})	mega	(10^6)
32	4 294 967 296	4 gibi	(2^{30})	giga	(10^9)
⋮			⋮		⋮
48	281 474 976 710 656	256 tebi	(2^{40})	tera	(10^{12})
64	18 446 744 073 709 551 616	16 384 pebi	(2^{50})	peta	(10^{15})

- ein einziger virtueller Adressraum kann so riesig sein, dass es schnell an Ablageplatz fehlt, um ausgelagerte Seiten zu speichern ☹
- darüber hinaus können seine Adressen ewig gültig sein...

> *A full 64-bit address space will last for 500 years if allocated at the rate of one gigabyte per second. [1, S. 272]*

Haldenspeicher I

first-fit

```
1  #include <stddef.h>
2
3  typedef struct cell {
4      struct cell *next;
5      size_t size;
6  } cell_t;
```

```
7   extern cell_t *list;
8
9   extern cell_t *acquire(size_t);
10  extern size_t  release(cell_t *);
```

- freien Speicher erwerben, einen passenden Bereich ausfindig machen:

```
11 cell_t *acquire(size_t want) {
12     size_t need = ((want + sizeof(cell_t) - 1) / sizeof(cell_t)) + 1;
13     cell_t **link = &list, *cell;
14
15     while ((cell = *link)) {              /* one/another hole listed, check it */
16         if (cell->size < need)            /* bad luck, it's too small */
17             link = &cell->next;           /* continue search... */
18         else {
19             if (cell->size > need) {                  /* hole is large enough */
20                 cell->size -= need;                   /* shorten it */
21                 (cell + cell->size)->size = need;     /* remember cell size */
22                 return (cell + cell->size + 1);       /* deliver initial address */
23             }
24             *link = cell->next;           /* hole fits exactly, remove from list */
25             return cell + 1;              /* deliver initial address */
26         }
27     }
28
29     return 0;
30 }
```

Next-fit *führt einen* `last`*-Zeiger ein, der den zuletzt untersuchten Listeneintrag (*`link`*) vermerkt. Suchläufe beginnen bzw. enden bei* `last`*, mit Umgriff am Listenende auf den Listenanfang (*`list`*).*

Haldenspeicher II

- Speicher freigeben, wenn möglich mit Nachbarlöchern verschmelzen:

```
31  size_t release(cell_t *hole) {
32      size_t free = 0;
33
34      if (hole-- && hole->size) {          /* start at cell head (descriptor) */
35          cell_t **link = &list;
36
37          while (*link && ((*link)->next < hole)) /* find location on list */
38              link = &(*link)->next;               /* continue search... */
39
40          if (*link) {          /* location found, try to merge */
41              if (((*link) + (*link)->size) == hole)      /* preceeding hole, merge */
42                  free = ((*link)->size += hole->size);
43              if ((hole + hole->size) == (*link)->next)   /* succeeding hole, merge */
44                  free = ((*link)->size += (*link)->next->size);
45          }
46
47          if (!free) {          /* merging was not possible, add hole to free list */
48              free = hole->size;           /* remember amount of freeing */
49              hole->next = *link;          /* link with successor hole */
50              *link = hole;                /* enlist freed hole */
51          }
52      }
53      return free * sizeof(cell_t);        /* deliver number of bytes freed */
54  }
```

- neben der Zellengröße ist der Test auf ein **Kennzeichen** sinnvoll, um „falsche“ Zellen zumindest ansatzweise erkennen zu können
- als Platzhalter des Kennzeichens bietet sich das `next`-Attribut an

Haldenspeicher III

- Nachbildung der funktionalen Eigenschaften von `malloc` und `free`:

```
1  #include "cell.h"
2
3  void *malloc(size_t want) {
4      cell_t *cell = acquire(want);    /* try request */
5      if (!cell) {                     /* failed, no hole found */
6          cell = enlarge(want);        /* ask for more memory */
7          if (cell)                    /* great, OS supplied */
8              cell = acquire(want);    /* retry request */
9      }
10     return cell;
11 }
12
13 void free(void *cell) {
14     size_t size = release((cell_t *)cell);
15     if (size)              /* new free memory released */
16         reclaim(size);     /* give operating system (OS) a hint */
17 }
```

- im Unterschied zum Original, teilt `free` dem Betriebssystem mit, dass Haldenspeicher endgültig rückgewonnen werden kann
- Informatikfolklore hält diesen Hinweis unnötig bei virtuellem Speicher

Interaktion Maschinenprogramm/Betriebssystem I

- UNIX-kompatible Betriebssystemschnittstelle nutzen:

```
1  #include <unistd.h>
```

- dynamischen Speicher vergrößern, logischen Adressraum verlängern:

```
2  cell_t *enlarge(size_t want) {
3      size_t size = ((want + getpagesize() - 1) / getpagesize()) * getpagesize();
4      cell_t *cell = (cell_t *)sbrk(size);     /* new program break */
5
6      if ((int)cell != -1) {
7          cell->next = 0;                      /* only cell this section */
8          cell->size = size / sizeof(cell_t); /* make cell size */
9          release(cell + 1);                   /* add section to free list */
10     }
11
12     return (int)cell == -1 ? 0 : cell;
13 }
```

- das Beispiel greift mit `sbrk` einen recht alten Ansatz auf,[3] der das Ende des Datensegments eines Maschinenprogramms verschiebt

  nach hinten – um mehr Speicher vom Betriebssystem zu erhalten

  nach vorne – um dem Betriebssystem Speicher zurück zu geben

- verändert wird die **Abbruchstelle** im Programm (*program break*), die Adresse der ersten Speicherstelle jenseits des gültigen Datensegments

[3] Nicht zuletzt, um Spielraum zum Selbststudium zu lassen. ☺

Interaktion Maschinenprogramm/Betriebssystem II

- Größenangabe auf Tauglichkeit zur Speicherrückgabe prüfen:

```
14  int adapted(size_t size) {
15      return ((size / getpagesize()) * getpagesize()) == size;
16  }
```

- dynamischen Speicher zurückgeben, logischen Adressraum verkürzen:

```
17  void *reclaim(size_t size) {
18      if (adapted(size)) {
19          cell_t **link = &list;
20
21          while ((*link)->next != 0)       /* skip to end of data section */
22              link = &(*link)->next;       /* BTW: a tail pointer is a good idea... */
23
24          if (((*link)->size * sizeof(cell_t) == size)     /* hole size fits and */
25          &&  ((*link) + (*link)->size) == sbrk(0)) {      /* hole is before break */
26              void *hole = *link;                          /* save return value */
27              sbrk(-((*link)->size * sizeof(cell_t)));     /* new program break */
28              *link = 0;                                   /* new list tail item */
29              return hole;
30          }
31      }
32      return 0;
33  }
```

- bei *first/next-fit* könnte die Adresse des letzten gelisteten Lochs eine neue Abbruchstelle im Maschinenprogramm markieren
- dazu muss die dem Loch folgende Adresse der aktuellen Abbruchstelle entsprechen und die Lochgröße muss Vielfaches der Kachelgröße sein

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VI.3 Betriebssystemkonzepte: Adressbindung

13. Juni 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

- Einführung
  - Semiotik
  - Informatik
- Namensauflösung
  - Speicheradressen
  - Pfadnamen
- Symbolauflösung
  - Übersetzer
  - Binder
  - Lader
- Zusammenfassung

Gliederung

- ein **Vorgang**, der die Adresse eines Adressraums $\mathbb{X}$ an die Adresse eines Adressraums $\mathbb{Y}$ bindet
  - die Adresse in $\mathbb{X}$ ist eine reale, physische Adresse
  - die Adresse in $\mathbb{Y}$ ist eine logische, virtuelle oder symbolische Adresse

  $\hookrightarrow$ eine Abbildung von einem Adressraum auf einen anderen Adressraum
- die **Bindung** geschieht zu verschiedenen Zeitpunkten, jedoch immer ist das Ziel, **absolute Adressen** zu generieren

Übersetzungszeit
- der Programmquelltext enthält symbolische Adressen
- übersetzt mit **Kompilierer** Ebene$_5$

$\hookrightarrow$ *compile time binding*

Ladezeit
- Programme enthalten relative/verschiebbare Adressen
- verlagert mit **Lader** Ebene$_3$

$\hookrightarrow$ *load time binding*

Ausführungszeit
- Programme enthalten relative/verschiebbare Adressen
- verlagert mit **Adressumsetzungseinheit**[1] Ebene$_2$

$\hookrightarrow$ *execution time binding*, *run-time binding*

[1] Allgemein ein Interpreter, speziell das Betriebssystem und die MMU.

Name $\Longleftrightarrow$ Symbol $\Longleftrightarrow$ Adresse

Definition (`www.duden.de`)

Kennzeichnende Benennung eines Einzelwesens, Ortes oder Dinges, durch die es von anderen seiner Art unterschieden wird; Eigenname.

- die Bezeichnung von einer **Entität** innerhalb eines Rechensystems, die „nach außen“ zugänglich sein müssen
  - allgemein Stellen im Haupt-, Arbeitsspeicher und in der Ablage
  - zu referenzierende Bezugspunkte von Programmtext und -daten
- dabei hat diese Bezeichnung nur innerhalb von einem bestimmten **Kontext** eine wohldefinierte und eindeutige Bedeutung
  - innerhalb eines (realen, logischen, virtuellen) Adressraum, Dateisystems
- es werden grundlegende **Abbildungsfunktionen** behandelt, die Bezeichnungen in **Adressen** umwandeln
  - Seitenadressierung, Segmentierung und Kombinationen davon
  - Indexierung bei Dateisystemen: Indexknotentabelle, Verzeichnis, Dateien
  - Symbolverwaltung von Kompilierer, Assemblierer, Binder und Lader

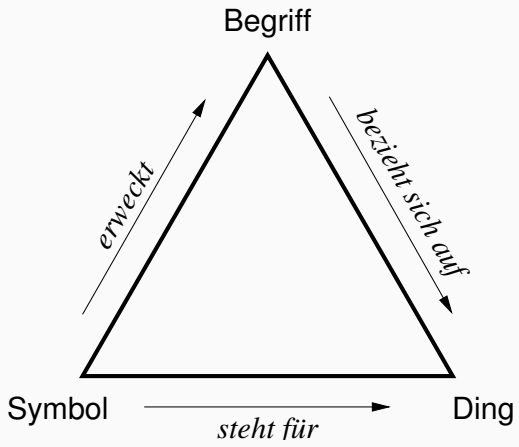
Einführung

Semiotik

Semiotisches Dreieck

Beziehung zwischen Benennung (*Bezeichnung*), Begriff (*Bedeutung*) und Gegenstand (*Bezeichnetes*).

- Benennung meint die Versprachlichung einer Vorstellung
  - ruft Begriffe ins Bewusstsein
  - bezeichnet einen Gegenstand, ein Ding
- Bezeichnung umfasst insbesondere auch nichtsprachliches wie:
  - **Symbole** und **Nummern**



- Bezeichnetes ist ein Objekt — nicht nur im informatischen Sinne
  - mit einem **Namen** versehen, benannt durch ein oder mehrere Wörter
- die Benennung ist sprachlich richtig und treffend auszulegen
  - möglichst genau und dabei knapp zugleich
  - am Sprachgebrauch orientieren
  - auf **Fachsprache** bezogen, nicht Flexibilität brauchende Gemeinsprache

Einführung

Informatik

Bezeichnung

- **Dienstprogramm** (*utility*) zum Nachschlagen einer Internetadresse

```
1   #include <sys/socket.h>
2   #include <netinet/in.h>
3   #include <arpa/inet.h>
4
5   #include <netdb.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8
9   int main(int argc, char **argv) {       /* greatly simplified variant of host(1) */
10      if (argc == 2) {
11          struct hostent *host = gethostbyname(argv[1]);
12          if (host != NULL) {
13              unsigned int i = 0;
14              printf("%s = ", host->h_name);
15              while (host->h_addr_list[i] != NULL)
16                  printf("%s ", inet_ntoa(*(struct in_addr*)(host->h_addr_list[i++])));
17              printf("\n");
18          }
19      }
20  }
```

  - darin werden verschiedene Dinge symbolisch bezeichnet
    - das Programm `main` samt Variablen `argc`, `argv`, `host` und `i`
    - sowie die Unterprogramme `gethostbyname`, `printf` und `inet_ntoa`
  - hinzu kommen Bezeichnungen, die in der Hand der Umgebung liegen
    - der Dateiname `host.c` und Pfadname `./a.out` bzw. `./host`
    - sowie eine Internetadresse als Programmparameter `argv[1]`

Auflösung von Namen

…in Ziffern

- durch Kompilierer, Assemblierer und Binder ($\text{Ebene}_5 \mapsto \text{Ebene}_3$)
  - symbolische (Text, Daten) auf numerische Referenz
- durch Betriebssystem ($\text{Ebene}_3 \mapsto \text{Ebene}_2$)
  - numerische Referenz auf virtuelle, logische oder reale Adresse
    - insbesondere auch Prozesskennung (PID) auf Prozesskontrollblock
    - aber ebenso Dateizeiger (`FILE*`) auf Dateideskriptor
    - allgemein/abstrakt: eine Handhabe (*handle*) auf eine Systemressource
  - symbolische auf numerische Referenz[2]
    - Pfadname (Verzeichnis, Datei) auf Indexknotennummer (*inode number*)
    - Internetadresse (URL) auf Netzwerkadresse (IP-Adresse)

Gemeinsamkeit: *Symbol* $\mapsto$ *Nummer*

Dies trifft auch zu auf die Herleitung einer virtuellen, logischen oder realen Adresse aus einer numerischen Referenz. Denn letztere ist als Zwischenschritt zu begreifen, der Dinge eines symbolisch formulierten Programms eine numerische Identität gibt. Daher ist die Abbildung $\text{Ebene}_5 \mapsto \text{Ebene}_2$ ganzheitlich zu sehen:

$\hookrightarrow$ symbolische Referenz auf virtuelle, logische oder reale Adresse.

[2] Die ggf. auf eine Speicheradresse abgebildet wird, wie eben geschildert.

Gliederung

Namensauflösung

Speicheradressen

Adresse als Synonym

Definition ([1, S. 157–158])

- *An address used by the programmer is called a “name” or a “virtual address,” and the set of such names is called the address space, or name space.*
- *An address used by the memory is called a “location” or “memory address,” and the set of such locations is called the memory space.*
- *Since the address space is regarded as a collection of **potentially** usable names for information items, there is no requirement that every virtual address “represent” or “contain” any information.*

- die Deutung ist **kontextabhängig**, ebenso was die Adresse bezeichnet
  - derselbe Name in verschiedenen Namensräumen kann verschiedene Orte (nicht nur in einem Rechensystem) adressieren
  - derselbe Ort kann über verschiedene Namen adressiert werden

Ortstransparente Namen

- Adressen, die **Speicherorte** bezeichnen, können realer, logischer oder virtueller Natur sein

  real — muss exakt dort liegen, intern
  logisch — kann woanders liegen, intern
  virtuell — kann woanders liegen, intern oder extern

  Den Ziffern allein ist der wirkliche Ort nicht anzusehen.

- als **interner Ort** ist ein Platz im Hauptspeicher gemeint, identifiziert durch eine Adresse im realen Adressraum
  - insbesondere die „anwesende Seite“ im Falle von Speichervirtualisierung
- demgegenüber drückt **externer Ort** aus, dass der Platz irgendwo im System aber eben nicht im Hauptspeicher liegt
  - in der Ablage oder im Hauptspeicher eines anderen Rechensystems
  - beide gegebenenfalls nur indirekt über ein Rechnernetz zugänglich
- dies schließt **speicherabgebildete** (*memory-mapped*) **Dinge** mit ein, also im logischen/virtuellen Adressraum platzierte Objekte
  - wie etwa Geräteregister, Bildspeicher oder **Dateien**
  - der Zugriff läuft dann über Lese-/Schreibaktionen der Befehlssatzebene

Organisation des Namensraums

- **Seitenadressierung** (*paging*) mittels **Seitentabelle** [5, S. 29–30]
  - jede von der CPU generierte Adresse wird gedeutet als $A_p = (p, o)$, wobei

    Versatz $o = [0, 2^w - 1]$, mit $9 \leq w \leq 30$ (*offset*)

    Seitennummer $p = [0, 2^{n-w} - 1]$, mit $32 \leq n \leq 64$, Tabellenindex
  - eine gewöhnliche **lineare Adresse** $\rightsquigarrow$ **eindimensionaler Adressraum**
    - d.h., Oktetts oder Worte in einer Dimension aufgereiht
- **Segmentierung** (*segmentation*) mittels **Segmenttabelle**
  - jede Adresse ist repräsentiert als Zweitupel $A_s = \langle s, d \rangle$, wobei

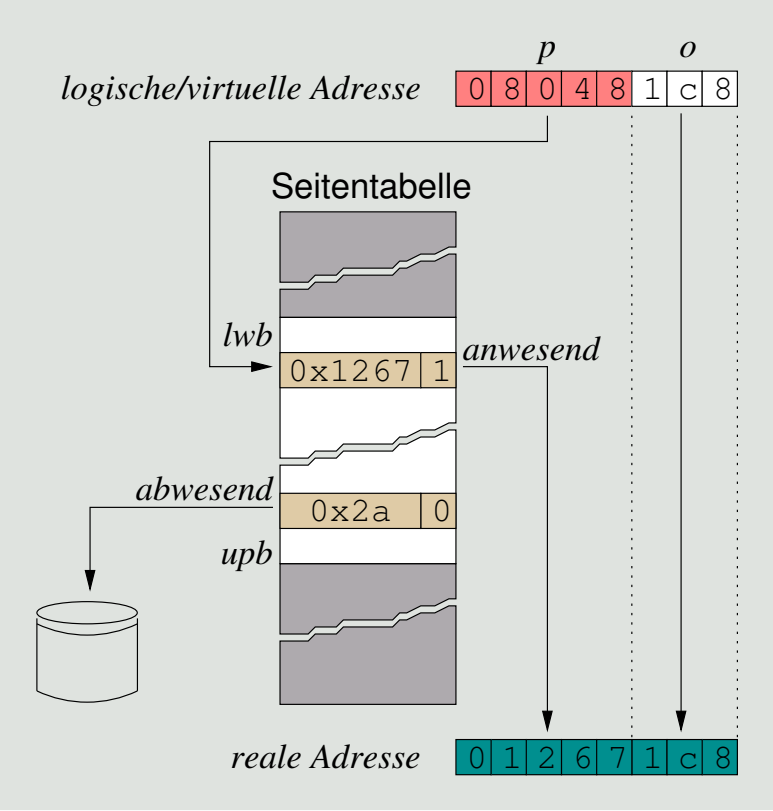
    Segmentname $s = [0, 2^m - 1]$, mit $12 \leq m \leq 18$, Tabellenindex

    Verschiebung $d = [0, 2^n - 1]$, mit $32 \leq n \leq 64$ (*displacement*)
  - Zweikomponentenadresse $\rightsquigarrow$ **zweidimensionaler Adressraum**
    - d.h., Segmente in der ersten und Segmentinhalte in der zweiten Dimension
- Kombination:
  - **segmentierte Seitenadressierung** (*segmented paging*)
    - die Seitentabellen sind segmentiert, d.h., $A_p = (p, o)$ mit $p = (s, d)$
  - **seitennummerierte Segmentierung** (*paged segmentation*)
    - die Segmente sind seitennummeriert, d.h., $A_s = \langle s, d \rangle$ mit $d = (p, o)$ oder die Segmenteinheit generiert eine lineare Adresse A_p für die Seiteneinheit

Seitenadressierung

paging

- auf Basis einer ein-/mehrstufigen **Seitentabelle** als statisches Feld:



- p ist **Indexwert** für gültige Seiten im Bereich $[P_{lwb}, P_{upb}]$
  - sei $P = 2^{n-i}$ max. Seitenanzahl
  - dann gilt $0 \leq P_{lwb} < P_{upb} \leq P - 1$
- ein möglicher **Indexfehler** muss erkannt werden
  - Tabelle auffüllen mit Einträgen, die Abbildungsfehler erzwingen
  - Grenzwertprüfung auf Basis eines *limit*-Registers ist unüblich
- P hängt ab von Seitengröße 2^i und bestimmt die Stufenanzahl

- ein **Seitenfehler** (*page fault*) bedeutet damit verschiedenerlei:
  - gültig falls $P_{lwb} \leq p \leq P_{upb}$, dann ist p abwesend oder ungenutzt
    - eine ungenutzte Seite ist gültig, sie wurde nur noch nicht abgebildet
  - sonst ungültig: die betreffende Seite gehört nicht zum Prozessadressraum

Namensauflösung

Pfadnamen

Adresse als Standort

…einer bestimmten Bestandsart

- gemeint sind Standortnamen von **Dateien** in der **Ablage**, aber auch zur Bezeichnung lokaler **Betriebsmittel** oder Systemstrukturen
  - ursprünglicher Bezugspunkt ist die Datei (*file*), d.h., eine abgeschlossene Einheit zusammenhängender Daten
  - dieses **speicherzentrische Betriebsmittel** ist aber nur das Beispiel einer einzelnen Bestandsart, andere sind etwa:

    Kommunikationsmittel – Kanal (*pipe*), Sockel (*socket*), Briefkasten

    Gerät – zeichen-, block-, stromorieniert

    Zustandsdaten – Prozesstabelle, Adressraumbelegung, …
  - allgemein sind so einige anwendungsrelevante und durch Betriebssysteme bereitgestellte **Typen** namentlich zugänglich
- wichtiger Aspekt in dem Zusammenhang ist das **Verzeichnis** solcher Namen und die zugrunde gelegte Organisationsform
  - Struktur eines Namensraums in lokaler und globaler Hinsicht
  - Art der Verknüpfung zwischen Namen und dem benannten Ding
- ist **Persistenz** von Namen, Verzeichnissen und Abbildungen verlangt, bietet ein **Dateisystem** eine adäquate Grundlage

Benennung

…von Exemplaren bestimmter Betriebsmitteltypen

`https://de.wikipedia.org/wiki/Benennung`

Bezeichnung eines Gegenstands durch ein Wort oder mehrere Wörter.

- gedeutet in Bezug auf grundlegende Betriebssystemkonzepte, die mit Multics [8] eingeführt wurden:

  ein Wort
  - der **Name** relativ zu einem bestimmten Kontext
  - lokal (in seinem Kontext) eindeutig, global mehrdeutig

  mehrere Wörter
  - der **Pfadname** im Namensraum zum benannten Ding
  - global eindeutige Bezeichnung des Namenskontextes

- die **Mehrwortbenennung** sieht einen „Trenntext“ als **Separator** vor, den die Namensverwaltung im Betriebssystem definiert, z.B.:

  `>` Multics (*greater-than*)
  `/` UNIX (*slash*)
  `\` Windows (*backslash*)

- wohingegen jedoch die Namen selbst für die Namensverwaltung ohne Bedeutung sind und i.A. von ihr nicht interpretiert werden[3]

[3] „*Name ist Schall und Rauch.*“ (Goethe: Faust I, Vers 3456 f.)

Alphabet und Wortlänge

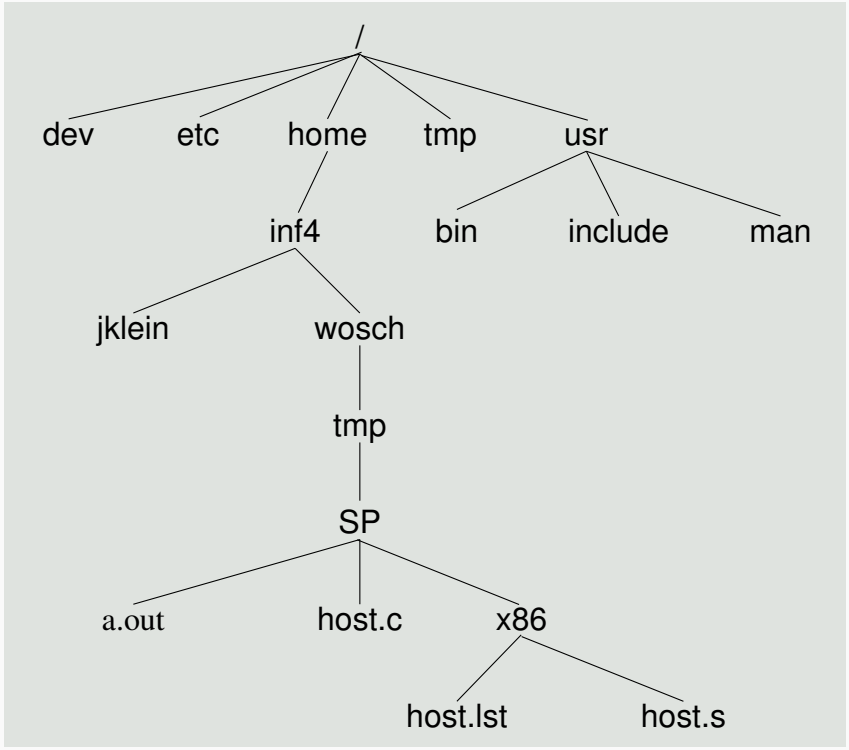
Jedes Programm (inkl. das Betriebssystem) kann eigenen Ressourcen Zeichenketten zuordnen und Prozessen diese bekanntgeben.

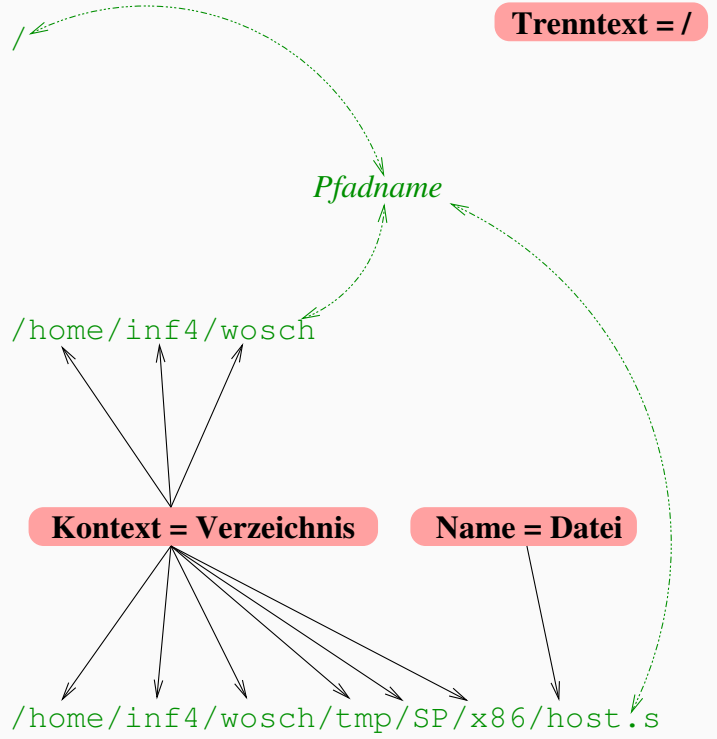
- den **Zeichenvorrat** für Namen und Komponenten eines Pfadnamens gibt die Namensverwaltung des Betriebssystems vor
  - allgemein die Menge der druckbaren Zeichen (ASCII) ohne Trennzeichen[4]
  - je nach Betriebssystem gibt es weitere Ausnahmen ($^{-} = \{", *, /, ?, |\}$)
- genau genommen werden Bezeichnungen aber aus **Ordnungszahlen** gebildet, die ein **Zeichensatz** erst in Zeichen umwandelt
  - z.B. UTF-8 (Linux), latin-1 (macOS) oder CP 437 (DOS)
  - nicht jede Ordnungszahl entspricht somit zwingend demselben Zeichen !
- ähnlich uneinheitlich ist die erlaubte Länge einer **Zeichenkette**, um Namen oder Pfadnamen zu formulieren
  - 1–255 Zeichen pro Name, bei UNIX auch pro Pfadnamenskomponente
  - bis zu $2^{15} - 1$ Zeichen pro Pfadname in Windows

[4] Nach ISO 9660: ausschließlich Großbuchstaben, Ziffern und Unterstrich.

Hierarchischer Namensraum I

Dateibaum (*file tree*)





- ein **Kontext** repräsentiert einen Namensraum flacher Struktur
  - darin muss Eindeutigkeit mit der Namenswahl selbst gewährtleistet sein
- im Gegensatz zu einem Namensraum hierarchischer Struktur (s. o.)
  - derselbe Name (`tmp`) kann in verschiedenen Kontexten definiert sein
  - durch den Pfadnamen wird sein jeweiliger Standort eindeutig gemacht

Hierarchischer Namensraum II

Dateiverzeichnis (*file directory*)

- fundamental für die Hierarchiebildung ist das **Verzeichnis**
  - es ordnet einen/mehrere Namen, auch Verzeichnisnamen, listenförmig
    - ist namentlich selbst in einem **Elterverzeichnis** (*parent directory*) gelistet
  - es gibt mehreren Namen ein gemeinsames Merkmal, denselben Kontext
    - bspw. Standort/Bezugspunkt innerhalb des Namensraums, Zugriffsrechte
  - es dient der Umsetzung von symbolischen in numerischen Adressen

Verzeichniseintrag (*directory entry*)

Speichert die Abbildung eines Namens auf eine Informationsstruktur.

- UNIX-artige Betriebssysteme bieten zudem vordefinierte Kontexte:
  - **Wurzelverzeichnis** (*root directory*) des Systems[5]
    - Einstiegspunkt in, aber auch „Steckverbinder“ für, den Namensraum
  - **Heimatverzeichnis** (*home directory*) eines authorisierten Benutzers
    - initiales Arbeitsverzeichnisses nach erfolgter Anmeldung (*login*)
  - **Arbeitsverzeichnis** (*working directory*) eines zugelassenen Prozesses
    - gegenwärtiger, relativer Standort des Prozesses im Namensraum

[5] Womit der Namensraum in einen bestehenden anderen Namensraum an einem Befestigungspunkt (*mount point*) gegebenenfalls eingebunden werden kann.

Informationsstruktur

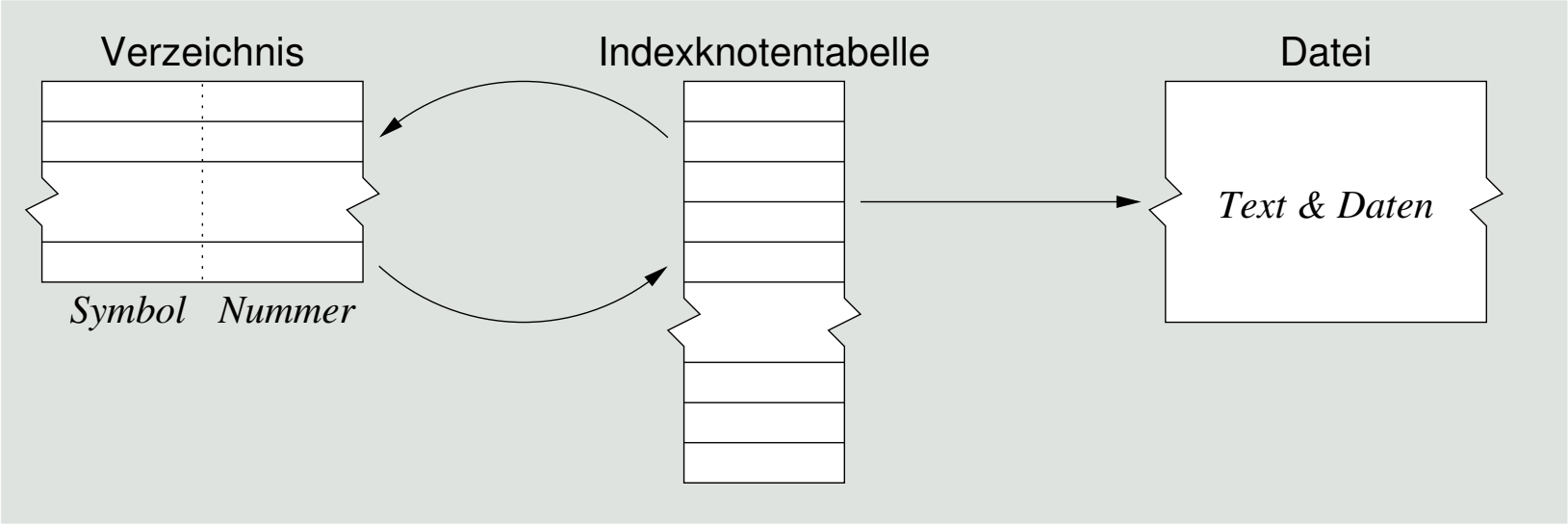
Indexknoten (*index node*, *inode*)

- aggregiert wesentliche **Attribute** eines benannten Gegenstands:
  - Eigentümer (*user ID*)
  - Gruppenzugehörigkeit (*group ID*)
  - Rechte (lesen, schreiben, ausführen: für Eigentümer, Gruppe und Welt)
  - Zeitstempel (letzter Zugriff, letzte Änderung (Typ, Zugriffsrechte))
  - Anzahl der Verweise („*hard link*“-Zähler)
  - Typ:
    - Verzeichnis
    - symbolische Verknüpfung (*symbolic link*)
    - Kommunikationskanal (*pipe*, *named pipe*)
    - Sockel (*socket*) zur Interprozesskommunikation
    - Gerätedatei $\rightsquigarrow$ zeichen-/blockorientiertes Gerät, Pseudogerät, Treiberklasse
    - reguläre Datei $\rightsquigarrow$ Größe in Bytes und Blocknummer(n) in der Ablage
- besitzt in einem Namensraum eine eindeutige **numerische Adresse**

Indexknotennummer (*inode number*)

Hat mit einer logischen Adresse gemeinsam, dass sie nur innerhalb ihres „Adressraums“ (d.h. Namensraums) eindeutig ist.

Datenstrukturen im Namensraum



- die **Indexknotentabelle** (*inode table*) ist ein statisches Feld (*array*) von Indexknoten und die zentrale Datenstruktur
  - ein Indexknoten ist **Deskriptor** insb. eines Verzeichnisses oder einer Datei
- das **Verzeichnis** (*directory*) ist eine **Abbildungstabelle**, es übersetzt symbolisch repräsentierte Namen in Indexknotennummern
  - eine von der Namensverwaltung des Betriebssystems definierte Datei
- die **Datei** (*file*) ist eine abgeschlossene Einheit zusammenhängender Daten beliebiger Repräsentation, Struktur und Bedeutung

Verzeichniseintrag I

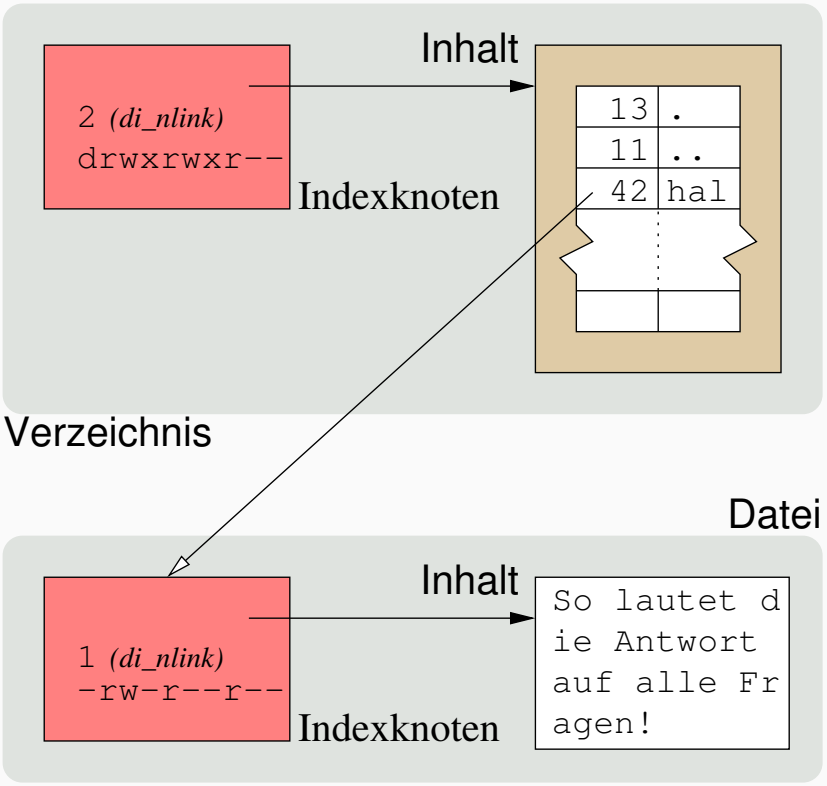
- die **feste Verknüpfung** (*hard link*) von einem Dateinamen mit einer Indexknotennummer (UNIX V7, `dir.h`):

```
1  typedef unsigned short ino_t;
2
3  #define DIRSIZ 14
4
5  struct direct {
6      ino_t d_ino;
7      char  d_name[DIRSIZ];
8  };
```

  - eine als **Wertepaar** gespeicherte **surjektive Abbildung**
  - mehrere Paare können zum selben Indexknoten (`d_ino`) zeigen
  - im selben Verzeichnis jedoch mit verschiedenen Namen (`d_name`)
  - in einem Indexknoten ist die Anzahl der auf ihn verweisenden Wertepaare desselben Namensraums gespeichert (*reference counter*)
  - alle Indexknoten eines Namensraums sind in einer **Indexknotentabelle** (*inode table*) im Namensraum daselbst gespeichert
    - `d_ino` ist der **Indexwert** eines Verzeichniseintrags für diese Tabelle
- Anlegen/Löschen erfordert **Schreibzugriffsrecht** auf das Verzeichnis
  - unabhängig von den Zugriffsrechten auf die referenzierte Datei

Verzeichniseintrag II

- ein Namenverzeichnis ist eine **spezielle Datei** der Namensverwaltung



- das selbst einen Namen hat, der einen Indexknoten bezeichnet
- über eine Verknüpfung erreichbar ist aus einem anderen Verzeichnis
- Namen getrennt von eventuellen Dateiinhalten speichert

*Verknüpfungen anlegen/löschen zu können, ist eine **Berechtigung**, die sich nur auf das Verzeichnis der betreffenden Verknüpfungen bezieht!*

- Selbstreferenz („*dot*“, 13) und Elterverzeichnis („*dot dot*“, 11) geben wenigstens zwei Verweise auf ein Verzeichnis
  - auch wenn das Verzeichnis selbst sonst keine weiteren Namen enthält

Bindung und Auflösung

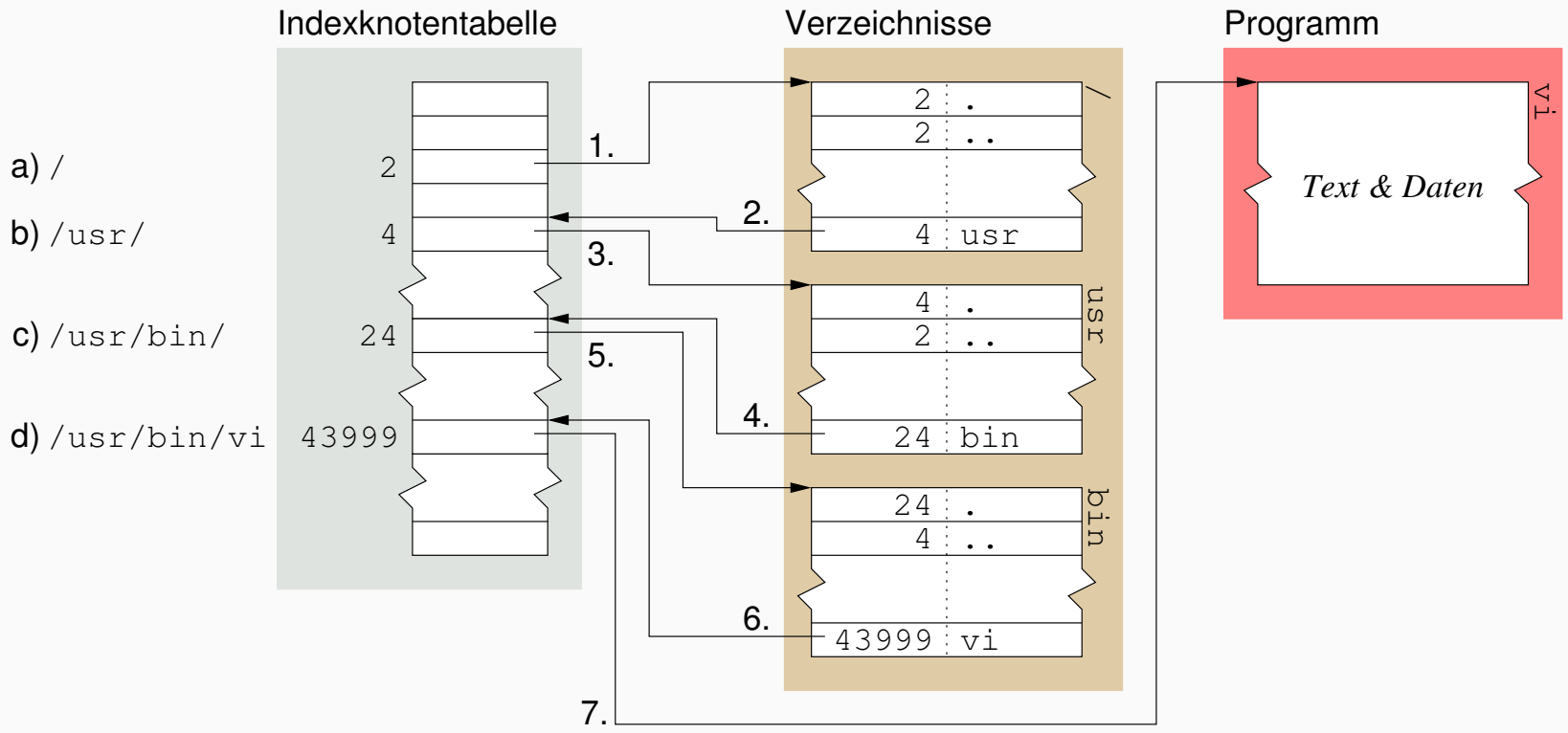
Abbildung $f : N_{symbolisch} \mapsto N_{numerisch}$

- **Namensbindung** (*name binding*) kommt zuerst, also die Abbildung der symbolischen Adresse in eine numerische Adresse
  - einen Pfadnamen mit einem Indexknoten assoziieren: `creat(2)`, `link(2)`
  - geschieht zum **Erzeugungszeitpunkt** eines Datei-/Verzeichnisnamens
    - diesen mit einem freien/belegten Indexknoten verknüpfen und
    - dann in ein Namensverzeichnis eintragen
- **Namensauflösung** (*name resolution*) kommt später, die Umsetzung der symbolischen Adresse in eine numerische Adresse
  - einen Indexknoten anhand eines Pfadnamens lokalisieren: `open(2)`
  - geschieht zum **Benutzungszeitpunkt** eines Datei-/Verzeichnisnamens
    - Verzeichnisse für jeden einzelnen Namen im Pfad durchsuchen und
    - schließlich den Dateinamen (Blatt) auffinden

Hinweis

*Der Indexknoten besitzt eine Adresse (a) in der Ablage und (b) im Arbeitsspeicher. Diese ist eine Art **virtuelle Adresse**, über die Inhalte von Dateien speicherabgebildet im virtuellen Adressraum indirekt zugänglich werden. Die Ladestrategie operiert vorausschauend.*

Auflösung am Beispiel von `/usr/bin/vi`



a) Wurzelverzeichnis des Namensraums öffnen
b) darin Namenseintrag `usr` suchen, ist ein Verzeichnis, öffnen
c) darin Namenseintrag `bin` suchen, ist ein Verzeichnis, öffnen
d) darin Namenseintrag `vi` suchen, ist ein Programm, öffnen

Symbolische Verknüpfung

symbolic link

- Indexknotennummern gelten in einem bestimmten **Namenraum**
  - der Zugriff auf einen anderen Namensraum ist darüber nicht möglich
    - sie teilen sich damit dieselbe Eigenschaft wie logische/virtuelle Adressen
    - sie sind ein **Tabellenindex**, wie der p- bzw. s-Anteil dieser Adressen
- der hierarchische Namensr. ist ein **gerichteter azyklischer Graph**
  - um den **Wurzelbaum** zu erhalten, scheiden feste Verknüpfungen aus
  - feste Verknüpfungen zu Verzeichnissen zerstören die azyklische Struktur
    - das Elterverzeichnis (. .) eines Verzeichnisses wäre dann uneindeutig
    - die Namensraumabsuche (*name space search*) könnte „endlos schleifen“
- Abbildung $f : N^d_{symbolisch} \mapsto N^z_{symbolisch}$ hat diese Merkmale nicht
  - feste Verknüpfungen sind ununterscheidbar, symbolische nicht: **Dateityp**
  - symbolische Verknüpfungen haben Indexknoten, feste nicht
    - sie gelten daher jeweils auch als:

      langsam weil indirekt gespeichert, in den Datenblöcken als reguläre Datei

      schnell weil direkt gespeichert, im Indexknoten selbst[6]
- Ursprung ist der **symbolische Name** (*symbolic name*) in Multics [2]
  - zur dynamischen Bindung von Namen an (besondere) E/A-Geräte

[6] Aber nur schnell bei einem Verweis auf einen Eintrag im selben Verzeichnis!

Gliederung

Symbolauflösung

Übersetzer

Adresse als Symbol

Abbildung $f : A_{symbolisch} \mapsto A_{numerisch}$

- am Anfang ist gemeinhin die **symbolische Adresse** in Programmen, die in einem mehrstufigen Verfahren aufzulösen ist

```
1  int main(int argc, char **argv) {
2      if (argc == 2) {
3          struct hostent *host = gethostbyname(argv[1]);
4          if (host != NULL) {
5              unsigned int i = 0;
6              printf("%s = ", host->h_name);
7              while (host->h_addr_list[i] != NULL)
8                  printf("%s ", inet_ntoa(*(struct in_addr*)(host->h_addr_list[i++])));
9              printf("\n");
10         }
11     }
12 }
```

Kompilierer
- verteilt Programmtext und -daten auf Programmsegmente
- generiert dazu Pseudobefehle, die der Assemblierer deutet

Assemblierer
- ordnet Programmsymbolen Werte und Attribute zu
- generiert Symbol- und Verlagerungstabellen für den Binder

Binder
- platziert das gebundene Programm im Adressraum
- produziert das Lademodul dazu für das Betriebssystem

- zusätzlich erfolgt die Generierung des Machinenkodes:
  - Kompilierung des Quellkodes in eine andere symbolische Darstellung
  - Assemblierung und Bindung in die benötigte numerische Auslegung

Programmkopf

- Programmsymbole auf **Binderabschnitte** (*linker sections*) verteilen

```
1           .file   "host.c"
2           .section        .rodata.str1.1,"aMS",@progbits,1
3   .LC0:
4           .string "%s = "
5   .LC1:
6           .string "%s "
7           .section        .text.startup,"ax",@progbits
8           .p2align 4,,15
9           .globl  main
10          .type   main, @function
11  main:
```

2
- beleg- und nur lesbares (`"aMS"`) **Datensegment** (2–6)
- Symbole `.LC0` und `.LC1` sind nur lokal definiert

7
- beleg- und ausführbares (`"ax"`) **Textsegment** (7–83, vgl. S. 34)

8
- Wert der Adresse des Abschnitts soll Vielfaches von 16 sein

9–10
- Symbol `main` ist global definiert, eine Funktion

- die Symbole werden bekannt gemacht und mit Attributen verknüpft
  - Text, Daten, Sichtbarkeit, Typ, Benutzungsart
- Adresswerte in Bezug auf den Prozessadressraum stehen noch aus
  - konkrete Werte für die Symbole und die einzelnen Maschinenbefehle

Programmfuß

- Abschluss der **Übersetzungseinheit** (*compilation unit*)

```
81  .LFE19:
82          .size   main, .-main
83          .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
84          .section        .note.GNU-stack,"",@progbits
```

82
- dem Symbol `main` die Länge des Programmtextes zuweisen

84
- das Programmteil benötigt kein ausführbares (`""`) **Stapelsegment**

- Ergebnis der Assemblierung ist das Binde- oder **Objektmodul**, das u.a. zwei (vom Assemblierer zusammengestellte) Tabellen enthält

Symboltabelle
- listet alle in dem Modul definierten Symbole und
- assoziiert jedes Symbol mit Werten und Attributen

Verlagerungstabelle
- listet alle „undefinierten Stellen“ in dem Modul
- Stellen, wo definierte Adresswerte noch fehlen

- jedes Text- und Datenelement besitzt eine **vorläufige Adresse**, die relativ zur Basisadresse 0 ausgelegt/-gerichtet ist
  - Stellen mit undefinierten Adressen korrigiert der **Binder** oder der **Lader**
    - d.h., es erfolgt die **Verlagerung** einer an dieser Stelle liegenden Referenz
  - Korrekturmaß ist die Ladeadresse des gebundenen Programms
    - die **Verlagerungskonstante**, gemäß Adressraummodell des Betriebssystems

Programmrumpf

- **Auflistung** (*listing*) der Übersetzungseinheit nach der Assemblierung (Ausschnitt des Quellprogramms von S. 32, dort Zeilen 3–6)

```
 1   26 0009 488B7E08    movq  8(%rsi), %rdi
 2   27 000d E8000000    call  gethostbyname
 3   27      00
 4   28 0012 4885C0      testq %rax, %rax
 5   29 0015 4889C5      movq  %rax, %rbp
 6   30 0018 745E        je    .L8
 7   31 001a 488B30      movq  (%rax), %rsi
 8   32 001d BF000000    movl  $.LC0, %edi
 9   32      00
10   33 0022 31C0        xorl  %eax, %eax
11   34 0024 31DB        xorl  %ebx, %ebx
12   35 0026 E8000000    call  printf
13   35      00
```

- typischerweise vier Bereiche:

1. Zeilennummer des Befehls, hier: 26–35
2. relative Adresse im Programm, hier: 0009–0026 (Hexadezimal)
3. numerischer Maschinenkode, hier: 32 Bit (4 Bytes) pro Zeile
4. Mnemonik und Operand(en)

- an folgenden relativen Adressen ist **Verlagerung** erforderlich:
  - **2/**000e — Adresse für `gethostbyname`, externe Referenz (`libc`)
  - **8/**001e — Adresse für `.LC0`, interne Referenz (vgl. S. 33)
  - **12/**0027 — Adresse für `printf`, externe Referenz (`libc`)
- die Programmausführung benötigt Korrektur an diesen Stellen
  - dort stehende **absolute Adressen** müssen zum Adressraummodell passen

Symbolauflösung

Binder

Maschinenprogramm

- derselbe Ausschnitt wie zuvor, jedoch dem **Lademodul** entnommen

```
1   0x00000000004003e9 <+9>:    mov    0x8(%rsi),%rdi
2   0x00000000004003ed <+13>:   callq  0x415e60 <gethostbyname>
3   0x00000000004003f2 <+18>:   test   %rax,%rax
4   0x00000000004003f5 <+21>:   mov    %rax,%rbp
5   0x00000000004003f8 <+24>:   je     0x400458 <main+120>
6   0x00000000004003fa <+26>:   mov    (%rax),%rsi
7   0x00000000004003fd <+29>:   mov    $0x48d1a4,%edi
8   0x0000000000400402 <+34>:   xor    %eax,%eax
9   0x0000000000400404 <+36>:   xor    %ebx,%ebx
10  0x0000000000400406 <+38>:   callq  0x401120 <printf>
```

  - vorher noch **unaufgelöste Referenzen** haben definierte Werte erhalten:

    `0x415e60` $\rightsquigarrow$ `gethostbyname`

    `0x48d1a4` $\rightsquigarrow$ `.LC0`

    `0x401120` $\rightsquigarrow$ `printf`

  - allen Text- und Datenelementen wurden **absolute Adressen** zugewiesen
    - der Programmausschnitt liegt im Adressbreich $[4003e9_{16}, 400406_{16}]$
    - bei Ausführung nimmt der Befehlszähler u.a. Werte aus diesem Bereich an

- die Adresswerte allein sagen nichts zur Art des Adressraums aus
  - es könnte ein realer, logischer oder virtueller Adressraum sein
  - Wissen über das **Adressraummodell** des Betriebssystems gibt Aufschluss

Symbolauflösung

Lader

Adressraummodell des Betriebssystems

x86_64 Linux

- **Adressraumbelegung** für dieses „Programm in Ausführung“

```
1   wosch@faui48e 111$ ./a.out faui40.cs.fau.de &
2   [1] 20857
3   faui40.informatik.uni-erlangen.de = 131.188.34.40
4   wosch@faui48e 112$ cat /proc/20857/maps
5   00400000-00401000 r-xp 00000000 00:2a 38020125          /home/inf4/wosch/tmp/a.out
6   00600000-00601000 rw-p 00000000 00:2a 38020125          /home/inf4/wosch/tmp/a.out
7   01ff9000-0201a000 rw-p 00000000 00:00 0                 [heap]
8   7f03d63ad000-7f03d6f82000 rw-p 00000000 00:00 0         shared libraries
9   7ffc79532000-7ffc79553000 rw-p 00000000 00:00 0         [stack]
10  7ffc795b8000-7ffc795b9000 r-xp 00000000 00:00 0         [vdso]
11  ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

  - **5** Textsegment, effektiv 4 KiB (eine Seite), insgesamt 2 MiB
  - **6** Datensegment, effektiv 4 KiB (eine Seite), insgesamt $\approx$ 26 MiB
  - **7** Haldenspeicher, vorgegeben 132 KiB, erweiterbar
  - **9** Stapelspeicher, vorgegeben 132 KiB, erweiterbar
  - die Größenangaben variieren mit dem Programm und dem Prozess !
- für einen Prozess adressier- aber nicht belegbar sind die Bereiche:
  - **8** Gemeinschaftsbibliotheken (*shared libraries*)
  - **10/11** Systemaufrufbeschleunigung (vgl. auch [4, S. 21–22])
- **Ladeadresse** — und somit **Verlagerungskonstante** — ist 0x400000
  - der Bereich $[0, 3\mathit{fffff}_{16}]$ ist nicht Teil des Prozessadressraums !

Gliederung

Resümee

…Namen sind nicht Schall und Rauch

- **symbolische Bezeichnungen** abstrahieren von konkreten Adressen
  - Programmtext- und -datenstellen, Datei-/Pfadnamen, …, URL
  - üblich ist die mehrstufige statische und dynamische Auflösung
- den Schwerpunkt bildete die **Namensauflösung** ↪ *dynamisch*
  - eine (reale, logische, virtuelle) Adresse ist Name einer Speicherstelle
    - je nach Abstraktionsebene der Haupt- oder Arbeitsspeicher
    - Seitenadressierung, Segmentierung, seitennummerierte Segmentierung
  - Datei-/Pfadnamen repräsentieren Adressen im Ablagesystem
    - der Namens-/Adressraum ist (nicht nur hier) hierarchisch organisiert
    - Orientierung am Dateisystem: Datei, Verzeichnis, Indexknoten, Verknüpfung
  - systemglobale, weltweite Eindeutigkeit liefert die Internetadresse
- Vorarbeit zu all dem leistet die **Symbolauflösung**: ↪ *statisch*

Kompilierer	▪ verteilt Programmtext und -daten auf Programmsegmente
Assemblierer	▪ ordnet Programmsymbolen Werte und Attribute zu
Binder	▪ platziert das gebundene Programm im Adressraum
Lader	▪ legt den „gefüllten“ Adressraum im Arbeitsspeicher ab

- das Zusammenspiel beider Verfahren ist typisch in Rechensystemen
  - nämlich Abbildungsfunktionen, die vor und zur Laufzeit greifen

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Denning, P. J.:
Virtual Memory.
In: *Computing Surveys* 2 (1970), Sept., Nr. 3, S. 153–189

[2] Feiertag, R. J. ; Organick, E. I.:
The Multics Input/Output System.
In: *Proceedings of the Third ACM Symposium on Operating System Principles (SOSP 1971), October 18–20, 1971, Palo Alto, California, USA*, ACM, 1971, S. 35–41

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Betriebssystemmaschine.
In: [6], Kapitel 5.3

[4] Kleinöder, J. ; Schröder-Preikschat, W. :
Maschinenprogramme.
In: [6], Kapitel 5.2

Literaturverzeichnis (2)

[5] Kleinöder, J. ; Schröder-Preikschat, W. :
Speicher.
In: [6], Kapitel 6.2

[6] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)

[7] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: [6], Kapitel 5.1

[8] Organick, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0–262–15012–3

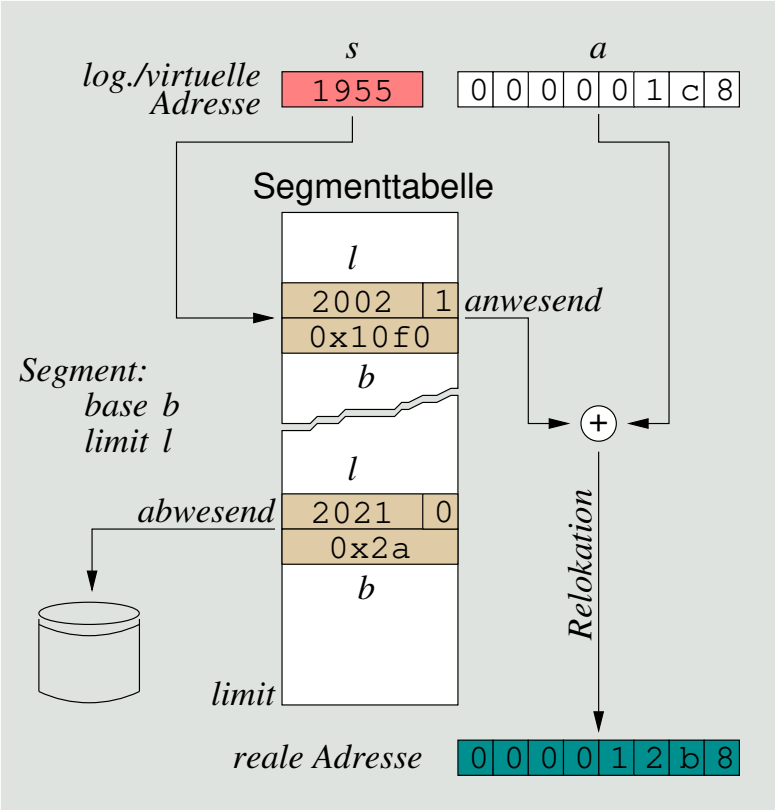
Anhang

Speicheradressen

Segmentierung

segmentation

- auf Basis einer einstufigen **Segmenttabelle** als dynamisches Feld:



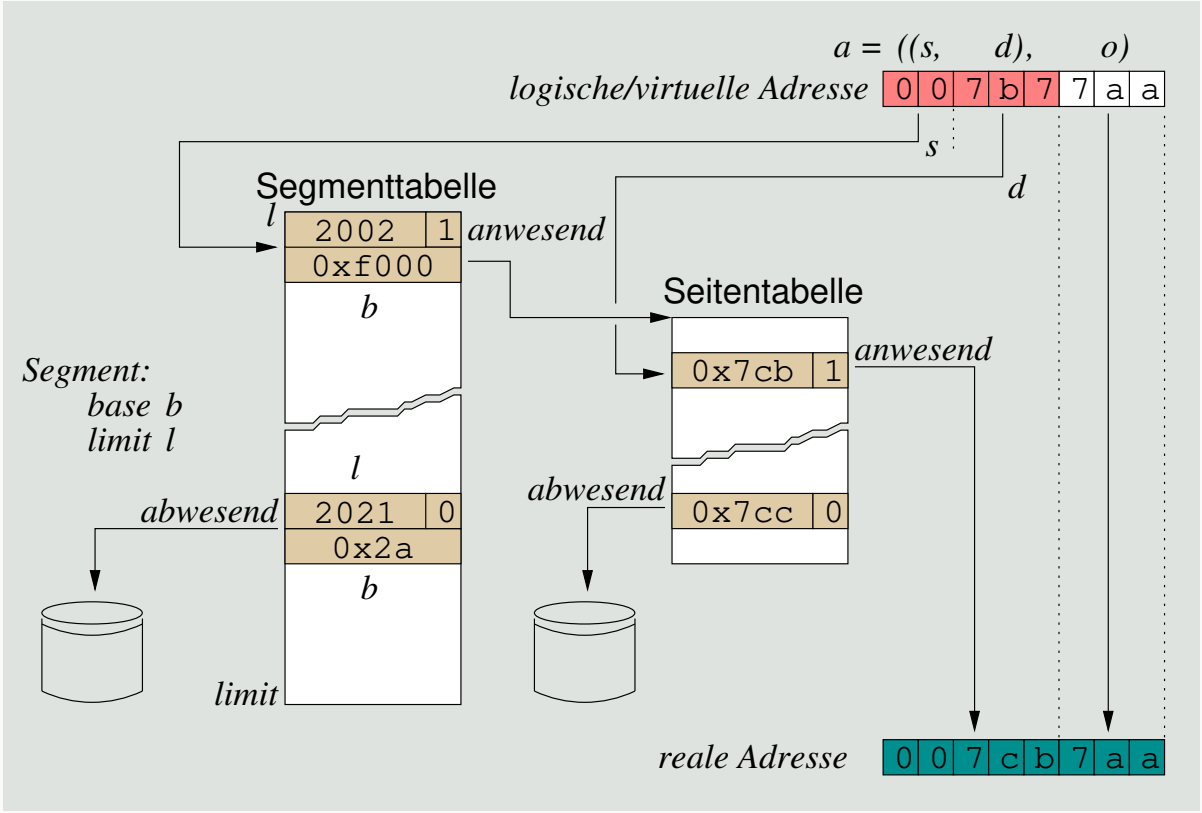
  - s ist **Indexwert**, der gültig ist im Bereich $[0, S_{limit}]$
  - sei $S = 2^m$ max. Segmentanzahl
  - dann gilt $0 < S_{limit} \leq S - 1$
  - ein möglicher **Indexfehler** muss erkannt werden
  - Grenzwertprüfung auf Basis eines *limit*-Registers (MMU) ist üblich
  - die Tabelle wird (norm.) nicht mit ungültigen Einträgen aufgefüllt
  - S hängt ab von der MMU und dem Betriebssystem

- ein **Segmentfehler** (*seg. fault*) bedeutet dann verschiedenerlei:
  - gültig falls $0 \leq s \leq S_{limit}$ und $a \leq l$, mit $l = Segmenttabelle[s].limit$, dann ist s abwesend (*swap-in*) oder ungenutzt (dynamisches Binden)
  - sonst ungültig: Segment s oder Adresse a ist für den Prozess undefiniert

Segmentierte Seitenadressierung

segmented paging

- auf Basis einer einstufigen **Segmenttabelle** als dynamisches Feld:



  - die Seitentabelle:
    - ist segmentiert, dynamisches Feld
    - enthält nur gültige Einträge
    - kann ausgelagert sein (*swapping*)
  - Adressraum:
    - eindimensional
    - *a* ist eine lineare Adresse
    - nicht wirklich segmentiert !

- mögliche Ausnahme (*exception*, vgl. [7, 3]) bei der Adressierung:

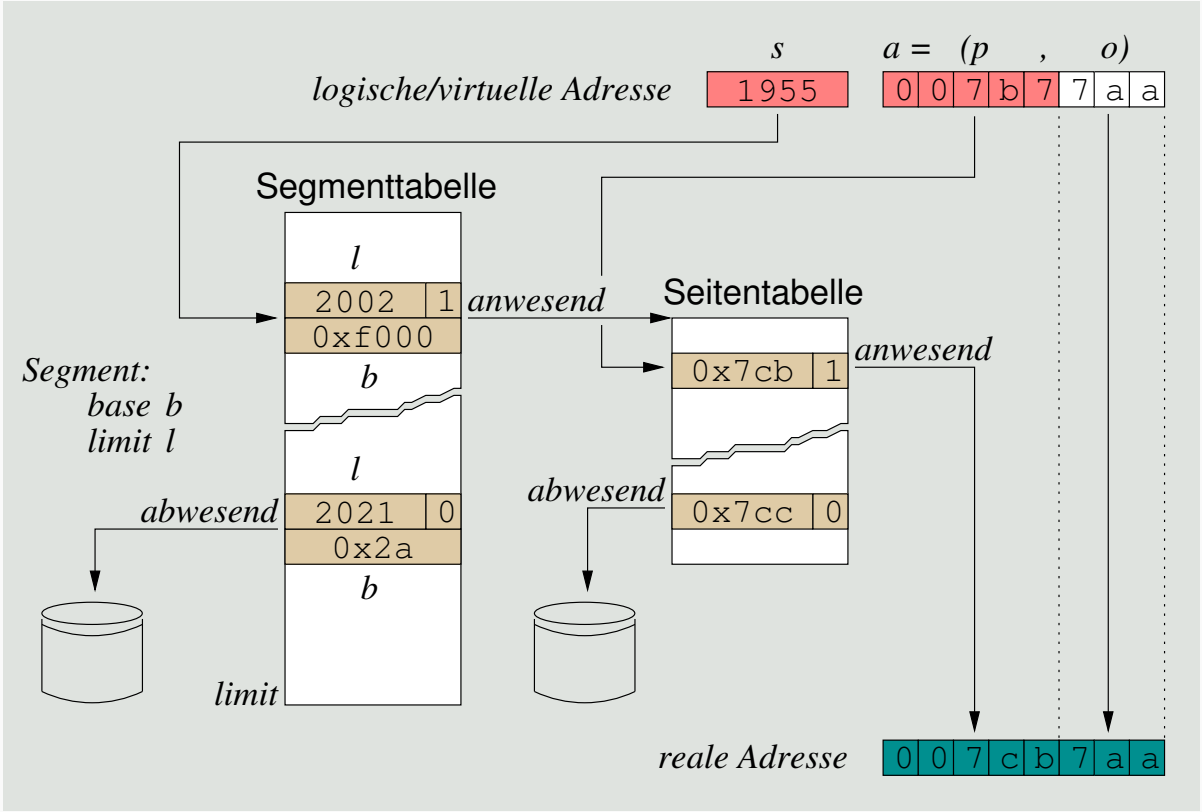
Segmentfehler
- falls $s > S_{limit}$ und $d > l$ (vgl. S. 46) . . . . . Abbruch

Seitenfehler
- falls Seite d abwesend (vgl. S. 16) . Wiederaufnahme

Seitennummerierte Segmentierung I

paged segmentation

- à la GE645/Multics, einstufige **Segmenttabelle**, dynamisches Feld:



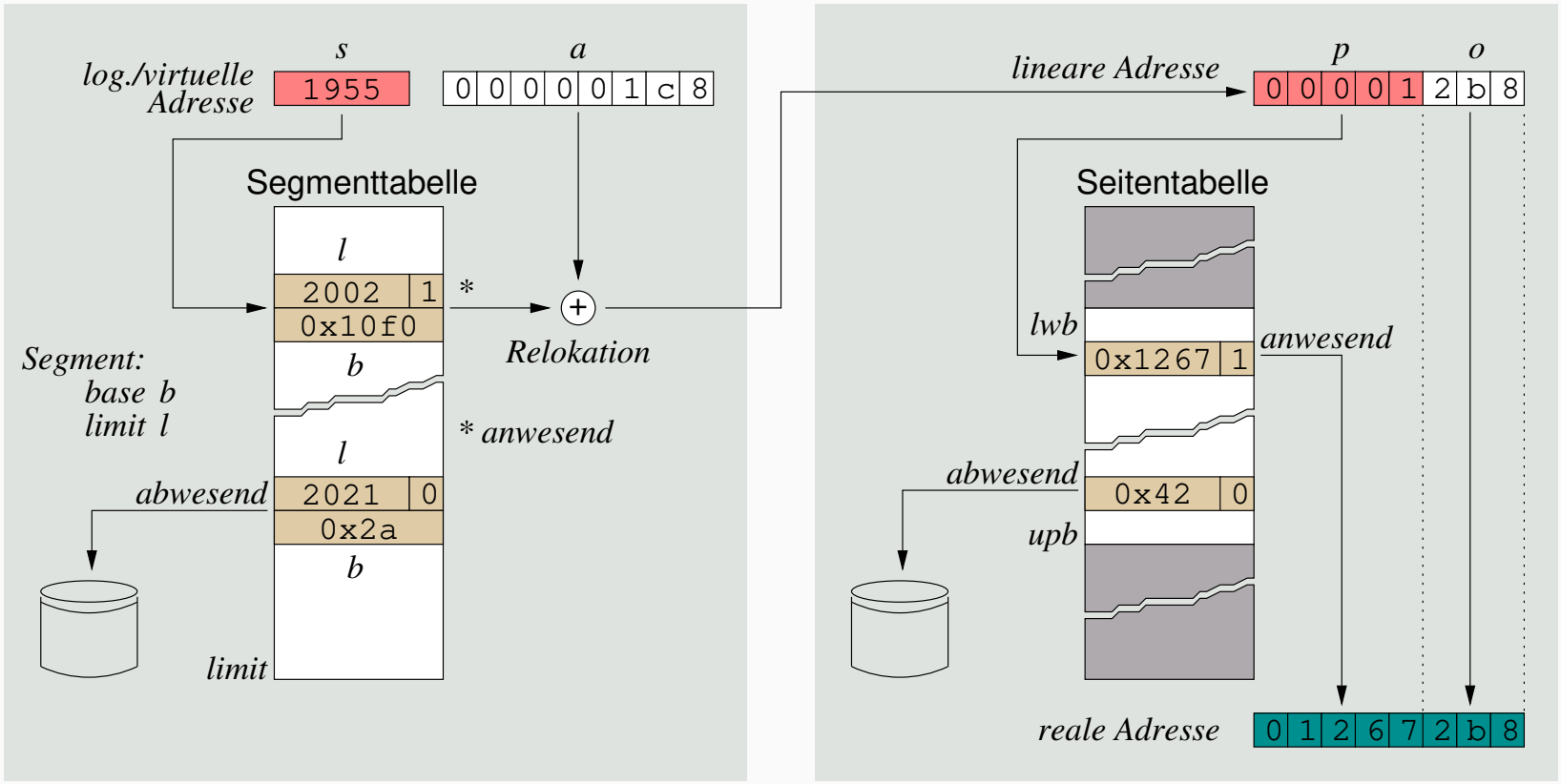
- die Seitentabelle:
  - ist segmentiert, dynamisches Feld
  - enthält nur gültige Einträge
  - kann ausgelagert sein (*swapping*)
- Adressraum:
  - zweidimensional
  - *a* hat zwei Komponenten
  - stark segmentiert

- Ausnahmen wie bei segmentierter Seitenadressierung (vgl. S. 47)
  - hohe Segmentanzahl fördert **dynamisches Binden** (Text, Daten) und **Mitbenutzung** (*sharing*) von Programm- und Datenstrukturen
  - Seiten behalten ihre Bedeutung: Entitäten des virtuellen Speichers

Seitennummerierte Segmentierung II

paged segmentation

- à la IA-32, **Reihenschaltung** von Adressumsetzungseinheiten:



Segmentadressierungseinheit

Seitenadressierungseinheit

- Ausnahmen wie bei segmentierter Seitenadressierung (vgl. S. 47)
  - ein-/mehrstufig ausgelegte (auslagerbare) Seitentabelle, statisches Feld
  - unterstützt die **partielle Mitbenutzung** eines einzelnen Seitenrahmens

Seitennummerierte Segmentierung: Unterschiede

- segmentierte Seitentabelle
  - der Segmentdeskriptor listet **Seitendeskriptoren**
  - er adressiert damit indirekt ein dynamisches Seitenfeld
  - alle Seiten darin sind gültig für den betreffenden Prozess
  - folglich auch alle Bytes eines jeweiligen Seitenrahmens
  - Fußbereiche von Seitenrahmen können jedoch brach liegen

  ↪ Seitenverschnitt unvermeidbar

- Reihenschaltung
  - der Segmentdeskriptor listet **Speicherworte**
  - er adressiert damit direkt ein dynamisches Bytefeld
  - alle Bytes darin sind gültig für den betreffenden Prozess
  - dies unabhängig davon, welche Seitenrahmen sie aufnehmen
  - folglich können Seitenrahmen partiell mitbenutzt werden

  ↪ Verschnitt darin vermeidbar

- Verschnitt im Seitenrahmen (bei Reihenschaltung) zu vermeiden, ist für gewöhnlich aufwendig und verstärkt zudem **Interferenz**
  - zwei Seiten ggf. zweier Segmente haben Anteile desselben Seitenrahmens
  - Ersetzung des Inhalts dieses Seitenrahmens kann zwei Prozesse „stören“
  - auch sind Löcher dann kein Vielfaches von Seitenrahmen mehr

    ↪ Verschnitt im Hauptspeicher unvermeidbar: **externe Fragmentierung**

Partielle Mitbenutzung von Seitenrahmen

- Platzierung von Segmentkopf und -fuß in denselben Seitenrahmen F, wobei Kopf- plus Fußlänge die Seitenrahmenlänge (4 KiB) ergibt
  - **erste Seite**: $P^{y}_{[0,4053]}$, Kopf in Segment S^{y}, liegt auf $F^{z}_{[42,4095]}$
  - **letzte Seite**: $P^{x}_{[0,41]}$, Fuß in Segment S^{x}, liegt auf $F^{z}_{[0,41]}$
  - dabei können S^{x} und S^{y} demselben Adressraum (eines Prozesses) oder verschiedenen Adressräumen (zweier Prozesse) angehören
  - falls derselbe Adressraum, kann sogar $S^{x} = S^{y}$ gelten, d.h., Kopf und Fuß desselben Segments liegen im selben Seitenrahmen
- zu beachten ist, dass sich die **lineare Adresse** des Segmentkopfes auf einen gekachelten logischen/virtuellen Adressraum bezieht
  - diese Adresse ist die im Segmentdeskriptor stehende Segmentbasis und sie muss überhaupt nicht seitenausgerichtet (*page aligned*) sein
  - innerhalb der ersten Seite in diesem Adressraum kann die Segmentbasis um einen Wert $v \in [0, \mathit{sizeof}(P) - 1]$ verschoben sein
  - dieser Wert v entspräche dann der Größe eines zugeteilten Speicherstücks (Segmentfuß) am Anfang eines Seitenrahmens
  - der Rest von $\mathit{sizeof}(P) - v$ Bytes in dem Seitenrahmen entspräche einem Speicherstück, dass einem Segmentkopf zugeteilt werden könnte

Seitenadressierung mit Segmentierung

…hat Mehrkosten

- **Synergie** der positiven Merkmale beider Adressumsetzungsarten
  - einfache Platzierungsstrategie, da die Speicherzuteilung kachelorientiert und damit immer in Einheiten gleicher Größe geschieht (vgl. [5, S. 18])
  - mehrstufige Seitentabellen fallen weg, da alle Tabellen Segmente und so jeweils in ihrer wirklichen Seitenanzahl beschränkt sind
  - bessere Trennung von Belangen, da Segmente und Seiten bzw. Kacheln verschiedenen Zielen dienen

    Segment – Abbildung und Erfassung von **Programmstrukturen**

    Seite – Optimierung von **Systemfunktionen** der Speicherverwaltung
- Segmentierung unterstützt insbesondere **dynamisches Binden**
  - die „Bindlinge“ sind symbolisch bezeichnete, **physische Segmente**
  - d.h., Programmstrukturen, Adressräume (Seitentabellen), …, Dateien
- ein Seg. dagegen als „Seitenfeld“ zu begreifen, ist etwas anderes
  - also Seiten zu Text-, Daten- oder Stapelsegmenten zusammenstellen[7]
  - Programmstrukturen lassen sich damit im System nicht wirklich abbilden

    – vom Verwaltungsaufwand mehrstufiger Seitentabellen einmal abgesehen

[7] So, wie es von UNIX-ähnlichen Betriebssystemen (inkl. Linux) bekannt und überhaupt nach Multics [8] eben nur noch gang und gäbe ist.

Anhang

Pfadnamen

Verknüpfungen überarbeitet

Gegenüberstellung

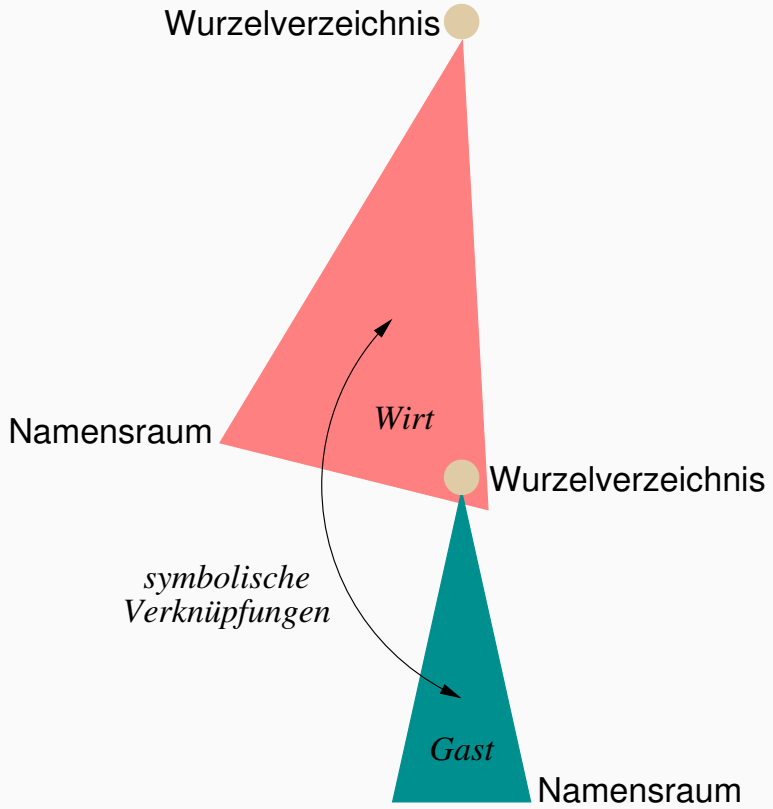
- feste Verknüpfung
  - nicht zu Verzeichnissen oder Dateien anderer Dateisysteme
  - überdauert die Umplatzierung einer Datei
  - bleibt bestehen, nur solange es noch Referenzen gibt
  - hat keinen eigenen Indexknoten
- symbolische Verknüpfung
  - auch zu Verzeichnissen und Dateien anderer Dateisysteme
  - ungültig nach Umplatzierung einer Datei
  - bleibt bestehen, auch wenn es keine Referenzen gibt
  - hat einen eigenen Indexknoten

„Nicht alles, was glänzt, ist Gold“ (Shakespeare, 1600)

```
wosch@lorien 1$ mkdir -p Laptop/faui43w; cd Laptop; ln -s faui43w lorien; ls -l
total 8
drwxr-xr-x  2 wosch  wosch  68 29 Apr 13:01 faui43w
lrwxr-xr-x  1 wosch  wosch   7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 2$ cd lorien
wosch@lorien 3$ cd ..; rmdir faui43w; cd lorien
-bash: cd: lorien: No such file or directory
wosch@lorien 4$ ls -l
total 8
lrwxr-xr-x  1 wosch  wosch   7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 5$ mkdir faui43w; cd lorien
wosch@lorien 6$ ln -s Fata\ Morgana SP1
```

Namensraumverbindung

- Namensräume können an einem **Befestigungspunkt** (*mount point*) miteinander verbunden werden
  - ohne damit jedoch eine Erweiterung des Namensraums vorzunehmen
- der Punkt ist ein **Verzeichnis** im Wirtsnamensraum
  - Einhängen (*mount*) blendet den Inhalt des Wirtsverzeichnisses aus
  - der Wurzelverzeichnisinhalt des Gastnamensraums erscheint
  - Aushängen (*unmount*) macht den alten Verzeichnisinhalt sichtbar
- ein **Pfadname** kann dann Wirts- und Gastnamensraum abdecken
  - einerseits streng hierarchietreu, von oben nach unten (*top down*)
  - andererseits quer verweisend, durch **symbolische Verknüpfung**



Anhang

Internetadressen

Adresse als Nummer

Definition (Nummerung (DIN 6763))

Bilden, Erteilen, Verwalten und Anwenden von Nummern.

- die **Eindeutigkeit** der Speicher- und Standortadressen ist begrenzt
  - Indexknotennummern durch den **Namensraum** ihres Dateisystems
  - reale, logische und virtuelle Adressen durch ihren (Prozess-) **Adressraum**
  - Prozesskennungen durch den **Nummernraum** ihres Rechensystem
- die **Internetprotokolladresse** (IP-Adresse) ist weltweit eindeutig

  IPv4 ▪ 32 Bit: vier Blöcke zu jeweils drei Dezimalstellen (8 Bit)

  IPv6 ▪ 128 Bit: acht Blöcke zu jeweils vier Hexadezimalstellen (16 Bit)
  - vom ARP (*address resolution protocol*) aufgelöst und umgewandelt in die **Netzwerkadapteradresse** (MAC, *media access control*)
  - rechensystemlokal wird das Adressenpaar in der ARP-Tabelle hinterlegt
- verallgemeinerte Form ist der/die **URL** (*universal resource locator*)
  - neben Adressinformationen ist zusätzlich die **Zugriffsmethode** enthalten, die durch `://` von den schemaspezifischen Angaben getrennt ist
  - die Internetadresse identifiziert dabei den **Wirt** (*host*) einer Webanfrage

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VII.1 Betriebsarten: Stapelverarbeitung

4. Juli 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

Gliederung

Lehrstoff

- Ziel ist es, „zwei Fliegen mit einer Klappe zu schlagen“, nämlich:
  i einen Einblick in **Betriebssystemgeschichte** zu geben und
  ii damit gleichfalls **Betriebsarten** von Rechensystemen zu erklären
- im Vordergrund stehen die Entwicklungsstufen im **Stapelbetrieb**
  - Adressraumschutz durch Abteilung, Eingrenzung und Segmentierung
  - Durchsatz-/Leistungssteigerung durch abgesetzten Betrieb, überlappte und abgesetzte Ein-/Ausgabe und Simultanbetrieb
  - Speicherminimierung durch Programmzerlegung und Überlagerungen

Hinweis

*Viele dieser Techniken — wenn nicht sogar alle — sind auch heute noch in einem **Universalbetriebssystem** auffindbar.*

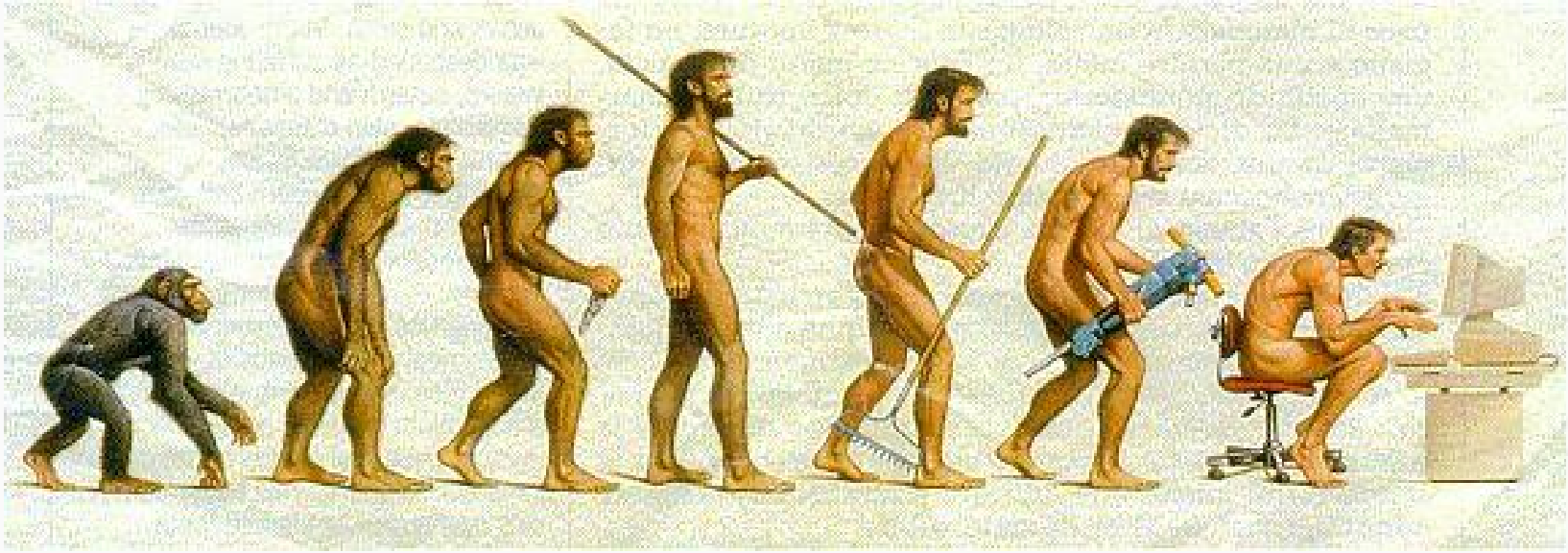
- kennzeichnend ist, die Programme **interaktionslos** auszuführen
  - hierzu ist eine vollständige Auftragsbeschreibung erforderlich
  - die in einer speziellen „Skriptsprache“ ausformuliert wird
  - um das fertige „Skript“ von einem Interpretierer verarbeiten zu lassen
  - der anfangs als Monitor und später als Betriebssystem in Erscheinung tritt

Einführung

Präludium

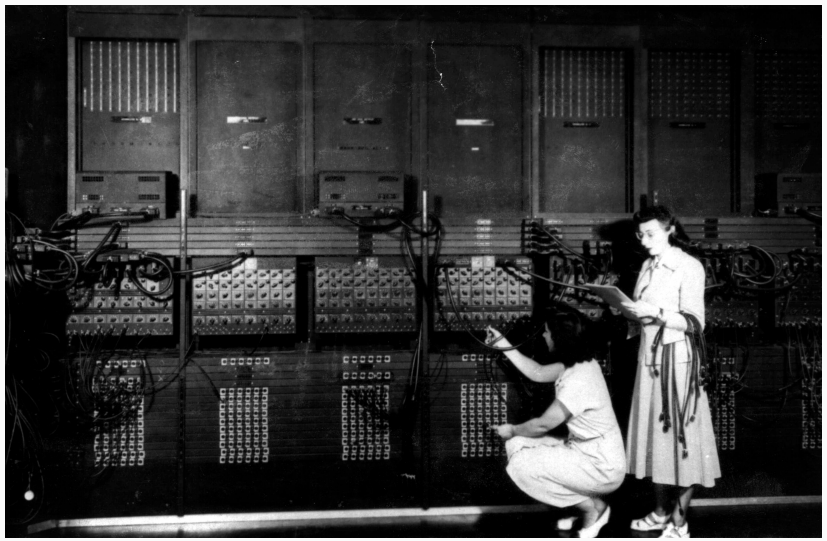
Betriebssystemevolution

Wenn wir nicht absichtlich unsere Augen verschließen, so können wir nach unseren jetzigen Kenntnissen annähernd unsere Abstammung erkennen, und dürfen uns derselben nicht schämen.
(Charles Darwin)



Am Anfang war das Feuer…

- ENIAC (*electronic numerical integrator and computer*), 1945 [18]



> A small amount of time is required in preparing the ENIAC for a problem by such steps as setting program switches, putting numbers into the function table memory by setting its switches, and establishing connections between units of the ENIAC for the communication of programming and numerical information. (Pressemitteilung, DoD, 1946)

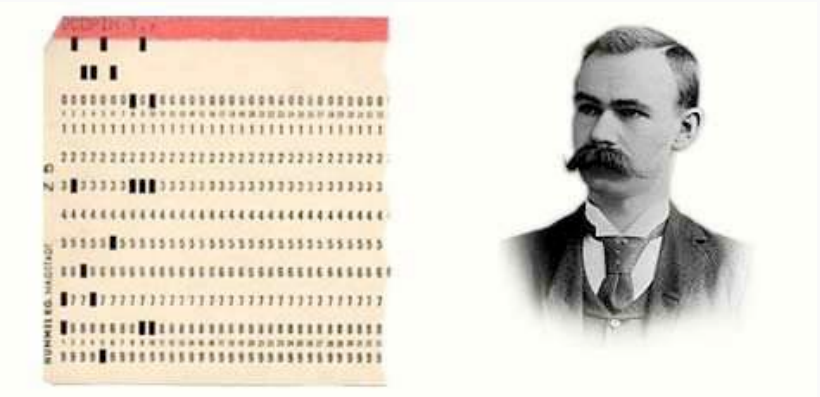
- elektronischer Allzweckrechner von 30 Tonnen Gewicht
- 15 $m \times 3\,m \times 5\,m$ große Zentraleinheit, 30 m lange Frontplatte
- für seinen Betrieb waren 18 000 Röhren zuständig
- die elektrische Anschlussleistung belief sich auf etwa 174 000 Watt

Gliederung

Einprogrammbetrieb

Manueller Rechnerbetrieb

Lochkartenverarbeitung



Hermann Holerith (1860–1929), Begründer der maschinellen Datenverarbeitung.

IBM ließ sich 1928 das Format patentieren: 80 Spalten, 12 Zeilen und rechteckige Löcher an den Schnittpunkten.

- **Ziffernlochkarte** (*numeric punch card*)
  - Datenaufzeichnung durch Lochung (Loch $\mapsto$ 1, kein Loch $\mapsto$ 0)
    - Dezimalzahlen werden mit einer Lochung dargestellt
    - Buchstaben und Sonderzeichen mit zwei oder drei
    - negative Vorzeichen ggf. durch eine Lochung in Zeile 11
  - manuell ca. 15 000 Zeichen/h, maschinell 4 000–10 000 Karten/h

Offene Stelle, an der die Substanz nicht mehr vorhanden ist — aber Information!

Alternativ kamen auch **Lochstreifen** (*punched [paper] tape*) zum Einsatz, mit ähnlichen Leistungsmerkmalen.

Manueller Betrieb des Rechners

- vollständige Kontrolle beim Programmier- oder Bedienpersonal[1]

1. Programme mit **Lochkartenstanzer** (*card puncher*) ablochen
2. Übersetzerkarten in den **Lochkartenleser** (*card reader*) geben
3. Lochkartenleser durch Knopfdruck starten
4. Programmkarten (aus Pt. 1 zur Übersetzung) in den Kartenleser einlegen
5. Eingabekarten in den Kartenleser einlegen
6. Leere Lochkarten für die Ausgabe in den Kartenstanzer einlegen
7. Ausgabekarten in den Kartenleser des Druckers einlegen
8. Ergebnisse der Programmausführung vom Drucker abholen

- **Problem**:
  - Urlader (z.B. Lochkartenleseprogramm) in den Rechner einspeisen

Programmieren bedeutet nicht Ausprobieren!

Wenigstens einen großen Vorteil hätte diese Betriebsart auch heute noch: man überlegt sich Programme sorgfältig und interpretiert (d.h., deutet) sie selbst, bevor ihre Eingabe in den Rechner erfolgt.

[1] „*Operator*“: ein Berufsbild, dass es in der Form heute nicht mehr gibt.

Lochkartenleseprogramm

- Rechnersystem durch **Ureingabe** (*bootstrap*) laden

1. Bitmuster einer Speicherwortadresse über Schalter einstellen
   - die Adresse der nächsten zu beschreibenden Stelle im Hauptspeicher
2. den eingestellten Adresswert in den PC der CPU laden
3. Bitmuster für den Speicherwortinhalt über Schalter einstellen
   - ein Befehl, ein Direktwert oder eine Operandenadresse des Programms
4. den eingestellten Datenwert an die adressierte Stelle laden

- **Problem**:
  - Bedienung, Mensch, Permanenz

Programmiertes Urladen als Firmware

Heute liegen Urlader nicht-flüchtig im Speicher (ROM/EPROM bzw. *flash*). Sie starten automatisch, wenn der Rechner angeschaltet und hochgefahren oder, während des Betriebs, manuell oder automatisch zum Neustart (*restart*) bzw. Zurücksetzen (*reset*) gezwungen wird.

Einprogrammbetrieb

Automatisierter Rechnerbetrieb

Automatisierter Betrieb des Rechners

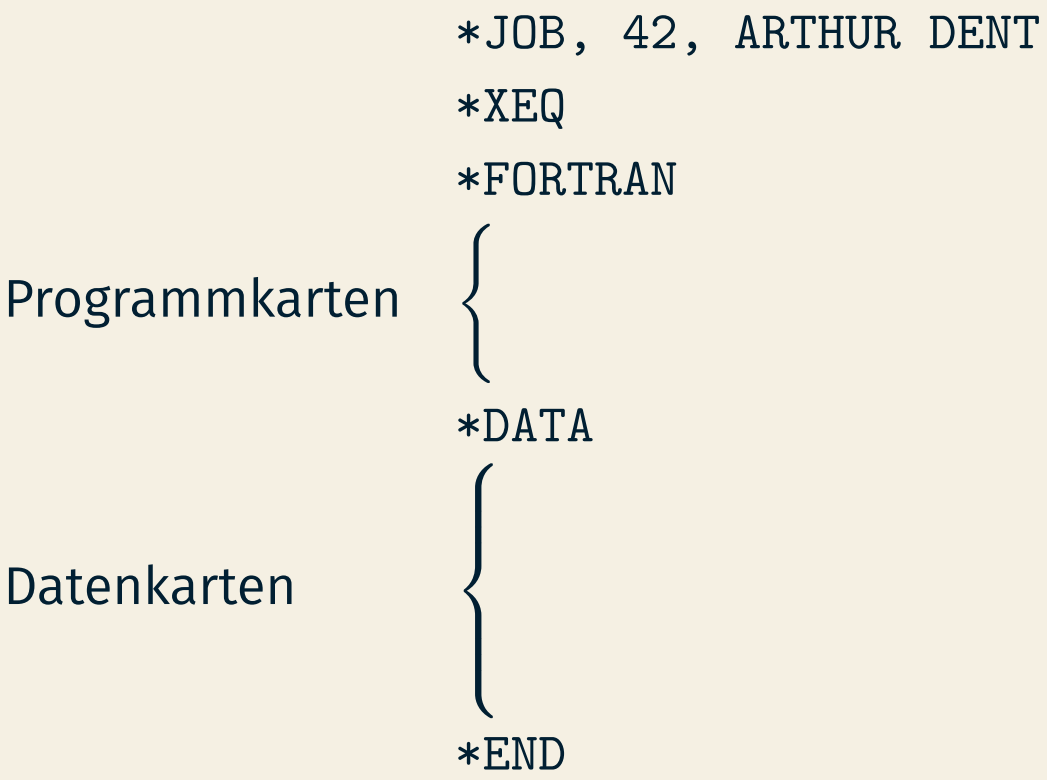
- **Dienstprogramme** sind abrufbereit im Rechnersystem gespeichert:
  - Systembibliothek, „Datenbank“ (Dateiverwaltung)
- **Anwendungsprogramme** fordern Dienstprogramme explizit an:
  - spezielle **Steuerkarten** sind dem Lochkartenstapel hinzugefügt
    - Systemkommandos, die auf eigenen Lochkarten kodiert sind
    - Anforderungen betreffen auch Betriebsmittel (z.B. Speicher, Drucker)
  - der erweiterte Lochkartenstapel bildet einen **Auftrag** (*job*)
    - an einen **Kommandointerpretierer** (*command interpreter*)
    - formuliert als **Auftragssteuersprache** (*job control language*, JCL)
  - die Programmausführung erfolgt ohne weitere Interaktion
- **Problem**:
  - vollständige Auftragsbeschreibung (inkl. Betriebsmittelbedarf)

Bestimmte Tätigkeiten sehr sicher und schnell durchführen

Eignet sich (nach wie vor) zur Bewältigung von „Routineaufgaben“.

Auftragssteuersprache

- FORTRAN Monitoring System, FMS [9], um 1957
  - wird zuweilen auch als „erstes Betriebssystem“ bezeichnet



Residentes Steuerprogramm

resident monitor

- **hauptspeicherresidente Systemsoftware**, zur Auftragssteuerung:
  - Kommandointerpreter, Lochkartenleseprogramm, E/A-Prozeduren
  - Systemprogramme, die ein „embryonales Betriebssystem“ bilden
- **Solitär**: einzelstehende und getrennt übersetzte Programmeinheit, die verschiedenartig vom Anwendungsprogramm entkoppelt ist
  - Einsprungtabelle (*jump table*)
  - partielle Interpretation von Monitoraufrufen
  - getrennte/partitionierte reale Adressräume (Schutzgatter)
  - Arbeitsmodi (System-/Benutzermodus, privilegiert/unprivilegiert)
- **Problem**:
  - Arbeitsgeschwindigkeit der Peripherie, sequentielle Ein-/Ausgabe

Systemsoftware als Firmware

Heute ist diese Funktionseinheit noch in dem „*basic input/output system*“ (BIOS) präsent, mit dem nahezu jeder Rechner ausgestattet ist und das nach wie vor grundlegende Aufgaben, die nicht nur zum Urladezeitpunkt anfallen, übernimmt.

Einprogrammbetrieb

Schutzvorkehrungen

Adressraumschutz durch Abgrenzung *fencing*

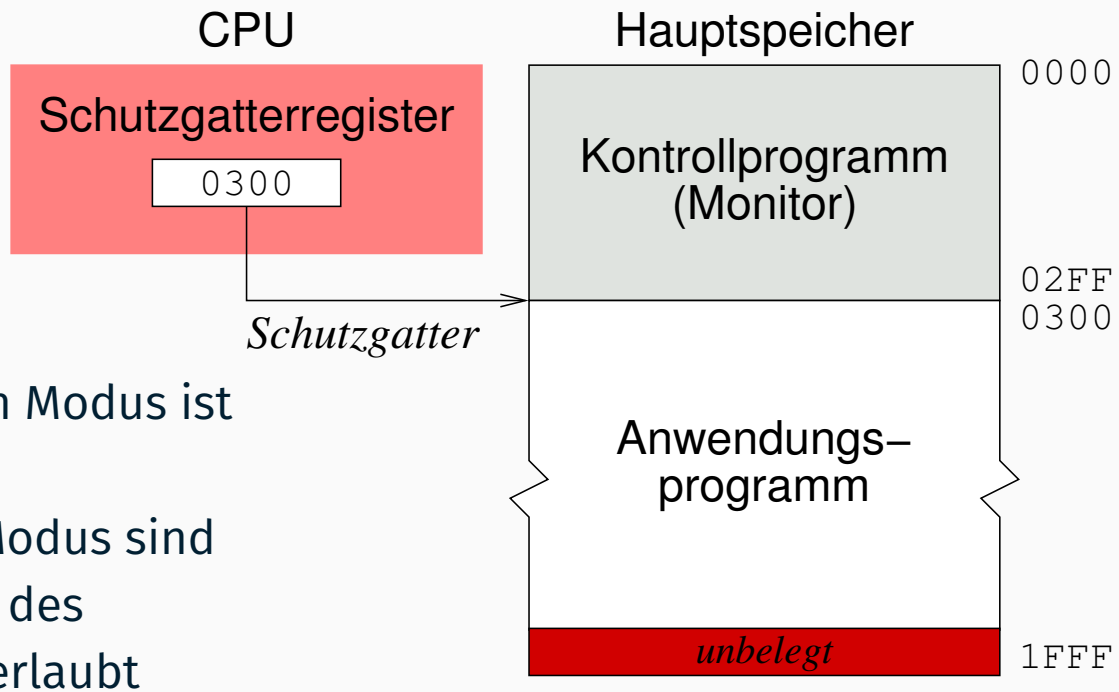
- ein **Schutzgatter** (*fence*) trennt den vom residenten Steuerprogramm und vom Programm belegten Hauptspeicher
  - Anwendungsprogramme werden komplett, d.h. statisch gebunden
    - das Schutzgatter entspricht einer unveränderlichen, realen Speicheradresse
    - Verlagerungskonstante beim Binden, Ladeadresse für die Programme
  - ggf. ist gewisse Flexibilität durch ein **Schutzgatterregister** gegeben
    - veränderliche, reale Speicheradresse; programmierbarer Wert
    - Verlagerungsvariable beim Binden, (zur Basis 0) relative Programmadressen
    - ein **verschiebender Lader** platziert das Programm im Hauptspeicher
- innerhalb der (Anwendungs-) **Partition**, die am Schutzgatter startet oder endet, ist dynamischer Speicher begrenzt zuteilbar
  - d.h., in der oberen oder unteren Hälfte des Hauptspeichers
  - der Monitor selbst betreibt jedoch nur **statische Speicherverwaltung**
- **Problem**:
  - übergroße und statisch gebundene Anwendungsprogramme

Isolation permanent benötigter Programme

Geschützt wird der Monitor, nicht jedoch das Anwendungsprogramm.

Schutzgatterregister I

fence register

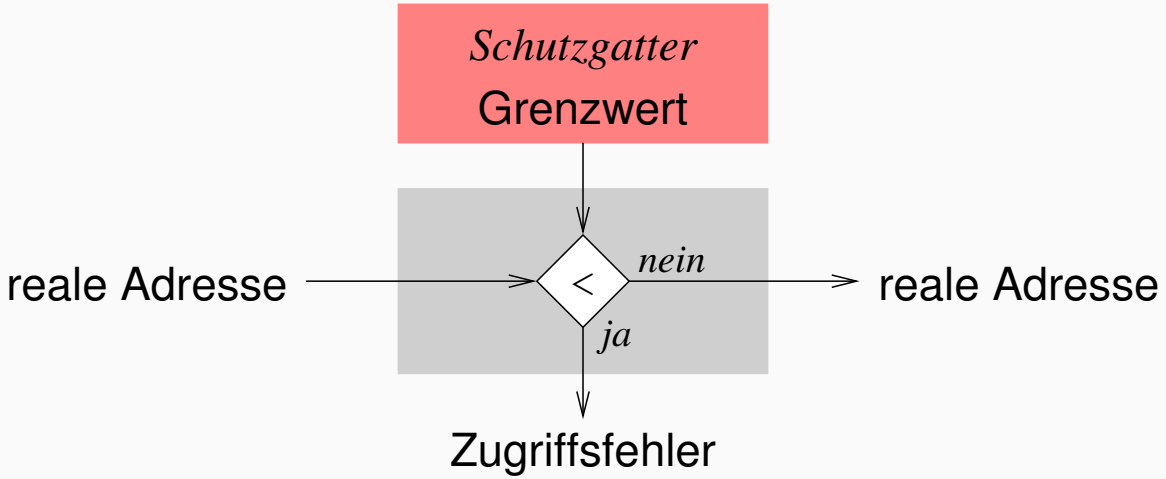


- **Arbeitsmodi**
  - nur im unprivilegierten Modus ist das Schutzgatter aktiv
  - nur im privilegierten Modus sind Änderungen am Inhalt des Schutzgatterregisters erlaubt

Physisch voneinenander unabhängige Bereiche

Es erfolgt die **Partitionierung** des realen Adressraums. Läuft jedoch die CPU im privilegierten Modus, umfasst der Monditoradressraum den Adressraum des Anwendungsprogramms. Ein ähnliches Modell des virtuellen Adressraums verfolgen heute Linux, macOS und Windows.

Schutzgatterregister II



- ein **Zugriffsfehler** führt zum Abbruch der Programmausführung
  - Schutzfehler (*segmentation fault*), *Trap*
- **Problem**:
  - „interne Fragmentierung“: Zugriff auf unbelegten Bereich (*false positive*)

Etwas fälschlich als gültigen Speicherzugriff akzeptieren müssen

Dieses Zugriffsproblem ist Merkmal einer jeden seitennummerierten Technik in Bezug auf den vom Programm **unbelegten Bereich einer Seite**, der jedoch im Adressraum gültig ist. ⇝ **interne Fragmentierung**

Einprogrammbetrieb

Aufgabenverteilung

Abgesetzter Betrieb I

- Berechnung erfolgt getrennt von Ein-/Ausgabe (*off-line*)
  - **Satellitenrechner** zur Ein-/Ausgabe mit „langsamer Peripherie“
    - Kartenleser, Kartenstanzer, Drucker
    - Ein-/Ausgabedaten werden über **Magnetbänder** transferiert
  - **Hauptrechner** zur Berechnung mit „schneller Peripherie“
    - Be-/Entsorgung des Hauptspeichers auf Basis von **Bandmaschinen**
    - dadurch erheblich verkürzte Wartezeiten bei der Ein-/Ausgabe

- **Problem**:
  - sequentieller Bandzugriff, feste Auftragsreihenfolge

Altbewährte Technik von hoher Aktualität

Heute finden zusätzlich **Wechselplatten** Verwendung. Auch die Ausführung von ausgewählten Systemfunktionen auf eigens dafür bereitgestellte Recheneinheiten auszulagern, hat durch **Multiprozessoren** und insb. auch **mehrkernige Prozessoren** eine Erneuerung erfahren [2, 19].

Abgesetzter Betrieb II

- dedizierte Rechner/Geräte für verschiedene Arbeitsphasen:

Phase	Medium			Rechner
Eingabe	Text/Daten Lochkarten	⇒ ⇒	Lochkarten Magnetband	Satellitenrechner
↓				↓
Verarbeitung	Magnetband Hauptspeicher	⇒ ⇒	Hauptspeicher Magnetband	Hauptrechner
↓				↓
Ausgabe	Magnetband Daten	⇒ ⇒	Lochkarten Drucker	Satellitenrechner

- **Problem**:
  - programmierte Ein-/Ausgabe: bei Satelliten-und Hauptrechner

Programmierte E/A

Ein-/Ausgabe, die ausschließlich durch **prozessorseitige Aktionen** vorangetrieben wird und damit eine CPU voll in Anspruch nimmt.

Überlappte Ein-/Ausgabe

- durch **asynchrone Programmunterbrechungen** [8, S. 225–227] die technische Voraussetzung für **gleichzeitige Pozesse** schaffen [1, 17]
  - die E/A-Geräte fordern der CPU weitere E/A-Aufträge ab, plötzlich
  - E/A und Berechnung desselben Programms überlappen sich
- **Speicherdirektzugriff** (*direct memory access*, DMA, [10]) ersetzt die bislang gebräuchliche **programmierte Ein-/Ausgabe**
  - **E/A-Kanäle**, die unabhängig von der CPU arbeiten
    - d.h., Zugriffskanäle zwischen einem E/A-Gerät und dem Hauptspeicher
  - Datentransfer erfolgt durch ein **Kanalprogramm** (*cycle stealing*, [7, 5])
- ermöglicht **nebenläufige Ausführung** von Gerätetreiberprogrammen und dem (einen) Hauptprogramm
  - Kooperation und Konkurrenz gleichzeitiger Prozesse sind zu koordinieren
  - d.h., Erfordernis zur **Synchronisation** [4, S. 28–33]
- **Problem**:
  - Leerlauf beim Auftragswechsel

Überlappte Auftragsverarbeitung

single-stream batch monitor

- Grundlage ist ein als **Verarbeitungsstrom** ausgelegter Stapel (*batch*) sequentiell auszuführender Programme/Aufträge
  - während Ausführung eines Auftrags wird der nachfolgende Auftrag bereits in den Hauptspeicher eingelesen (Zwischenpuffern)
  - Programme werden im **Vorgriffsverfahren** (*prefetching*) abgearbeitet
- **Auftragseinplanung** (*job scheduling*) geschieht im Hintergrund, aber mitlaufend (*on-line*) zur gegenwärtigen Programmausführung
  - statische (*off-line*) Einplanung nach unterschiedlichsten Kriterien
    - Ankunftszeit, erwartete Laufzeit, erwarteter Betriebsmittelbedarf
  - die Aufträge werden dazu im Hintergrundspeicher zusammengestellt
    - *wahlfreier Zugriff* (z.B. Plattenspeicher [6]) beschleunigt die Vorgriffe
- **Problem**:
  - Hauptspeicher, Monopolisierung der CPU, Leerlauf bei Ein-/Ausgabe

Monopolisierung

Jedes Programm bekommt die CPU **exklusiv** zugewiesen und gibt sie erst bei vollendeter Ausführung **freiwillig** ab („*run to completion*“).

Abgesetzte Ein-/Ausgabe I

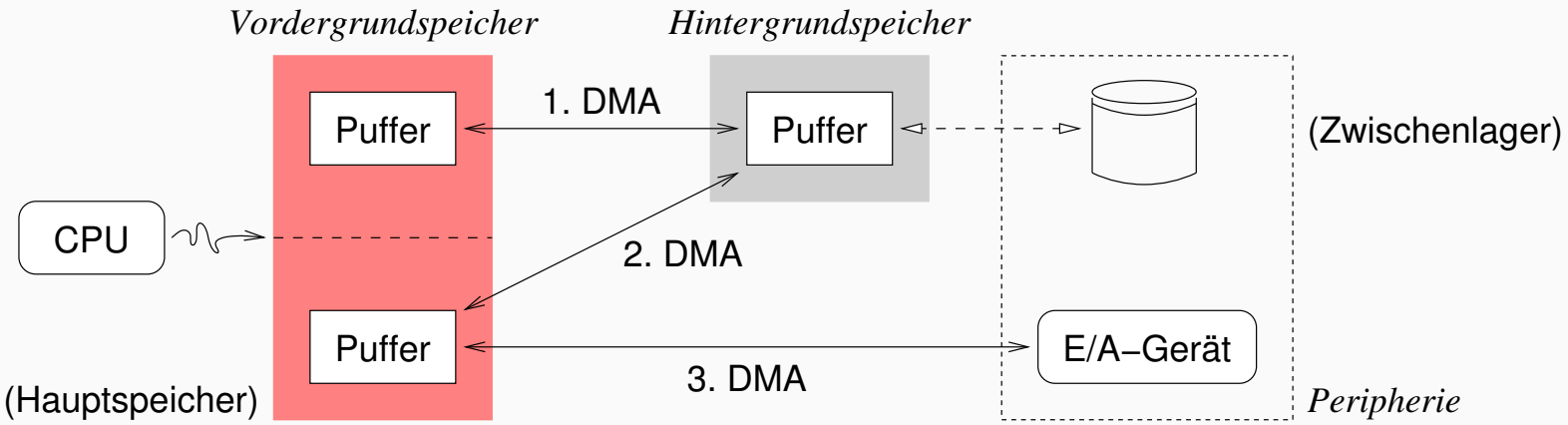
- Programmausführung ist eine periodische Abfolge von Phasen:
  - **CPU-Stoß**
    - Aktionen mit ausschließlich Berechnungen („*CPU burst*“)
    - jede einzelne Aktion verläuft vergleichsweise schnell
  - **E/A-Stoß**
    - Aktionen mit ausschließlich Ein-/Ausgabe („*I/O burst*“)
    - jede einzelne Aktion verläuft vergleichsweise langsam
- CPU- & E/A-Stoß lassen sich durch **Puffer** zeitlich entkoppeln [14]
  - ein CPU-Stoß endet mit der Pufferung das Auftrags für einen E/A-Stoß, er setzt nur einen E/A-Auftrag ab und dann seine Ausführung fort
  - ein E/A-Stoß läuft im Hintergrund der Ausführung des CPU-Stoßes ab, indem ein gepufferter E/A-Auftrag zur Ausführung gebracht wird
- **Problem**:
  - Leerlauf im Wartezustand: konsumier-/wiederverwendbare Betriebsmittel

Spooling (*simultaneous peripheral operations online*)

Bewerkstelligt durch Systemprogramme einerseits zum Absetzen und andererseits zur Überwachung/Steuerung der Verarbeitung von Ein- oder Ausgabeaufträgen (in UNIX: `lpr(1)` bzw. `lpd(8)`).

Abgesetzte Ein-/Ausgabe II

- überlappte E/A befüllt/entleert Datenpuffer — 1. und 2. DMA
  - Puffer im **Vordergrundspeicher** (*primary/main store*) und
  - Puffer im **Hintergrundspeicher** (*secondary/backing store*)
- **effektive E/A** in Bezug auf die Endgeräte operiert dann mit den im Hintergrundspeicher gepufferten Daten — 2. und 3. DMA
  - beansprucht dazu jedoch die CPU und nutzt wiederum überlappte E/A



Abgesetzter Betrieb

Hier sind die im Hauptspeicher liegenden Puffer dem Haupt- (oben) und Satellitenrechner (unten) zugeordnet.

Gliederung

Mehrprogrammbetrieb

Multiplexverfahren

Maßnahmen zur Leistungssteigerung

- in Raum vermöge **Mehrprogrammbetrieb** (*multi-programming*) und Zeit mittels **Simultanverarbeitung** (*multiprocessing*)
  - Multiplexen der CPU zwischen mehreren Programmen[2]
    - **Nebenläufigkeit** (*concurrency*)
  - Abschottung der sich in Ausführung befindlichen Programme
    - **Adressraumschutz** (*address space protection*)
  - Überlagerung unabhängiger Programmteile
    - **Segmentierung** (*segmentation*, [15]) — im Sinne von „Aufteilung“
- beides auch Maßnahmen eines Betriebssystems zur Unterstützung der besseren Strukturierung von Programmen

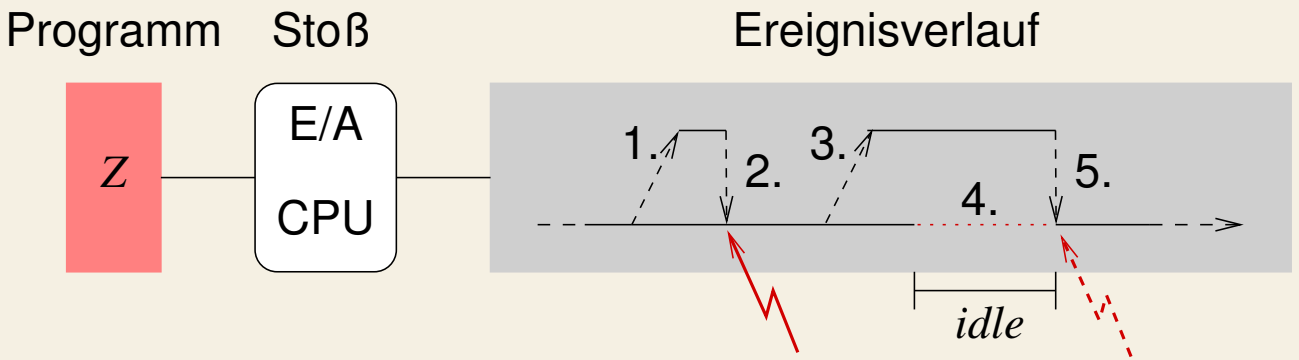
Nichtsequentielle (System-) Programmierung

*If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate **languages** that give us the one without the other? (Alan Perlis [16, Epigram 68])*

[2] Zunächst als „befremdliches Ergebnis“ von *Interrupts* angesehen [12, S. 43].

Aktives Warten

- überlappte Ein-/Ausgabe bedeutet die parallele Ausführung von CPU- & E/A-Stößen — im **Einprogrammbetrieb** jedoch nicht immer



1. Programm Z löst einen zum CPU-Stoß nebenläufigen E/A-Stoß aus
2. die Beendigung des E/A-Stoßes wird durch eine Unterbrechung angezeigt
3. der Unterbrechungshandhaber[3] löst einen weiteren E/A-Stoß aus
4. Z muss auf E/A warten, dem logischen Verlauf nach endet der CPU-Stoß
5. die CPU ist untätig (*idle*) bis der E/A-Stoß endet, ein CPU-Stoß beginnt

- **Problem**:
  - Durchsatz, Auslastung (CPU, E/A-Geräte)

[3] *interrupt handler*

Multiplexen des Prozessors

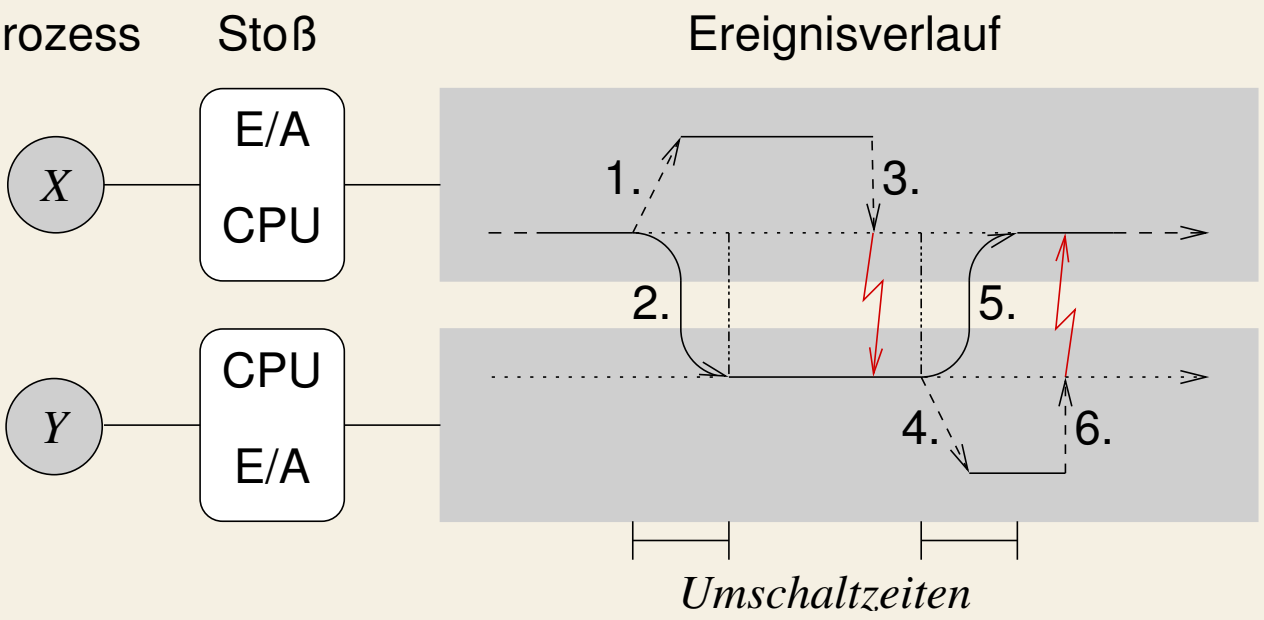
- die Auslastung der CPU steigt, wenn die Wartezeit eines Programms als Laufzeit eines anderen Programms nutzbar ist
  - aktives Warten (*busy waiting*) ist grundsätzlich unproduktiv und beschert der CPU nur kostbare **Leerlaufzeit** (*idle time*)
  - im Falle eines alternativen Ausführungsstrangs ist eine solche Wartephase allerdings als **Produktivzeit** nutzbar
  - dazu ist die CPU zur Aufnahme eines solchen Ausführungsstrangs (eines anderen Programms) umzuschalten $\rightsquigarrow$ „passives Warten“
- der **Umschaltmechanismus** selbst ist unabhängig von den konkret auszuführenden Programmen, er ist generisch auslegbar
  - dazu wird von dem „Programm in Ausführung“ abstrahiert, der **Prozess** eingeführt, dessen Inkarnation einen eigenen **Prozessorzustand** besitzt
  - Umschalten der CPU bedeutet dann lediglich, den Prozessorzustand eines anderen Prozesses zu aktivieren

Leerlaufzeit

Es gibt jedoch Fälle, wo aktives Warten auf einen Ereigniseintritt die effiziente Alternative darstellt: **Umschaltzeit** $>$ erwartete Wartezeit.

Passives Warten I

- Programmverarbeitung im **Mehrprozessbetrieb**:



  - Prozesse X und Y als Produkte desselben nichtsequentiellen Programms oder verschiedener sequentieller Programme
  - der CPU-Stoß von Prozess Y wird durch Beendigung des E/A-Stoßes von Prozess X unterbrochen und umgekehrt
  - solche Unterbrechungsszenarien rufen unvorhersehbare **Interferenz** hevor, die für zeitabhängige Prozesse kritisch sein kann

Passives Warten II

…zum Programmablauf auf S. 33

- ein Prozess, der warten muss, gibt die CPU zugunsten eines anderen Prozesses ab, der nicht warten muss:
  Prozessormultiplexverfahren

1. Prozess *X* löst einen E/A-Stoß und beendet seinen CPU-Stoß
2. Prozess *Y* wird eingelastet und beginnt seinen CPU-Stoß
3. Beendigung des E/A-Stoßes von Prozess *X* unterbricht Prozess *Y*
   - die Unterbrechungsbehandlung überlappt sich mit Prozess *Y*
4. Prozess *Y* löst einen E/A-Stoß und beendet seinen CPU-Stoß
5. Prozess *X* wird eingelastet und beginnt seinen CPU-Stoß
6. Beendigung des E/A-Stoßes von Prozess *Y* unterbricht Prozess *X*
   - die Unterbrechungsbehandlung überlappt sich mit Prozess *X*

- **Einlastung** (*dispatching*) eines Prozesses ist der Moment, in dem die CPU umgeschaltet wird
  - den Prozessorzustand des die CPU abgebenden Prozesses sichern und des die CPU erhaltenden Prozesses (wieder) herstellen
  - das Betriebssystem vollzieht genau damit den **Prozesswechsel**
- **Problem**:
  - Interferenz, Adressraumschutz, Koordination

Mehrprogrammbetrieb

Schutzvorkehrungen

Adressraumschutz

- bei mehr als einem Programm, das neben dem residenten Monitor im Hauptspeicher liegen muss, reichen Schutzgatter nicht
  - jedes einzelne Programm ist von anderen Programmen zu isolieren
- der **Bindungszeitpunkt**, zu dem Namen mit Speicherorten verknüpft werden, begründet verschiedene Schutztechniken:

  vor Laufzeit $\rightsquigarrow$ Schutz durch **Einfriedung**
  - Programme laufen (weiterhin) im realen Adressraum
  - ein **verschiebender Lader** besorgt die Bindung zur *Ladezeit*
  - die CPU/MPU überprüft die realen Adressen zur Ausführungszeit

  zur Laufzeit $\rightsquigarrow$ Schutz durch **Segmentierung** [3]
  - jedes Programm läuft in seinem eigenen **logischen Adressraum**
  - der Lader bestimmt die Bindungsparameter (reale Basisadresse)
  - die CPU (bzw. MMU) besorgt die Bindung zur *Ausführungszeit*
    - dynamische Programmumlagerung (*program relocation*, [13])

Positionsrelative Ausrichtung des Adressraums

In beiden Fällen ist der Namensraum eines Programms relativ zu der logischen Anfangsadresse 0 ausgerichtet.

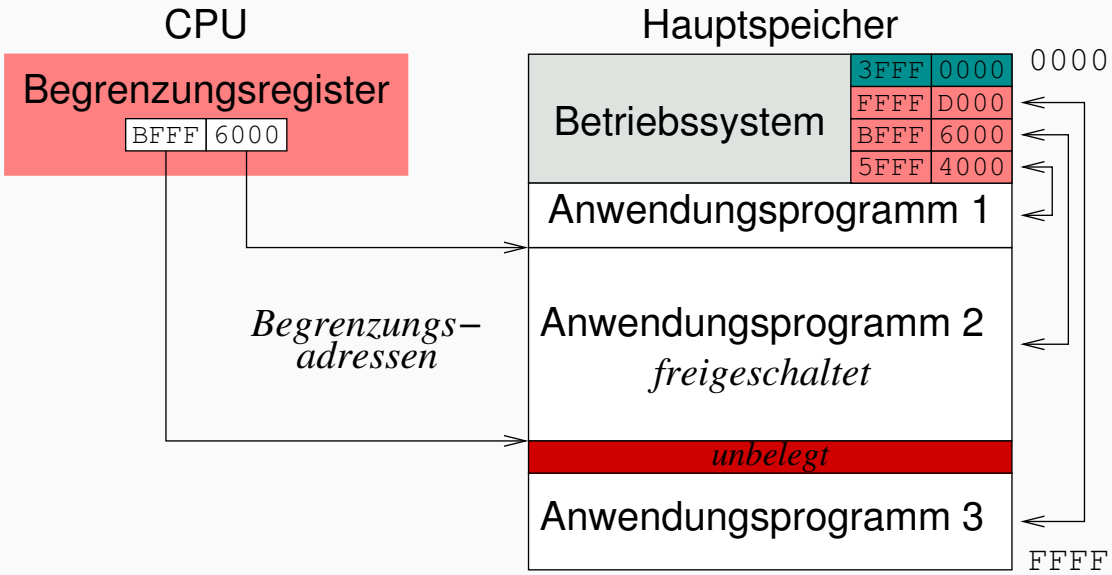
Adressraumschutz durch Einfriedung

boundary fence

- **Begrenzungsregister** (*bounds register*) legen die Unter-/Obergrenze eines Programms im realen Adressraum fest
  - jedem Programm wird ein solches Wertepaar spätestens zur Ladezeit fest zugeordnet und seiner Prozessinkarnation zur „Einzäunung“ mitgegeben
    - die *Softwareprototypen* dieser Register der **Adressüberprüfungshardware**
    - verwaltet als Attribute des Prozesskontrollblocks
  - bei Prozesseinlastung werden die *Hardwareprototypen* dieser Register mit den in den Softwareprototypen vermerkten Werten definiert
    - wodurch der Hauptspeicherbereich des Prozesses „freigeschaltet“ wird
    - zu einem Zeitpunkt ist damit immer nur ein solcher Bereich freigegeben
- innerhalb des durch die Begrenzungsregister festgelegten Bereichs ist dynamischer Speicher bedingt zuteilbar
  - der Bedarf dafür muss jedoch spätestens zum Ladezeitpunkt bekannt sein
  - das Betriebssystem kann nur **statische Speicherverwaltung** betreiben
- **Problem**:
  - Fragmentierung, Verdichtung

Begrenzungsregister I

bounds register

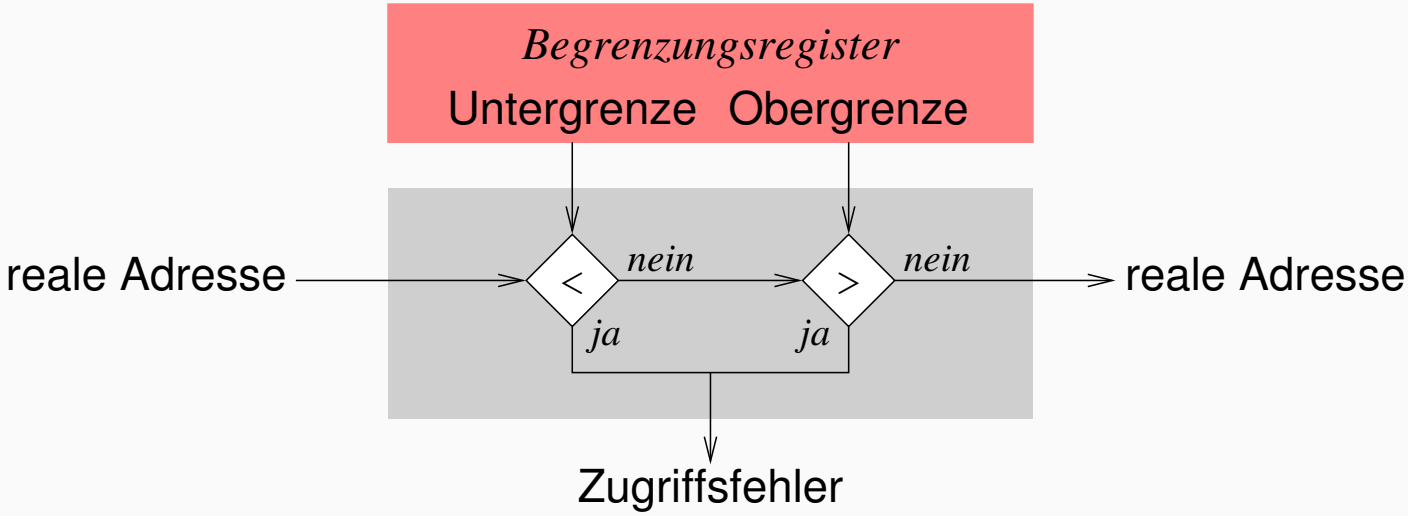


- konsequente Umsetzung des Modells ist es, Anwendungsprogramme auch vor direkten Zugriffen durch das Betriebssystem zu schützen
  - in dem Fall verfügt jeder Arbeitsmodus über eigene Begrenzungsregister
  - ein **Arbeitsmoduswechsel** aktiviert das jeweilige Registerpaar implizit

Hinweis

Heute in Form einer „memory protection unit“ (MPU) gebräuchlich.

Begrenzungsregister II



- ein **Zugriffsfehler** führt zum Abbruch der Programmausführung
  - Schutzfehler (*segmentation fault*), *Trap*
- **Problem** wie beim Schutzgatterkonzept (vgl. S. 20), zusätzlich:
  - „externe Fragmentierung“, d.h., unbelegte Hauptspeicherbereiche

Externe Fragmentierung

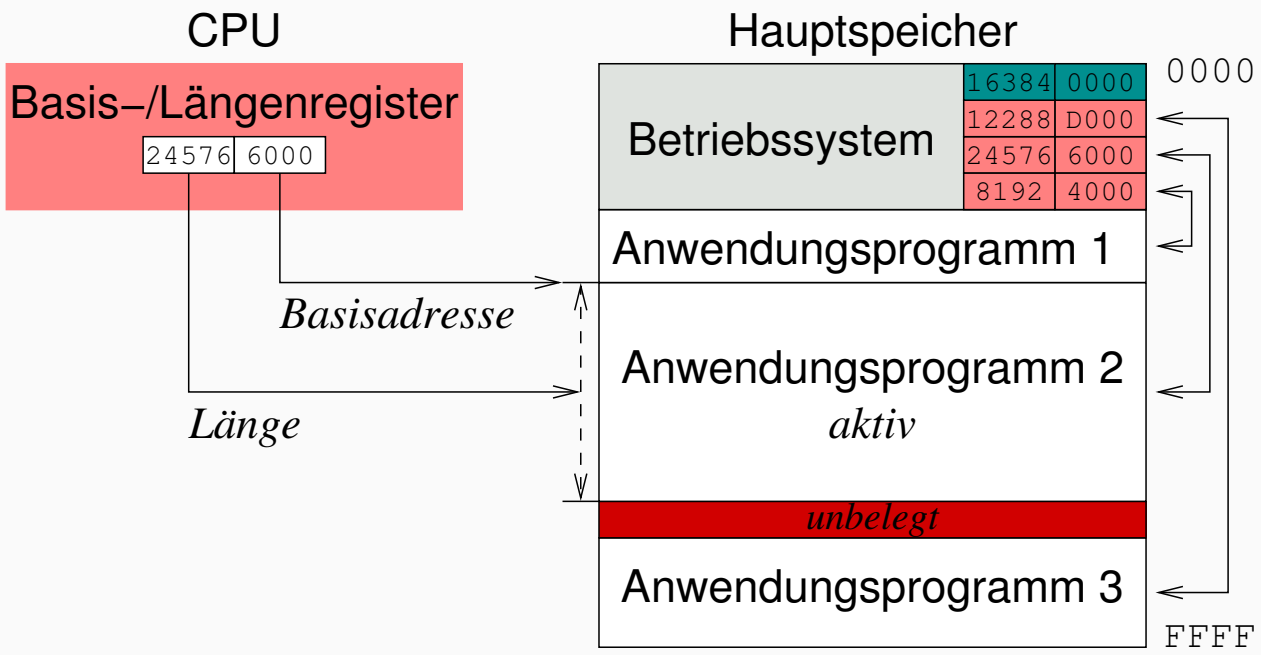
Freie Speicherbereiche sind jeder für sich zu klein, insgesamt jedoch groß genug für die Speicheranforderung eines Prozesses. Sie bleiben dennoch nutzlos, da sie im realen Adressraum nicht linear zusammenhängend angeordnet sind.

Adressraumschutz durch Segmentierung

- **Basis-/Längenregister** (*base/limit register*) legen Segmentort sowie -größe eines Programms im realen Adressraum fest
  - jedem Programm wird ein solches Wertepaar zur Ladezeit zugeordnet und seiner Prozessinkarnation mitgegeben
    - die *Softwareprototpypen* dieser Register der **Adressverschiebungshardware**
    - verwaltet als Attribute des Prozesskontrollblocks
  - bei Prozesseinlastung werden die *Hardwareprototypen* dieser Register mit den in den Softwareprototypen vermerkten Werten definiert
    - wodurch der Hauptspeicherbereich des Prozesses „aktiviert“ wird
    - zu einem Zeitpunkt ist damit immer nur ein solches Segment freigegeben
- innerhalb des durch die Basis-/Längenregister definierten Bereichs ist dynamischer Speicher bedingt zuteilbar
  - der Bedarf dafür braucht erst zur Ausführungszeitpunkt bekannt zu sein
    - ist die **maximale Größe** nicht erreicht, kann das Segment expandiert werden
    - verhindert dies ein angrenzendes Segment, wird das Programm umgelagert
    - ggf. wird der Hauptspeicher für einen passenden Bereich verdichtet
  - das Betriebssystem kann **dynamische Speicherverwaltung** betreiben
- **Problem**:
  - externe Fragmentierung, Hauptspeichergröße

Basis-/Längenregister I

base/limit register

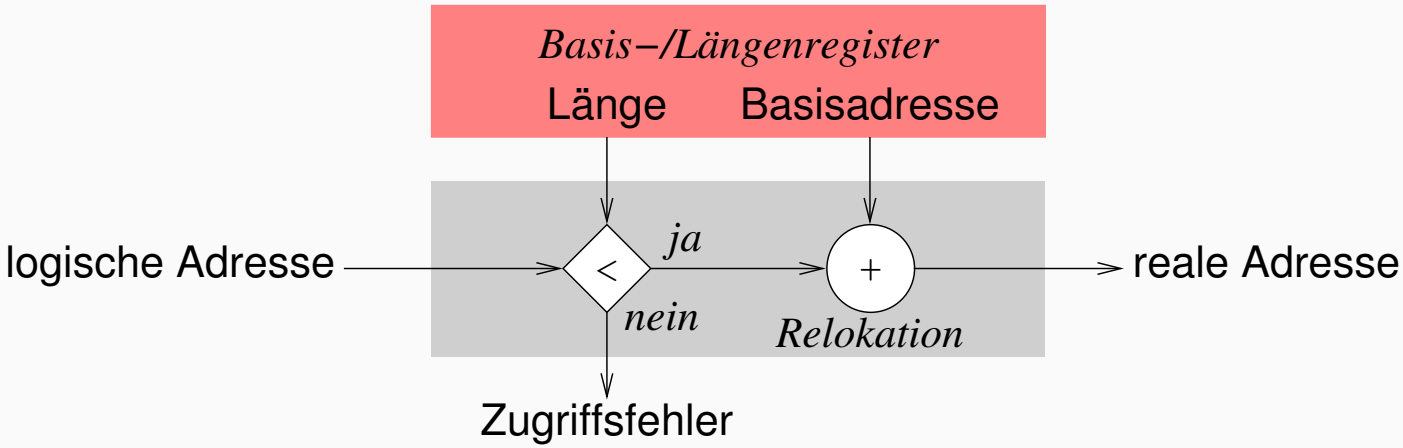


- das Betriebssystem ist in einem eigenen Segment gekapselt, durch Basis-/Längenregister für den privilegierten Arbeitsmodus

Segmentierende MMU

Kennzeichnend für eine „*memory management unit*“ (MMU), allerdings wie hier gezeigt nur zur eher unüblichen Isolation eines ganzen ausführbaren Programms durch ein einzelnes Segment.

Basis-/Längenregister II



- ein **Zugriffsfehler** führt zum Abbruch der Programmausführung
  - Schutzfehler (*segmentation fault*), *Trap*
- **Problem**:
  - Unisegmentansatz, d.h., pro Programm nur ein Segment

Verlagerung

Die MMU verschiebt die bei Programmausführung erzeugte effektive logische Adresse um die Segmentbasisadresse (Verlagerungskonstante/-variable). Der Basisregisterinhalt ist zwischen den einzelnen Ausführungsphasen veränderlich, das Programm kann somit zur Laufzeit im Hauptspeicher umgelagert werden.

Mehrprogrammbetrieb

Dynamisches Laden

Überlagerungstechnik I

overlay [15]

- ein einzelnes Programm ausführen zu können, obwohl es in seiner Gesamtheit nicht in den Hauptspeicher passt:
  - i es ist überhaupt zu groß für den Hauptspeicher, auch falls es als einziges Anwendungsprogramm zur Ausführung kommen soll, oder
  - ii es ist größer als die einem Anwendungsprogramm statisch zur Verfügung stehende **Hauptspeicherpartition**
- dazu wird das Programm in hinreichend kleine Teile zergliedert, die nicht ständig im Hauptspeicher vorhanden sein müssen
  - dies betrifft sowohl den Text- als auch den Datenbereich
  - nicht benötigte Teile liegen abrufbereit im Hintergrundspeicher
  - sie werden bei Bedarf nachgeladen, überlagern nicht mehr benötigte Teile
- das Nachladen ist programmiert, d.h., in dem Programm festgelegt, die Entscheidung dazu fällt zur Programmlaufzeit
  - ein Programm löst **dynamisches Laden** von Überlagerungen aus
- **Problem**:
  - „optimale“ Überlagerungsstruktur, manueller Ansatz

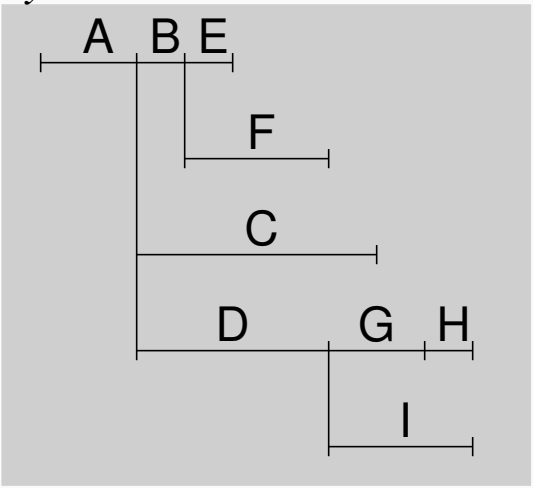
Überlagerungstechnik II

- Hauptspeicherbedarf für ein unzerteiltes Programm:

statisch

- Hauptspeicherbedarf für ein zerteiltes Programm:

dynamisch



eingelagert

ABE

ABF

AC

ADGH

ADI

  - nicht alle Bestandteile (A – I) eines Programms müssen gleichzeitig im Hauptspeicher vorliegen, damit das Programm auch ausführbar ist
  - wann welches Programmteil zur Ausführung gelangt, legt der dynamische Ablauf des Programms selbst fest

Mehrprogrammbetrieb

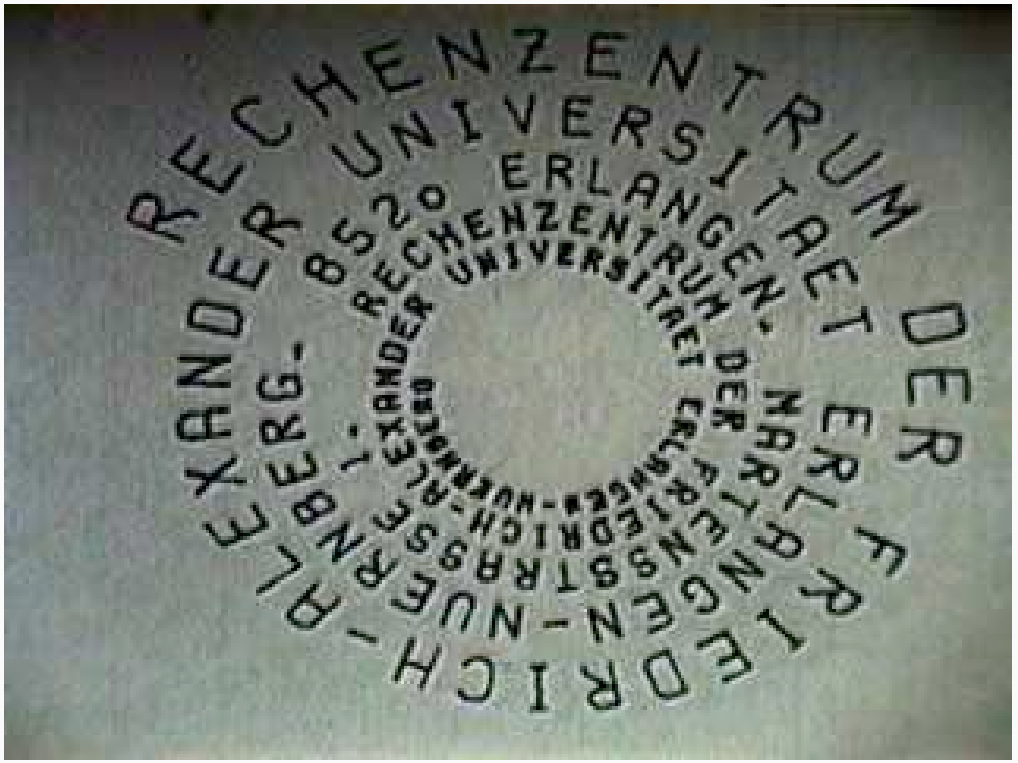
Simultanverarbeitung

Multistapelbetrieb

- **Simultanverarbeitung** (*multiprocessing*) mehrerer Auftragsströme
  - das Betriebssytem als „*multi-stream batch monitor*“
- echt/quasi parallele Verarbeitung von Auftragsströmen sequentieller Programme, gleichzeitig, im **Multiplexverfahren**
  - sequentielle Ausführung der Programme desselben Auftragstapels
    - „wer [innerhalb des Stapels] zuerst kommt, mahlt zuerst“…
  - nichtsequentielle Ausführung der Programme verschiedener Auftragstapel
- die Stapelwechsel kommen **kooperativ** (*cooperative*), **verdrängend** (*preemptive*) oder als Kombination von beiden zustande
  - kooperativ bei (programmierter) Ein-/Ausgabe, also bei Beendigung des CPU-Stoßes eines Prozesses, und am Programmende
  - verdrängend bei Ablauf einer Frist (*time limit*), innerhalb welcher die Ausführung eines jeweiligen Programms abgeschlossen sein muss
- **Problem**:
  - interaktionsloser Betrieb, Mensch/Maschine-Schnittstelle

Gliederung

Rechnerbetrieb am RRZE um 1973



[4] https://www.video.uni-erlangen.de/clip/id/4251.html

Resümee

…immer nach dem gleichen Schema zu verrichtende Arbeit

- **Stapelbetrieb** meint die sequentielle Abwicklung von Aufgaben
  - anfangs manuelle Bestückung des Rechners durch Programmierpersonal
  - später automatisierte Bestückung mittels **Kommandointerpreter**
    - Programmierer organisier(t)en ihren Auftragsstapel mittels Steuerkarten
    - Operateure übern{a,e}hmen die manuelle Bestückung des Rechners
  - **Aufgabenverteilung** dehnt(e) sich auch auf systeminterne Abläufe aus
    - abgesetzter Betrieb, überlappte Ein-/Ausgabe
    - überlappte Auftragsverarbeitung, abgesetzte Ein-/Ausgabe
- **Mehrprogrammbetrieb** hält mehrere Programme im Hauptspeicher
  - Hauptspeicherknappheit wird durch dynamisches Laden begegnet
  - der Prozessor wird im Multiplexverfahren betrieben: **Prozesse**
  - Schutzvorkehrungen schotten Prozessadressräume voneinander ab
  - um das Rechensystem damit sicher in **Simultanbetrieb** fahren zu können
- nach wie vor bestens für **Routinearbeit** geeignet, wobei:
  - nur Operateure interaktiven Zugang zum Rechensystem bek{a,om}men
  - die Anwender mit dem Rechensystem jedoch interaktionslos arbeit(et)en

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Bell, C. G. ; Newell, A. :
Computer Structures: Readings and Examples.
New York, NY, USA : McGraw-Hill Inc., 1971. –
668 S.

[2] Brüning, U. ; Giloi, W. K. ; Schröder-Preikschat, W. :
Latency Hiding in Message-Passing Architectures.
In: Siegel, H. J. (Hrsg.): *Proceedings of the 8th International Symposium on Parallel Processing, April 26–29, 1994, Cancún, Mexico*.
Washington, DC, USA : IEEE Computer Society, 1994. –
ISBN 0–8186–5602–6, S. 704–709

[3] Dennis, J. B.:
Segmentation and the Design of Multiprogrammed Computer Systems.
In: *Journal of the ACM* 12 (1965), Okt., Nr. 4, S. 589–602

Literaturverzeichnis (2)

[4] Dijkstra, E. W.:
***Communication with an Automatic Computer*, Universiteit van Amsterdam, Diss., Okt. 1959**

[5] IBM:
Data Processing Machine Including Program Interrupt Feature.
U.S. Pat. No. 3,319,230 (1967), 1956

[6] IBM:
IBM 503 RAMAC.
`http://www-03.ibm.com/ibm/history/exhibits/storage/storage_PH0305.html`, 1956

[7] IBM:
709 Data Processing System.
`http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html`, Jan. 1957

Literaturverzeichnis (3)

[8] Knuth, D. E.:
***The Art of Computer Programming*. Bd. 1: *Fundamental Algorithms*.**
Reading, MA, USA : Addison-Wesley, 1973

[9] Larner, R. A.:
FMS: The FORTRAN Monitor System.
In: Butler, M. K. (Hrsg.) ; Brown, J. M. (Hrsg.) ; American Federation of Information Processing Societies, Inc. (Veranst.): *1987 Proceedings of the National Computer Conference, June 15–18, 1987, Chicago, Illinois, USA* Bd. 56 American Federation of Information Processing Societies, Inc., AFIPS Press, 1987, S. 815–820

[10] Leiner, A. L.:
System Specifications for the DYSEAC.
In: *Journal of the ACM* 1 (1954), Apr., Nr. 2, S. 57–81

Literaturverzeichnis (4)

[11] Leiner, A. L. ; Alexander, S. N.:
System Organization of DYSEAC.
In: *IRE Transactions on Electronic Computers* EC-3 (1954), März, Nr. 1, S. 1–10

[12] Loopstra, B. J.:
The X-1 Computer.
In: *The Computer Journal* 2 (1959), Nr. 1, S. 39–43

[13] McGee, W. C.:
On Dynamic Program Relocation.
In: *IBM Systems Journal* 4 (1965), Sept., Nr. 3, S. 184–199

[14] Mock, O. ; Swift, C. J.:
The Share 709 System: Programmed Input-Output Buffering.
In: *Journal of the ACM* 6 (1959), Apr., Nr. 2, S. 145–151

Literaturverzeichnis (5)

[15] Pankhurst, R. J.:
Operating Systems: Program Overlay Techniques.
In: *Communications of the ACM* 11 (1968), Febr., Nr. 2, S. 119–125

[16] Perlis, A. J.:
Epigrams on Programming.
In: *SIGPLAN Notices* 17 (1982), Nr. 9, S. 7–13

[17] Smotherman, M. :
***Interrupts*.**
`http://www.cs.clemson.edu/~mark/interrupts.html`, Jul. 2008

Literaturverzeichnis (6)

[18] US War Department, Bureau of Public Relations:
For Radio Broadcast: Physical Aspect, Operation of ENIAC are Described.
`http://americanhistory.si.edu/collections/comphist/pr4.pdf`, Febr. 1946

[19] Wentzlaff, D. ; Agarwal, A. :
Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores.
In: *ACM SIGOPS Operating Systems Review* 43 (2009), Apr., Nr. 2, S. 76–85

Anhang

Präludium

Generationen von Hardware und Betriebssoftware

- zeitliche Einordnung von Technologiemerkmalen
  - „unscharf“, Übergänge fließend

Generation	Epoche	Hardware	Rechnerbetriebsart
1	1945	Röhre	Stapelbetrieb
2	1955	Halbleiter, Gatter	Echtzeitbetrieb
3	1965	MSI/LSI, WAN	Mehrprogrammbetrieb
4	1975	VLSI, LAN	Mehrzugangsbetrieb
5	1985	ULSI, RISC	Massenparallelbetrieb
?	1995	WLAN, RFID, SoC	?
?	?	?	Quantenrechenbetrieb

Legende: MSI, LSI, VLSI, ULSI — *medium*, *large*, *very large*, *ultra large scale integration*
LAN — *local area network* (Ethernet)
WAN — *wide area network* (Arpanet)
RISC — *reduced instruction set computer*
WLAN — *wireless* LAN
RFID — *radio frequency identification* (Funkerkennung)
SoC — *system on chip*

Anhang

Dynamisches Laden

Überlagerungsstruktur eines Programms

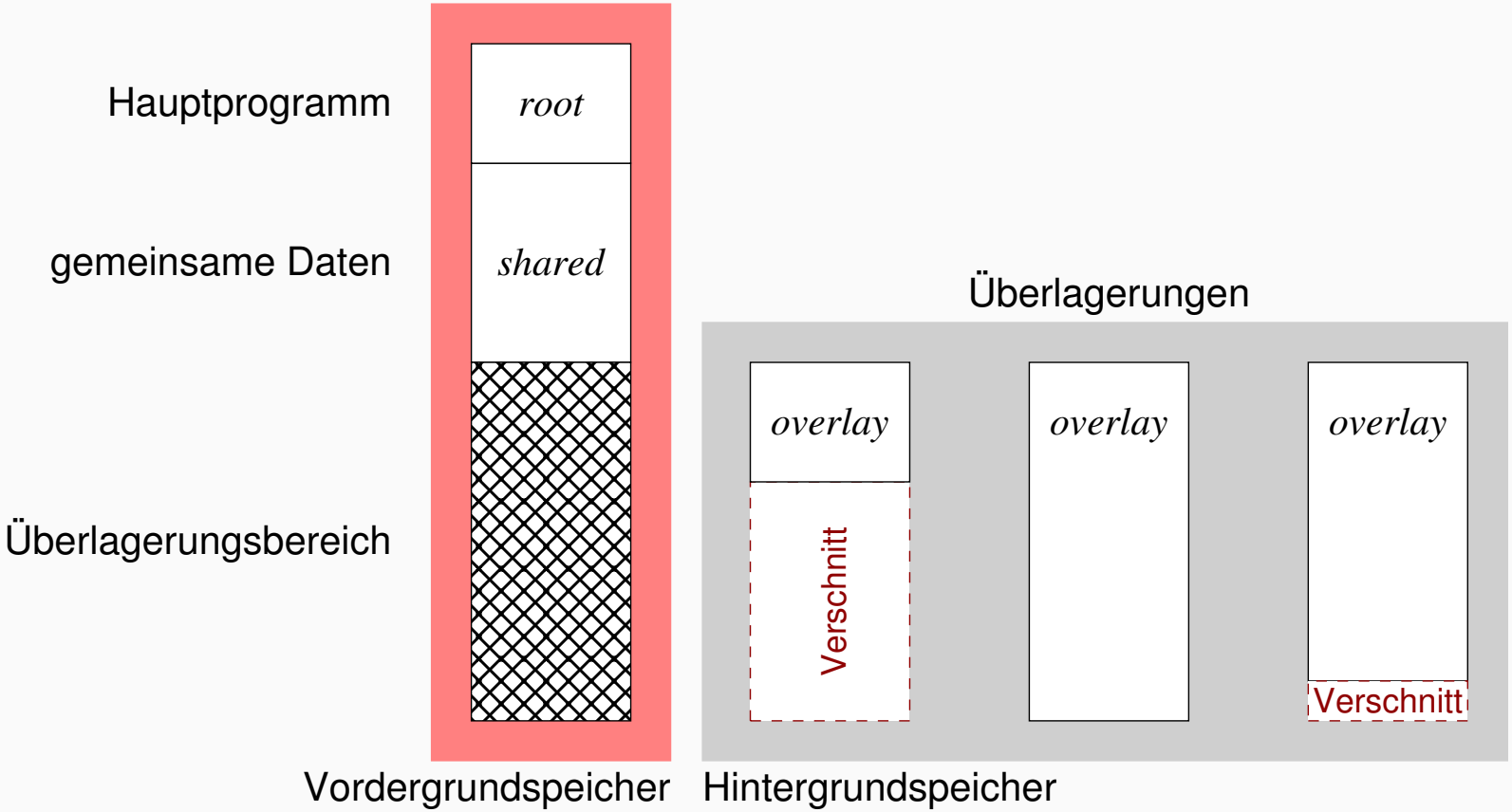
- Programme, die Überlagerungen enthalten, sind (grob) dreigeteilt:
  i. ein hauptspeicherresidenter Programmteil (*root program*), repräsentiert das Überlagerungen aufrufende Hauptprogramm
  ii. mehrere in Überlagerungen zusammengefasste Unterprogramme, die ihrerseits Überlagerungen aufrufen können
  iii. ein gemeinsamer Datenbereich (*common section*), zur Speicherung überlagerungsübergreifender Information
- Überlagerungen sind das Ergebnis einer **Programmzerlegung** zur Programmier-, Übersetzungs- und/oder Bindezeit
  - manuelle bzw. automatisierte **Abhängigkeitsanalyse** des Programms
  - Anzahl und Größe von Überlagerungen werden **statisch** festgelegt
  - lediglich aktivieren (d.h. laden) von Überlagerungen ist dynamisch

Hinweis

Überlagerungen sind Teile eines Programms und keine Elemente einer mehreren Programmen gemeinsamen Bibliothek (shared library) oder DLL (dynamic link library).

Programm-/Adressraumstruktur

- Organisation des Prozessadressraums im Hauptspeicher:



- **Problem**:
  - Verkettungstechnik, kein Zurückschreiben von Überlagerungen

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VII.2 Betriebsarten: Dialog- und Echtzeitverarbeitung

---

11. Juli 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

Gliederung

Lehrstoff

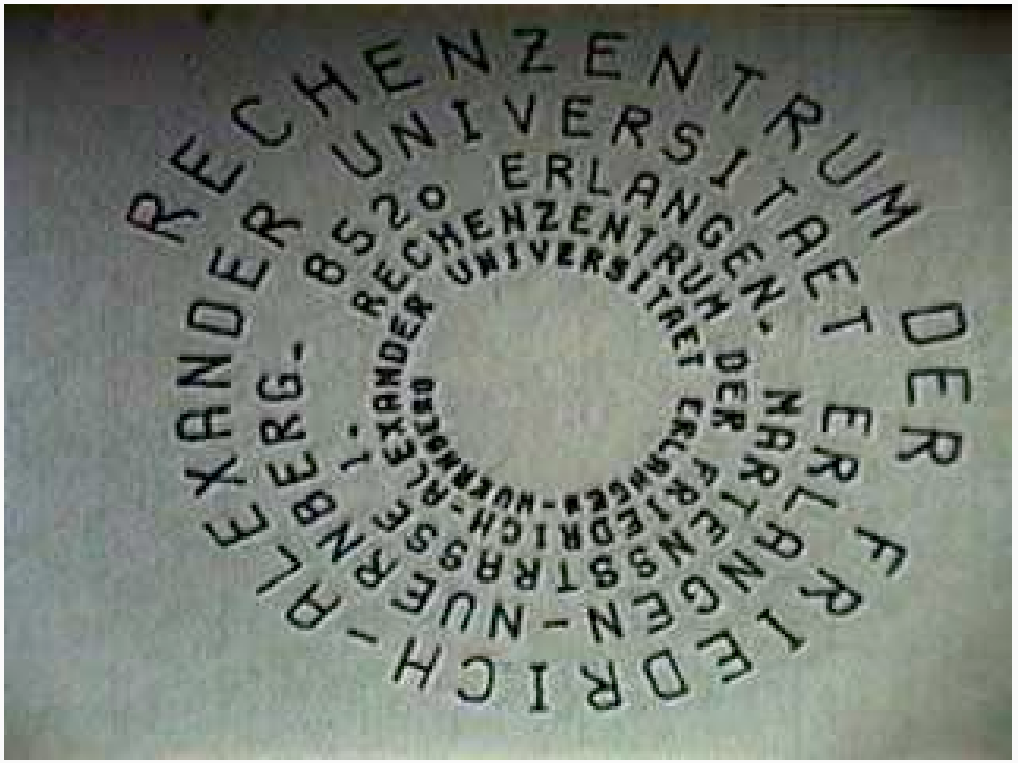
- weiterhin ist das Ziel, „zwei Fliegen mit einer Klappe zu schlagen“:
  i einen Einblick in **Betriebssystemgeschichte** zu geben und
  ii damit gleichfalls **Betriebsarten** von Rechensystemen zu erklären
- im Vordergrund stehen die Entwicklungsstufen im **Dialogbetrieb**, der Dialogprozesse einführt, d.h., Prozesse:
  - die an der Konkurrenz um gemeinsame Betriebsmittel teilnehmen
  - die Benutzer/innen an einer Dienstleistung teilhaben lassen
- kennzeichnend ist, Programmausführung **interaktiv** zu gestalten
  - mitlaufend (*on-line*) den Prozessfortschritt beobachten und überwachen
  - dazu spezielle Schutzvorkehrungen und eine effektive Speicherverwaltung

Hinweis

*Viele dieser Techniken — wenn nicht sogar alle — sind auch heute noch in einem **Universalbetriebssystem** auffindbar.*

- des Weiteren erfolgt ein kurzer Einblick in den **Echtzeitbetrieb**, der an sich quer zu all den betrachteten Betriebsarten liegt

Rechnerbetrieb am RRZE[1] um 1973



[1] https://www.video.uni-erlangen.de/clip/id/4251.html

Gliederung

Mehrzugangsbetrieb

Allgemeines

Dialogbetrieb

conversational mode

- Benutzereingaben und Verarbeitung wechseln sich anhaltend ab
  - E/A-intensive Anwendungsprogramme interagieren mit Benutzer/inne/n
  - Zugang über **Dialogstationen** (*interactive terminals*)
    - **Datensichtgerät** und **Tastatur** (seit 1950er Jahren, Whirlwind/SAGE)
    - später die ***Maus*** (Engelbart/English, SRI, 1963/64; vorgestellt 1968)
- **dynamische Einplanung** (*dynamic scheduling*, *on-line*) hält Einzug, bevorzugt **interaktive** (E/A-intensive) **Prozesse**
  - Beendigung von Ein-/Ausgabe führt zur „prompten“ Neueinplanung
    - im Falle von E/A-Operationen, die sich blockierend auswirken
  - Benutzer/innen erfahren eine schnelle Reaktion insb. auf Eingaben
    - sofern auch die Einlastung von Prozessen „prompt“ geschieht
- **Problem**:
  - Zusatz (*add-on*) zum Stapelbetrieb, Monopolisierung der CPU, Sicherheit

Anekdote (*add-on*: eines Studenten)

Hin und wieder verliefen Sitzungen über CMS (conversational monitor system) an den Dialogstationen des IBM System/360 schon recht träge. Unter den Studierenden hatte sich schnell herumgesprochen, mittels Tastatureingaben die Dringlichkeit ihrer im Hintergrund ablaufenden Programmausführung anheben zu können.

Dialogororientiertes Monitorsystem

- Prozesse „im Vordergrund“ starten & „im Hintergrund“ vollziehen
  - in **Konversation** Aufträge annehmen, ausführen und dabei überwachen
    - d.h. Prozesse starten, stoppen, fortführen und ggf. abbrechen
  - zur selben Zeit laufen im Rechensystem mehrere Programme parallel ab
  - mehrere Aufgaben (*task*) werden „gleichzeitig“ bearbeitet (*multitasking*)
- in weiterer Konsequenz lässt sich so **Mischbetrieb** unterstützen:

Vordergrund	echtzeitabhängige Prozesse $\rightsquigarrow$ Echtzeitbetrieb (Realzeit)
Mittelgrund	E/A-intensive Prozesse $\rightsquigarrow$ Dialogbetrieb (Antwortzeit)
Hintergrund	CPU-intensive Prozesse $\rightsquigarrow$ Stapelbetrieb (Rechenzeit)

- **Problem**:
  - Hauptspeicher(größe)

Mischbetrieb

Zeit ist ein wichtiger Aspekt, jedoch ist dabei das **Bezugssystem** zu beachten: Antwort-/Rechenzeit hat nur das Rechensystem, Echtzeit jedoch vor allem die (phys.) Umgebung als Bezugsrahmen.

Mehrzugangsbetrieb

Teilnehmerbetrieb

Dialog mit teilnehmenden Prozessen

time sharing

- eigene Dialogprozesse werden interaktiv gestartet und konkurrieren mit anderen (Dialog-) Prozessen um gemeinsame Betriebsmittel
  - um CPU-Monopolisierung vorzubeugen, werden CPU-Stöße partitioniert, indem Prozesse nur eine **Zeitscheibe** (*time slice*) lang „laufend“ sind
    - ist die Zeitscheibe abgelaufen, wird der Prozess von der CPU verdrängt
    - er erhält die CPU sodann für eine neue Zeitscheibe erneut zugeteilt
  - CPU-Zeit ist damit eine Art **konsumierbares Betriebsmittel**, um das wiederverwendbare Betriebsmittel „CPU“ beanspruchen zu können
    - jeder Dialogprozess „nimmt teil“ an der **Konkurrenz** um Betriebsmittel
- technische Grundlage liefert ein **Zeitgeber** (*timer*), der für **zyklische Unterbrechungen** (*timer interrupt*) sorgt
  - der unterbrochene Prozess wird neu eingeplant: „laufend“ $\mapsto$ „bereit“
    - ihm wird die CPU zu Gunsten eines anderen Prozesses entzogen
    - er erfährt die **Verdrängung** (*preemption*) von „seinem“ Prozessor
  - aber nur, sofern es einen anderen Prozess im Zustand „bereit“ gibt
- **Problem**:
  - Hauptspeicher, Einplanung, Einlastung, Ein-/Ausgabe, Sicherheit

Bahnbrecher und Wegbereiter I

- **CTSS** (*Compatible Time-Sharing System* [1], MIT, 1961)
  - Pionierarbeit zu interaktiven Systemen und zur Prozessverwaltung
    - **partielle Virtualisierung**: Prozessinkarnation als virtueller Prozessor
    - **mehrstufige Einplanung** (*multi-level scheduling*) von Prozessen
    - zeilenorientierte Verarbeitung von Kommandos (u.a. `printf` [1, S. 340])
  - vier Benutzer gleichzeitig: drei im Vordergrund, einen im Hintergrund[2]
- **ITS** (*Incompatible Time-sharing System* [5], MIT, 1969)
  - Pionierarbeit zur Ein-/Ausgabe und Prozessverwaltung:
    - **geräteunabhängige Ausgabe** auf Grafikbildschirme, virtuelle Geräte
    - netzwerktransparenter Dateizugriff (über ARPANET [24])
    - **Prozesshierarchien**, Kontrolle untergeordneter Prozesse (`^Z` [5, S. 13])
  - „Seitenhieb“ auf CTSS und Multics, wegen der eingeschlagenen Richtung

Zeitteilverfahren

Time-sharing was a misnomer. While it did allow the sharing of a central computer, its success derives from the ability to share other resources: data, programs, concepts. [22]

[2] *Time-sharing introduced the engineering constraint that the interactive needs of users [were] just as important as the efficiency of the equipment. (F. J. Corbató)*

Mehrzugangsbetrieb

Teilhaberbetrieb

Dialog mit teilhabenden Prozessen

transaction mode

- ein von mehreren Dialogstationen aus gemeinsam benutzter, zentraler Dialogprozess führt die abgesetzten Kommandos aus
  - mehrere Benutzer „haben Teil“ an der Dienstleistung eines Prozesses, die Bedienung regelt ein einzelnes Programm
  - gleichartige, bekannte und festverdrahtete (*hard-wired*) Aktionen können von verschiedenen Benutzern zugleich ausgelöst werden
- das den Dialogprozess vorgebende **Dienstprogramm** steht für einen **Endbenutzerdienst** mit festem, definiertem Funktionsangebot
  - Kassen, Bankschalter, Auskunft-/Buchungssysteme, …
  - allgemein: **Transaktionssysteme**
- **Problem**:
  - Antwortverhalten (weiche/feste Echtzeit), Durchsatz

Teilhabersystem

So auch ein Klient/Anbieter-System (*client/server system*), in dem **Dienstnehmer** (*service user*) mit einem **Dienstgeber** (*service provider*) interagieren.

Gliederung

Echtzeitbetrieb

Prozesssteuerung

Dialog mit Echtzeitprozessen

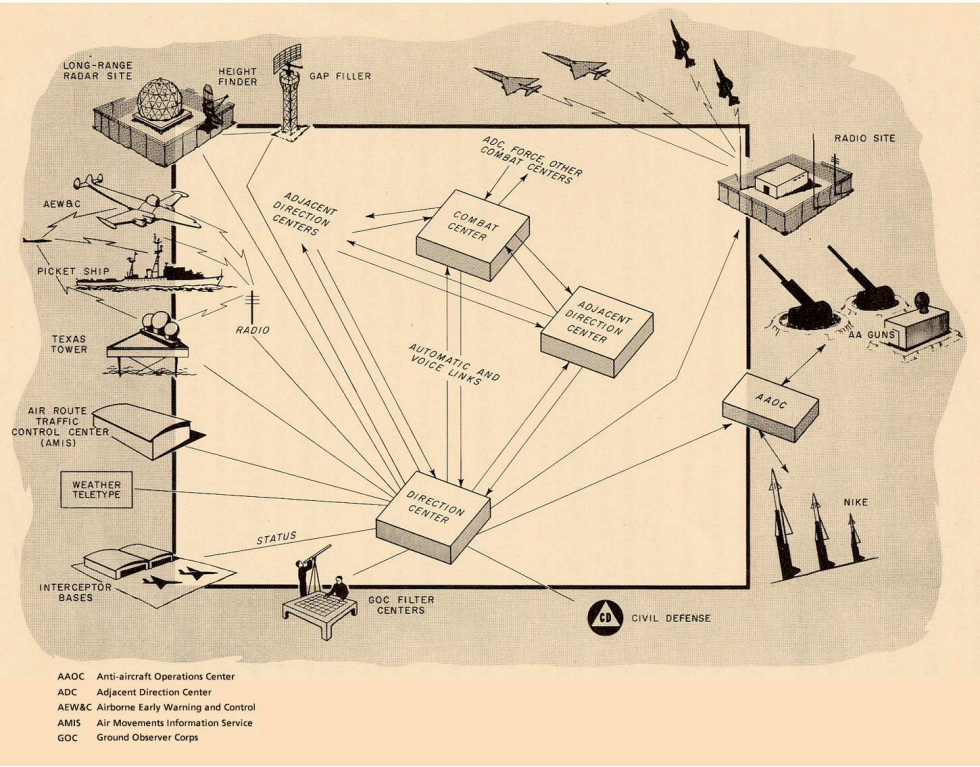
Definition (Echtzeitbetrieb, in Anlehnung an DIN 44300 [4])

[...] Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten werden nach einer zeitlich zufälligen Verteilung (ereignisgesteuert) oder zu vorbestimmten Zeitpunkten (zeitgesteuert) verwendet.

- kennzeichnend ist, dass die Zustandsänderung von Prozessen durch eine Funktion der **realen Zeit** [14] definiert ist
  - das korrekte Verhalten des Systems hängt nicht nur von den logischen Ergebnissen von Berechnungen ab
  - zusätzlicher Aspekt ist der **physikalische Zeitpunkt** der Erzeugung und Verwendung der Berechnungsergebnisse
- **interne Prozesse** des Rechensystems müssen **externe Prozesse** der phyikalischen Umgebung des Rechensystems steuern/überwachen

Bahnbrecher und Wegbereiter II

- **SAGE** (*semi-automatic ground environment*, 1958–1983)
  - Whirlwind (MIT, 1951), AN/FSQ-7 (Whirlwind II, IBM, 1957)
- erstes Echtzeitrechensystem — eine Schöpfung des „Kalten Krieges“:



- 27 Installationen über die USA verteilt
  - *Nonstop*-Betrieb
  - 25 Jahre
- durch Datenfernleitungen miteinander gekoppelt
  - Telefonleitungen
  - Vorläufer des Internets
- pro Installation:
  - 100 Konsolen
  - 500 KLOC Assembler

Echtzeitfähigkeit bedeutet Rechtzeitigkeit

- im Vordergrund steht die **zuverlässige Reaktion** des Rechensystems auf Ereignisse in der Umgebung des Rechensytsems
  - interne Prozesse erhalten jeweils einen **Termin** (*deadline*) vorgegeben, bis zu dem ein Berechnungsergebnis abzuliefern ist
  - die **Terminvorgaben** sind weich (*soft*), fest (*firm*) oder hart (*hard*), sie sind durch die externen Prozessen bestimmt
- Geschwindigkeit liefert keine Garantie, um rechtzeitig Ergebnisse von Berechnungen abliefern und Reaktionen darauf auslösen zu können
  - die im Rechensystem verwendete Zeitskala muss mit der durch die Umgebung vorgegebenen identisch sein
  - **„Zeit“ ist keine intrinsische Eigenschaft des Rechensystems**

Determiniertheit und Determinismus

Einerseits sind bei ein und derselben Eingabe verschiedene Abläufe zulässig, die dann jedoch stets das gleiche Resultat liefern müssen. Andererseits muss zu jedem Zeitpunkt im Rechensystem bestimmt sein, wie weitergefahren wird.

Echtzeitbetrieb

Echtzeitbedingungen

Terminvorgaben

- externe (physikalische) Prozesse definieren, was genau bei einer nicht termingerecht geleisteten Berechnung zu geschehen hat:
  weich (*soft*) auch „schwach“
    - das Ergebnis ist weiterhin von Nutzen, verliert jedoch mit jedem weiteren Zeitverzug des internen Prozesses zunehmend an Wert
    - die Terminverletzung ist tolerierbar

  fest (*firm*) auch „stark“
    - das Ergebnis ist wertlos, wird verworfen, der interne Prozess wird abgebrochen und erneut bereitgestellt
    - die Terminverletzung ist tolerierbar

  hart (*hard*) auch „strikt“
    - Verspätung der Ergebnislieferung kann zur „Katastrophe“ führen, dem int. Prozess wird eine **Ausnahme** zugestellt
    - Terminverletzung ist nicht tolerierbar — aber möglich…

- **Problem**:
  - Termineinhaltung unter allen Last- und Fehlerbedingungen

Terminvorgaben: fest $\Longleftrightarrow$ hart

- eine Terminverletzung bedeutet grundsätzlich eine Ausnahme, deren Behandlung jedoch auf verschiedenen Ebenen erfolgt
  - im Betriebssystem (fest) oder im Maschinenprogramm (hart)
  - im „harten Fall“ also im Anwendungsprogramm des späten Prozesses
- das Betriebssystem erkennt die Verletzung, die Anwendung muss aber weiterarbeiten (fest) | den sicheren Zustand finden (hart)
  - das Betriebssystem bricht die Berechnung ab
  - die nächste Berechnung wird gestartet
  - transparent für die Anwendung
  - das Betriebssystem löst eine Ausnahmesituation aus
  - die Ausnahmebehandlung führt zum sicheren Zustand
  - **intransparent** für die Anwendung

Terminverletzung

Auch wenn der Ablaufplan von Prozessen und das Betriebssystem in Theorie „am Reißbrett“ deterministisch sind, kann in Praxis das Rechensystem Störeinflüssen unterworfen sein und so Termine verpassen.

Gliederung

Systemmerkmale

Multiprozessoren

Symmetrische Simultanverarbeitung

Definition (SMP, *symmetric multiprocessing*)

Zwei oder mehr gleiche (identische) Prozessoren, eng gekoppelt über ein gemeinsames Verbindungssystem.

- erfordert ganz bestimmte **architektonische Merkmale** sowohl von der Befehlssatz- als auch von der Maschinenprogrammebene
  - jeder Prozessor hat gleichberechtigten Zugriff auf den Hauptspeicher (*shared-memory acccess*) und die Peripherie
  - der Zugriff auf den Hauptspeicher ist für alle Prozessoren gleichförmig (*uniform memory access*, UMA)
    - bedingt in nichtfunktionaler Hinsicht, sofern nämlich die **Zyklenanzahl pro Speicherzugriff** betrachtet wird
    - unbedingt aber im funktionalen Sinn bezogen auf die Maschinenbefehle
- die Prozessoren stellen ein **homogenes System** dar und sie werden von demselben Betriebssystem verwaltet
- **Problem**:
  - Synchronisation, Skalierbarkeit

Speichergekoppelter Multiprozessor

Definition (SMP, *shared-memory processor*)

Ein Parellelrechnersystem, in dem alle Prozessoren den Hauptspeicher mitbenutzen, ohne jedoch einen gleichberechtigten/-förmigen Zugriff darauf haben zu müssen.

- **architektonische Merkmale** der Befehlssatzebene geben bestimmte Freiheitsgrade für die Simultanverarbeitung vor
  - **asymmetrisch**
    - hardwarebedingter, zwingender asym. Betrieb
    - Programme sind ggf. prozessorgebunden
    - ↪ *asymmetric multiprocessing*
  - **symmetrisch**
    - **anwendungsorientierter Betrieb** wird ermöglicht
    - das Betriebssystem legt die Multiprozessorbetriebsart fest
    - *symmetric/asymmetric multiprocessing*
- die Maschinenprogrammebene kann ein **heterogenes System** bilden, in funktionaler und nichtfunktionaler Hinsicht
- **Problem**:
  - Synchronisation, Skalierbarkeit, Anpassbarkeit

Parallelverarbeitung

parallel processing

- das Multiprozessorsystem kann…
  - N verschiedene oder identische Programme,
  - N Fäden dieser Programme oder
  - N Fäden ein und desselben Programms

  …**echt parallel** ausführen
- jeder Prozessor kann…
  - M verschiedene oder identische Programme,
  - M Fäden dieser Programme oder
  - M Fäden ein und desselben Programms

  …**pseudo/quasi parallel** im Multiplexbetrieb ausführen
- $N \times M$ Ausführungsstränge können **nebenläufig** stattfinden

Synchronisation

Die Art und Weise der Koordination der Kooperation und Konkurrenz gleichzeitiger Prozesse ist nur bedingt davon abhängig, ob der Betrieb des Rechensystems echt oder pseudo/quasi parallel geschieht.

Systemmerkmale

Schutzvorkehrungen

Sicherheit

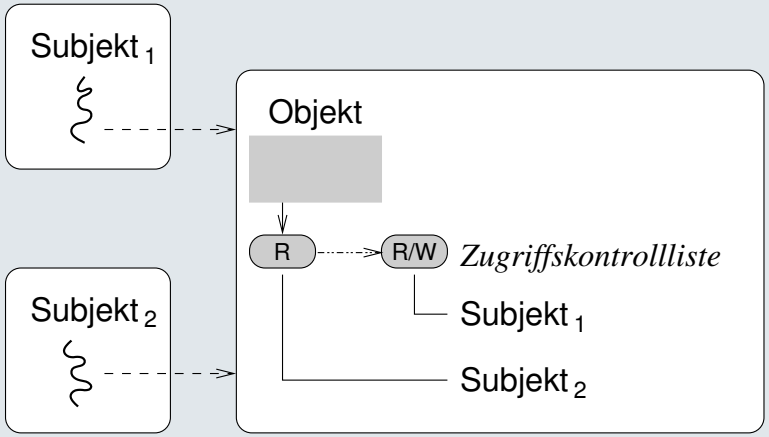
security

- **Schutz** (*protection*) vor unautorisierten Zugriffen durch Prozesse, der in Körnigkeit und Funktion sehr unterschiedlich ausgelegt sein kann:
  i jeden Prozessadressraum in **Isolation** betreiben
    - Schutz durch Eingrenzung oder Segmentierung
    - Zugriffsfehler führen zum Abbruch der Programmausführung
    - i.A. keine selektive Zugriffskontrolle möglich und sehr grobkörnig
  ii Prozessen eine **Befähigung** (*capability* [3, 29, 6]) zum Zugriff erteilen
    - den verschiedenen **Subjekten** (Prozesse) individuelle Zugriffsrechte geben, z.B., ausführen, lesen, schreiben oder ändern dürfen
    - und zwar auf dasselbe von ihnen mitbenutzte **Objekt** (Datum, Datei, Gerät, Prozedur, Prozess)
  iii Objekten eine **Zugriffskontrollliste** (*access control list*, ACL [26]) geben
    - ein Listeneintrag legt das Zugriffsrecht eines Subjekts auf das Objekt fest
    - vereinfacht auch in „Besitzer/in-Gruppe-Welt“-Form (*user/group/world*)
- **Problem**:
  - verdeckter Kanal (*covered channel*) bzw. Seitenkanal

Gegenüberstellung

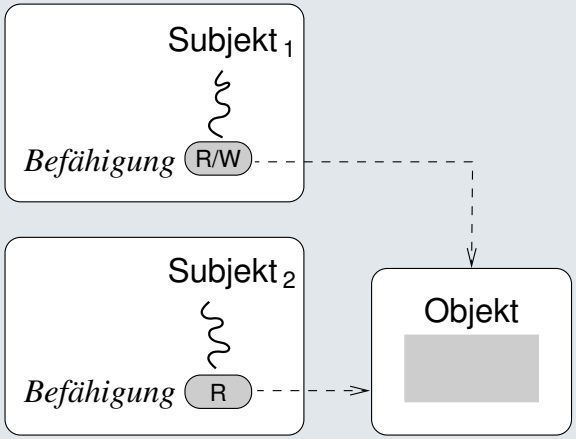
- feinkörniger Schutz durch **selektive Autorisierung** der Zugriffe:

Zugriffskontrollliste



- Rechtevergabe einfach (lokal)
- Rechterücknahme: einfach (lokal)
- Rechteüberprüfung: aufwendig (Suche)
- dito Subjektrechtebestimmung (entfernt)
- Objektsicht-Rechtebestimmung: einfach
- Kontrollinformation: zentral gespeichert

Befähigungen



- aufwendig (entfernt)
- aufwendig (entfernt)
- einfach (lokal)
- einfach (Zugriff)
- aufwendig (Sammelruf)
- dezentral gespeichert

Methodologie: Zugriffsmatrix

access matrix [15, 9]

- in diesem allgemeinen Modell spezifiziert jeder Eintrag in der Matrix das individuelle Zugriffsrecht eines Subjekts auf ein Objekt:

Subjekte	Objekte		
	Cyan	Grau	Blau
1	R/X	R/W	–
2	–	R	–

- ***read*** R
- ***write*** W
- ***execute*** X

- je nach **Abspeicherung** und Verwendung der in der Matrix kodierten Information ergeben sich verschiedene Implementierungsoptionen:

Totalsicht
- in Form einer systembezogenen **Zugriffstabelle** ☹
- ineffizient, wegen der i.d.R. dünn besetzten Matrix

Objektsicht
- in Form einer objektbezogenen **Zugriffskontrollliste** ☺
- spaltenweise Speicherung der Zugriffsmatrix

Subjektsicht
- in Form subjektbezogener **Befähigungen** ☺
- zeilenweise Speicherung der Zugriffsmatrix

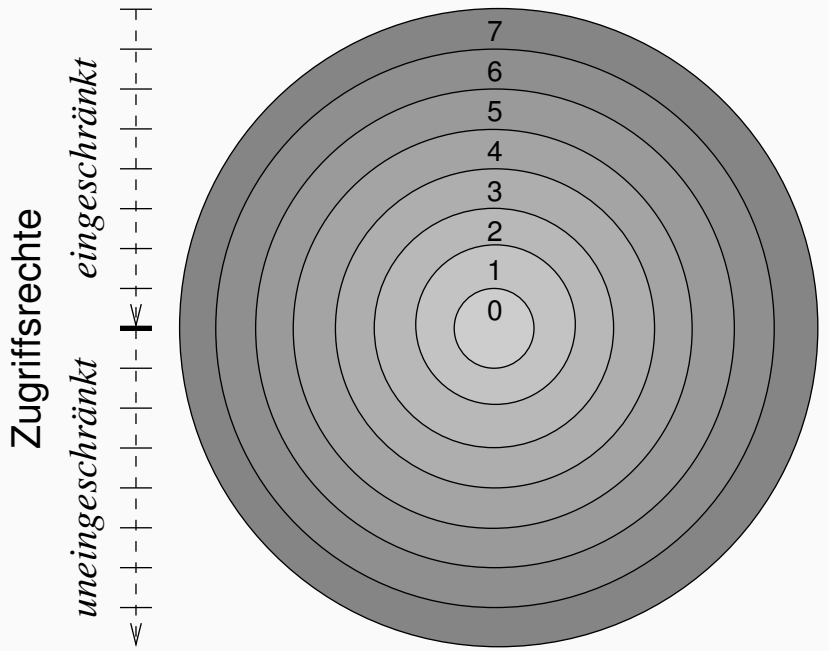
Bahnbrecher und Wegbereiter III

- **Multics** (*Multiplexed Information and Computing Service* [21], 1965)
  - setzt den Maßstab in Bezug auf Adressraum-/Speicherverwaltung:
    1. *jede* im System gespeicherte abrufbare Information ist direkt von einem Prozessor adressierbar und jeder Berechnung referenzierbar
    2. *jede* Referenzierung unterliegt einer durch **Hardwareschutzringe** implementierten mehrstufigen Zugriffskontrolle [27, 25]
  - **ringgeschützte seitennummerierte Segmentierung** (*ring-protected paged segmentation*)
    - das ursprüngliche Konzept (für den GE 645) sah 64 Ringe vor, letztendlich bot die Hardware (Honeywell 6180) Unterstützung für acht Ringe
    - nicht in Hardware implementierte Ringe wurden durch Software emuliert
  - eng mit dem Segmentkonzept verbunden war **dynamisches Binden**
    - jede Art von Information, ob Programmtext oder -daten, war ein Segment
    - Segmente konnten bei Bedarf (*on demand*) geladen werden
    - Zugriff auf ungeladenes Segment bedeutete **Bindungsfehler** (*linkage fault*)
    - in Folge machte eine **Bindelader** (*linking loader*) das Segment verfügbar
- **Problem**:
  - Hardwareunterstützung

Schutzringe: Multics

Honeywell 6180

- Verwendung der Schutzringe:
  - **0–3** Betriebssystem
    - **0–1** Hauptsteuerprogramm
    - **2–3** Dienstprogramme
  - **4–7** Anwendungssystem
    - **4–5** Benutzerprogramme
    - **6–7** Subsysteme
- Ringwechsel, Zugriffe
  - kontrolliert durch die Hardware



- je nach Prozessattribut/-aktion sind **Ringfehler** (*ring fault*) möglich
  - Folge ist die Teilinterpretation der Operation auf Ring 0 (*supervisor*)
  - unautorisierte Operationen führen zum **Schutzfehler** (*protection fault*)
- **Problem**:
  - Schichtenstruktur, Ringzuordnung: **funktionale Hierarchie** [11]

Systemmerkmale

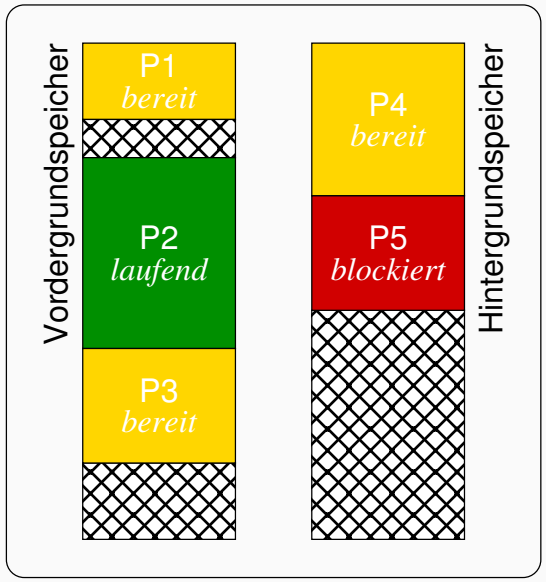
Speicherverwaltung

Grad an Mehrprogrammbetrieb

- die selektive Überlagerung des Hauptspeichers durch programmiertes dynamisches Laden (*overlay*) hat seine Grenzen
  - Anzahl $\times$ Größe hauptspeicherresidenter Text-/Datenbereiche begrenzt die Anzahl der gleichzeitig zur Ausführung vorgehaltenen Programme
  - variabler Wert, abhängig von Struktur/Organisation der Programme und den Fähigkeiten der Programmierer/innen
- **Umlagerung** der Speicherbereiche aktuell nicht ausführbereiter Prog. (*swapping*) verschiebt die Grenze nach hinten
  - schafft Platz für ein oder mehrere andere (zusätzliche) Programme
  - lässt mehr Programme zu, als insgesamt in den Hauptsspeicher passt
- Berücksichtigung solcher Bereiche der sich in Ausführung befindlichen Programme (*paging*, *segmentation*) gibt Spielraum [2]
  - im Unterschied zu vorher werden nur Teile eines Programms umgelagert
  - Programme liegen nur scheinbar („virtuell“) komplett im Hauptspeicher
- Prozesse belegen **Arbeitsspeicher**, nämlich den zu einem bestimmten Zeitpunkt beanspruchten Verbund von **Haupt- und Ablagespeicher**

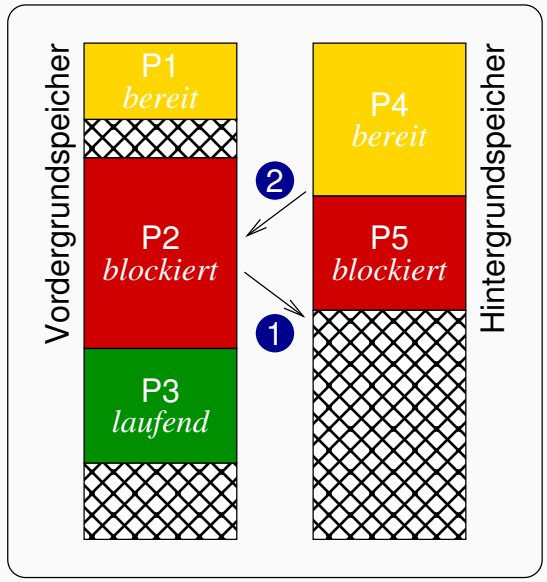
Umlagerung nicht ausführbereiter Programme

- Funktion der mittelfristigen Einplanung (*medium-term scheduling*)



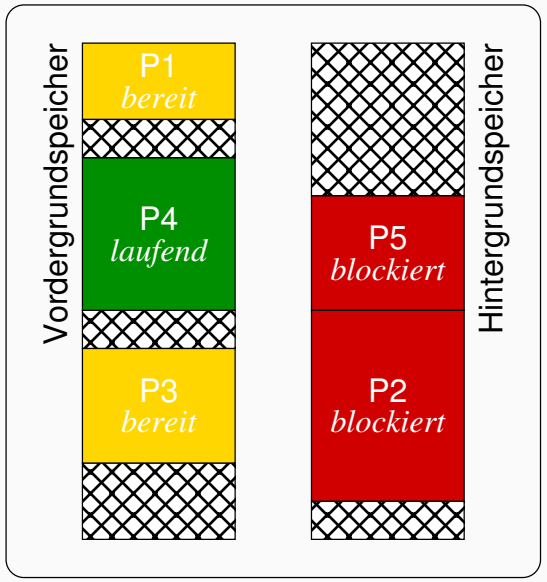
Ausgangssituation:

- P[1-3] im RAM
- P2 belegt die CPU



Umlagerung:

1. P2 *swap out*
2. P4 *swap in*



Resultat:

- P[134] im RAM
- P4 belegt die CPU

- **Problem**:
  - Fragmentierung, Verdichtung, Körnigkeit

Umlagerung laufender Programme

- Prozesse schreiten voran, obwohl die sie kontrollierenden Programme nicht komplett im Hauptspeicher vorliegen: **virtueller Speicher** [10]
  - die von einem Prozess zu einem Zeitpunkt scheinbar nicht benötigten Programmteile liegen im Hintergrund, im Ablagespeicher
    - sie werden erst bei Bedarf (*on demand*) nachgeladen
    - ggf. sind als Folge andere Programmteile vorher zu verdrängen
  - Zugriffe auf ausgelagerte Programmteile unterbrechen die Prozesse und werden durch **partielle Interpretation** ausgeführt
    - logisch bleibt der unterbrochene Prozess weiter in Ausführung
    - physisch wird er jedoch im Zuge der Einlagerung (E/A) blockieren
  - Aus- und Einlagerung wechseln sich mehr oder wenig intensiv ab
- **Problem**:
  - Lade- und Ersetzungsstrategien, Arbeitsmenge (*working set*)

Hauptspeicherüberbuchung und -überbelegung

Der Platzbedarf der scheinbar (virtuell) komplett im Hauptspeicher liegenden und laufenden Programme kann die Größe des wirklichen (realen) Hauptspeichers weit überschreiten.

Granularität der Umlagerungseinheiten

- Programmteile, die ein-, aus- und/oder überlagert werden können, sind **Seiten** oder **Segmente**:

  Seitennummerierung (*paging*) Atlas [7]
  - Einheiten (von Bytes) fester Größe
  - **Problem**: interne Fragmentierung $\rightsquigarrow$ „*false positive*“ (Adresse)

  Segmentierung (*segmentation*) B 5000 [20]
  - Einheiten (von Bytes) variabler Größe
  - **Problem**: externe Fragmentierung $\rightsquigarrow$ „*false negative*“ (Bruchstücke)

  seitennummerierte Segmentierung (*paged segmentation*)[3] GE 635 [8]
  - Kombination: Segmente aber in Seiten untergliedern
  - **Problem**: interne Fragmentierung (wegen Seitennummerierung)
- sie werden abgebildet auf gleich große Einheiten des Hauptspeichers (eingelagert) oder Ablagespeichers (ausgelagert)
- **Problem**:
  - Fragmentierung (des Arbeitsspeichers) $\rightsquigarrow$ Verschnitt

[3] Beachte: nicht „segmentierte Seitenadressierung“ (*segmented paging*)!

Automatische Überlagerung

- seiten- und/oder segmentbasierte Umlagerung zeigt Ähnlichkeiten zur **Überlagerungstechnik** (*overlay*), jedoch:
  - die Seiten-/Segmentanforderungen sind nicht im Maschinenprogramm zu finden, stattdessen im Betriebssystem (*pager, segment handler*)
    - die Anforderungen stellt stellvertretend ein Systemprogramm
    - Ladeanweisungen sind so vor dem Maschinenprogramm verborgen
  - Zugriffe auf ausgelagerte Seiten/Segmente fängt die Befehlssatzebene ab, die sie dann ans Betriebssystem weiterleitet (*trap*)
    - das Maschinenprogramm wird von CPU bzw. MMU unterbrochen
    - der gescheiterte Zugriff wird vom Betriebssystem partiell interpretiert
- des Weiteren fällt die **Wiederholung** des unterbrochenen Befehls an, die vom Betriebssystem zu veranlassen ist
  - der Speicherzugriff scheiterte beim Befehls- oder Operandenabruf
  - die CPU konnte die Operation noch nicht vollständig ausführen (*rerun*)
- **Problem**:
  - Komplexität, Determiniertheit

Systemmerkmale

Universalität

Universalbetriebssystem (*general purpose operating system*)

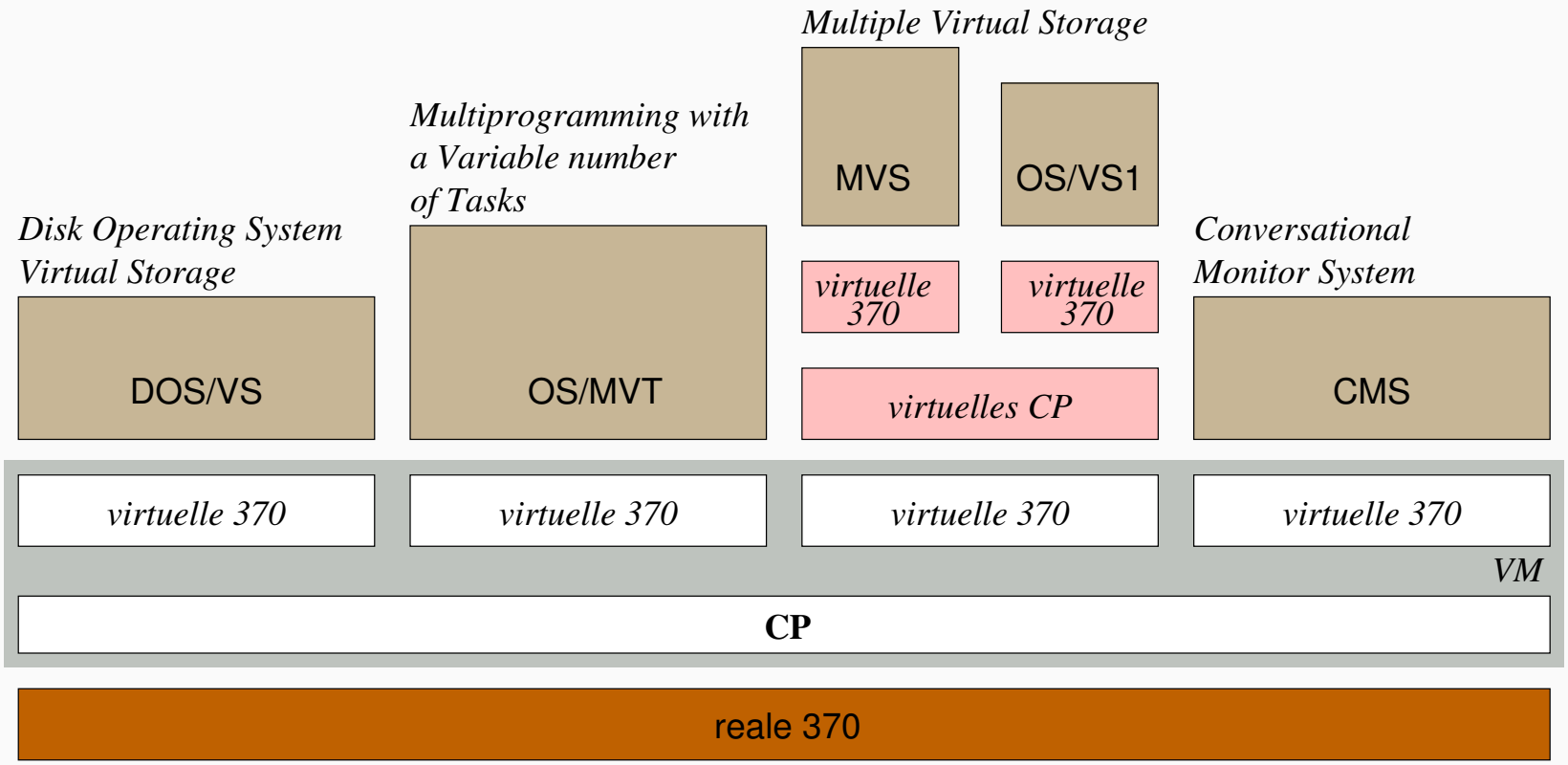
- bei einem Betriebssystem handelt es sich um Software, die zwischen Baum und Borke steckt, womit sich ein **Dilemma** ergibt

Lister, „Fundamentals of Operating Systems“ [19]

- *Clearly, the operating system design must be strongly influenced by the type of use for which the machine is intended.*
- *Unfortunately it is often the case with ‘general purpose machines’ that the type of use cannot easily be identified;*
- *a common criticism of many systems is that, in attempting to be all things to all individuals, they end up being totally satisfactory to no-one.*

- ein **Allzweckbetriebssystem** ist geprägt von Kompromissen, die sich quer durch die Implementierung ziehen
  - damit Echtzeitbetrieb aber ausschließen, der kompromisslos sein muss!
- Ansätze für verbesserte Akzeptanz sind Virtualisierung einerseits und „Konzentration auf das Wesentliche“ andererseits
  - auch damit bleibt ein Betriebssystem **domänenspezifische Software**

- Spezialisierung durch virtuelle Maschinen:



CP

- Abk. für *control program*: **Hypervisor**, VMM
- **Selbstvirtualisierung** (para/voll, [13, S. 30]) des realen System/370

Konzentration auf das Wesentliche

- UNIX [23, 18, 17]
  - ein Betriebssystemkern von 10^4 Zeilen C und nicht 10^6 Zeilen PL/I

Multics		**UNICS**
Multiplexed	$\Longleftrightarrow$	Uniplexed
Information and Computing Service		

- ITS nicht zu vergessen (S. 12)

„Lotta hat einen Unixtag“, Astrid Lindgren [16, S. 81–89]

Die drei Jahre alte Lotta ist die kleine Schwester der Erzählerin. Läuft am Tag vieles schief bei ihr, sagt sie „Unixtag“, meint aber „Unglückstag“.

UNIX $\overset{?}{\mapsto}$ Unglück $\overset{?}{\mapsto}$ macOS/Linux

Vom ursprünglichen Ansatz eines nur wesentliche Dinge enthaltenes, schlankes Betriebssystem ist heute wenig zu spüren.

Gliederung

Stand der Kunst[4]

Viel Getöse für altbackene Technik?

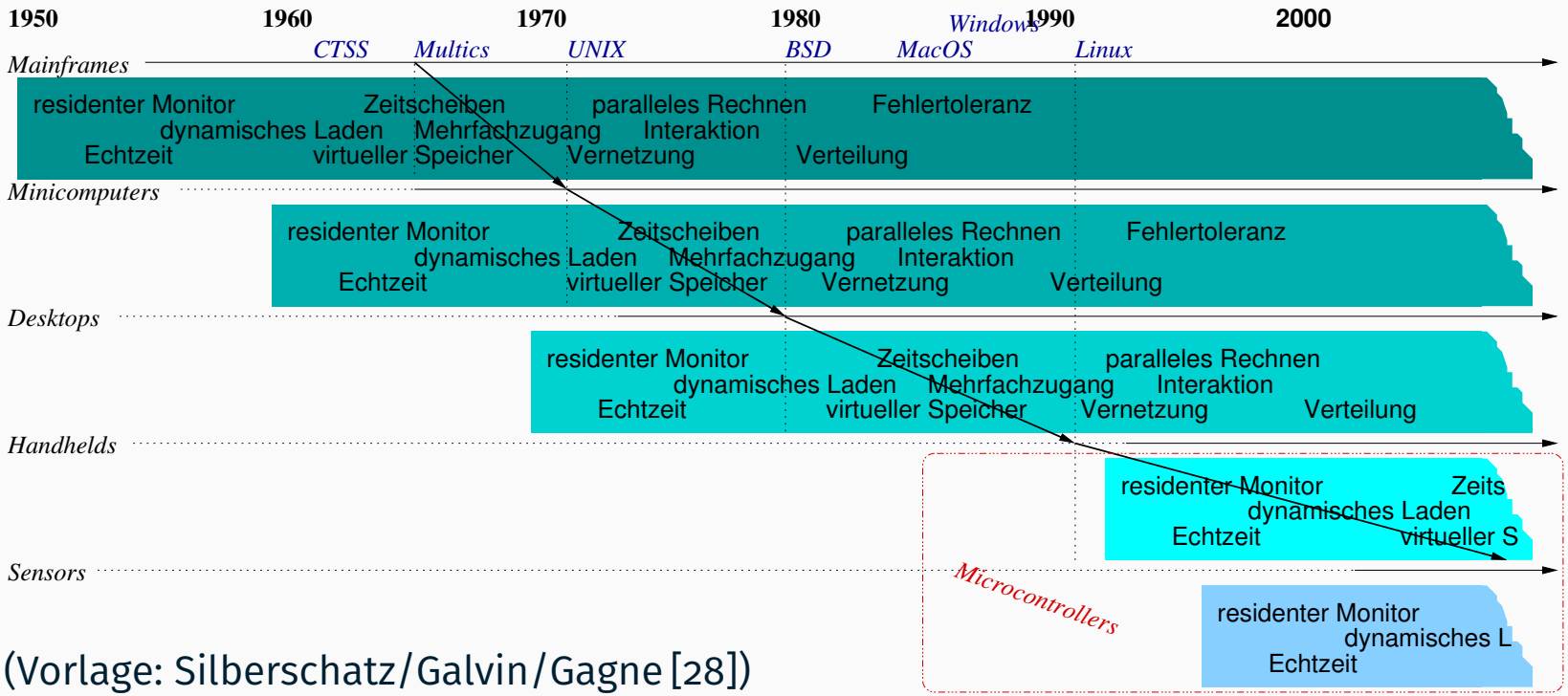
- **Linux** „*yet another UNIX-like operating system*“, aber was soll's…
  - Entwicklungsprozess und -modell sind der eigentliche „Kick“ 2 %
  - 70er-Jahre Technologie — ohne Multics (funktional) zu erreichen
- **Windows** „*new technology*“, wirklich?
  - vor WNT entwickelte Cuttler VMS (DEC): $WNT = VMS + 1$
  - nach wie vor Marktführer im PC-Sektor 77 %
- **macOS**, ein vergleichsweise echter Fortschritt?
  - solides UNIX (Free**BSD**) auf solider Mikrokernbasis (**Mach**)
  - Apple bringt PC-Technologie erneut voran 18 %

„Des Kaisers neue Kleider“

Funktionsumfang wie auch Repräsentation vermeintlich moderner Betriebssysteme lässt den Schluss zu, dass so einige Male das Rad neu erfunden wurde.

[4] Weltweiter Marktanteil, Statista 2020

Migration von Betriebssystemkonzepten



(Vorlage: Silberschatz/Galvin/Gagne [28])

- Fähigkeit zur „Wanderung“ zu anderen, kleineren Gefilden fällt nicht vom Himmel, sondern bedarf sorgfältiger **Konzeptumsetzung**
- Voraussetzung dafür ist eine **Domänenanalyse**, um gemeinsame und variable Konzeptanteile zu identifizieren

Resümee

…die eierlegende Wollmilchsau gibt es nicht!

- **Mehrzugangsbetrieb** ermöglicht Arbeit und Umgang mit einem Rechensystem über mehrere Dialogstationen
  - im Teilnehmerbetrieb setzen Dialogstationen eigene Dialogprozesse ab
  - im Teilhaberbetrieb teilen sich Dialogstationen einen Dialogprozess
- **Echtzeitbetrieb** muss kompromisslos sein, da das Zeitverhalten des Rechensystems sonst unvorhersehbar ist
  - Zustandsänderung von Programmen wird zur Funktion der realen Zeit
    - „Zeit“ ist keine intrinsische Eigenschaft des Rechensystems mehr
  - „externe Prozesse“ definieren Terminvorgaben, die einzuhalten sind
    - die Echtzeitbedingungen dabei gelten als weich, fest oder hart
- wichtige **Systemmerkmale** insbesondere für Mehrzugangsbetrieb:
  - Parallelverarbeitung durch (speichergekoppelte) Multiprozessoren
  - über bloße Adressraumisolation hinausgehende Schutzvorkehrungen
  - auf Programm(teil)umlagerung ausgerichtete Speicherverwaltung
- **Allzweckbetriebssysteme** sind universal, indem sie Fähigkeiten für die verschiedensten Bereiche umfassen — aber nicht für alle…

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[1] Corbató, F. J. ; Merwin-Daggett, M. ; Dalex, R. C.:
An Experimental Time-Sharing System.
In: *Proceedings of the AIEE-IRE '62 Spring Joint Computer Conference*, ACM, 1962, S. 335–344

[2] Denning, P. J.:
Virtual Memory.
In: *Computing Surveys* 2 (1970), Sept., Nr. 3, S. 153–189

[3] Dennis, J. B. ; Horn, E. C. V.:
Programming Semantics for Multiprogrammed Computations.
In: *Communications of the ACM* 9 (1966), März, Nr. 3, S. 143–155

[4] Deutsches Institut für Normung:
Informationsverarbeitung — Begriffe.
Berlin, Köln, 1985 (DIN 44300)

Literaturverzeichnis (2)

[5] Eastlake, D. E. ; Greenblatt, R. D. ; Holloway, J. T. ; Knight, T. F. ; Nelson, S. :
ITS 1.5 Reference Manual / MIT.
Cambridge, MA, USA, Jul. 1969 (AIM-161A). – Forschungsbericht

[6] Fabry, R. S.:
Capability-Based Addressing.
In: *Communications of the ACM* 17 (1974), Jul., Nr. 7, S. 403–412

[7] Fotheringham, J. :
Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store.
In: *Communications of the ACM* 4 (1961), Okt., Nr. 10, S. 435–436

Literaturverzeichnis (3)

[8] General Electric Company (Hrsg.):
GE-625/635 Programming Reference Manual.
CPB-1004A.
Phoenix, AZ, USA: General Electric Company, Jul. 1964

[9] Graham, G. S. ; Denning, P. J.:
Protection—Principles and Practice.
In: *1972 Proceedings of the Spring Joint Computer Conference, May 6–8, 1972, Atlantic City, USA* American Federation of Information Processing Societies, AFIPS Press, 1972, S. 417–429

[10] Güntsch, F.-R. :
***Logischer Entwurf eines digitalen Rechengeräts mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb*, Technische Universität Berlin, Diss., März 1957**

Literaturverzeichnis (4)

[11] Habermann, A. N. ; Flon, L. ; Cooprider, L. W.:
Modularization and Hierarchy in a Family of Operating Systems.
In: *Communications of the ACM* 19 (1976), Mai, Nr. 5, S. 266–272

[12] Kernighan, B. W.:
UNIX: A History and a Memoir.
Kindle Direct Publishing, 2020. –
ISBN 978–169597855–3

[13] Kleinöder, J. ; Schröder-Preikschat, W. :
Virtuelle Maschinen.
In: Lehrstuhl Informatik 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.1

Literaturverzeichnis (5)

[14] Kopetz, H. :
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997. –
ISBN 0–7923–9894–7

[15] Lampson, B. W.:
Protection.
In: *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems.*
New Jersey, USA : Department of Electrical Engineering, Princeton University, März 1971, S. 437–443

Literaturverzeichnis (6)

[16] *Kapitel* Lotta hat einen Unixtag.

In: Lindgren, A. :
Die Kinder aus der Krachmacherstraße.
Oettinger-Verlag, 1957. –
ISBN 3–7891–4118–6, S. 81–89

[17] Lions, J. :

A Commentary on the Sixth Edition UNIX Operating System.
The University of New South Wales, Department of Computer Science, Australia :
`http://www.lemis.com/grog/Documentation/Lions`, 1977

[18] Lions, J. :

UNIX Operating System Source Code, Level Six.
The University of New South Wales, Department of Computer Science, Australia : `http://v6.cuzuco.com`, Jun. 1977

Literaturverzeichnis (7)

[19] Lister, A. M. ; Eager, R. D.:
Fundamentals of Operating Systems.
The Macmillan Press Ltd., 1993. –
ISBN 0–333–59848–2

[20] Mayer, A. J. W.:
The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?
In: *ACM SIGARCH Computer Architecture News* 10 (1982), Jun., Nr. 4, S. 3–10

[21] Organick, E. I.:
The Multics System: An Examination of its Structure.
MIT Press, 1972. –
ISBN 0–262–15012–3

Literaturverzeichnis (8)

[22] Pouzon, L. :
The Origin of the Shell.
In: *Multics Home*.
Multicians, 2000, Kapitel `http://www.multicians.org/shell.html`

[23] Ritchie, D. M. ; Thompson, K. :
The UNIX Time-Sharing System.
In: *Communications of the ACM* 17 (1974), Jul., Nr. 7, S. 365–374

[24] Roberts, L. G.:
Multiple Computer Networks and Intercomputer Communication.
In: Gosden, J. (Hrsg.) ; Randell, B. (Hrsg.): *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67), October 1–4, 1967, Gatlinburg, TN, USA*, ACM, 1967, S. 3.1–3.6

Literaturverzeichnis (9)

[25] Saltzer, J. H.:
Protection and the Control of Information Sharing in Multics.
In: *Communications of the ACM* 17 (1974), Jul., Nr. 7, S. 388–402

[26] Saltzer, J. H. ; Schroeder, M. D.:
The Protection of Information in Computer Systems.
In: *Proceedings of the IEEE* 63 (1975), Sept., Nr. 9, S. 1278–1308

[27] Schroeder, M. D. ; Saltzer, J. H.:
A Hardware Architecture for Implementing Protection Rings.
In: *Proceedings of the Third ACM Symposium on Operating System Principles (SOSP 1971), October 18–20, 1971, Palo Alto, California, USA*, ACM, 1971, S. 42–54

Literaturverzeichnis (10)

[28] Silberschatz, A. ; Galvin, P. B. ; Gagne, G. :
Operating System Concepts.
John Wiley & Sons, Inc., 2001. –
ISBN 0–471–41743–2

[29] Wulf, W. A. ; Cohen, E. S. ; Corwin, W. M. ; Jones, A. K. ; Levin, R. ; Pierson, C. ; Pollack, F. J.:
HYDRA: The Kernel of a Multiprocessor Operating System.
In: *Communications of the ACM* 17 (1974), Jun., Nr. 6, S. 337–345

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil B – VIII. Zwischenbilanz

18. Juli 2023

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

SP1

- Lehrziele
- C
- UNIX
- Einleitung
- Rechnerorganisation
- Betriebssystemkonzepte
- Betriebsarten

SP2

- Ausblick

Gliederung

SP1

Lehrziele

Systemnahe Software

Definition (Systemprogrammierung)

Erstellen von Softwareprogrammen, die Teile eines Betriebssystems sind beziehungsweise mit einem Betriebssystem direkt interagieren oder die Hardware (genauer: Zentraleinheit[a] und Peripherie[b]) eines Rechensystems betreiben müssen.

[a] *central processing unit* (CPU), ein-/mehrfach, ein-, mehr- oder vielkernig.
[b] Geräte zur Ein-/Ausgabe oder Steuerung/Regelung „externer Prozesse“.

Auch schon mal zwischen zwei Stühlen sitzend:

- **Anwendungssoftware** („oben“) einerseits
  - ermöglichen, unterstützen, nicht entgegenwirken
- **Plattformsysteme** („unten“) andererseits
  - anwendungsspezifisch verfügbar machen
  - problemorientiert betreiben, *bedingt verbergen*
    - nachteilige Eigenschaften versuchen zu kaschieren

Quelle: `arcadja.com`, Franz Kott

SP1

C

Kurzeinführung in C

Schlüsselwörter

```
auto      break     case      char      const     continue  default   do
double    else      enum      extern    float     for       goto      if
int       long      register  return    short     signed    sizeof    static
struct    switch    typedef   union     unsigned  void      volatile  while
```

Operatoren, Selektoren, Klammerungen und andere „Satzzeichen“

```
!     "     %     &     '     (     )     *     +     ,     -     .     /
:     ;     <     =     >     ?     [     ]     ^     {     }     ~
```

- was macht dieses Programm?

```
1  #include <unistd.h>
2
3  int main() {
4      printf("%d\n", getpid());
5  }
```

- was kann man daraus machen?
  - ***buffer overflow exploit***

- was geschieht nun?

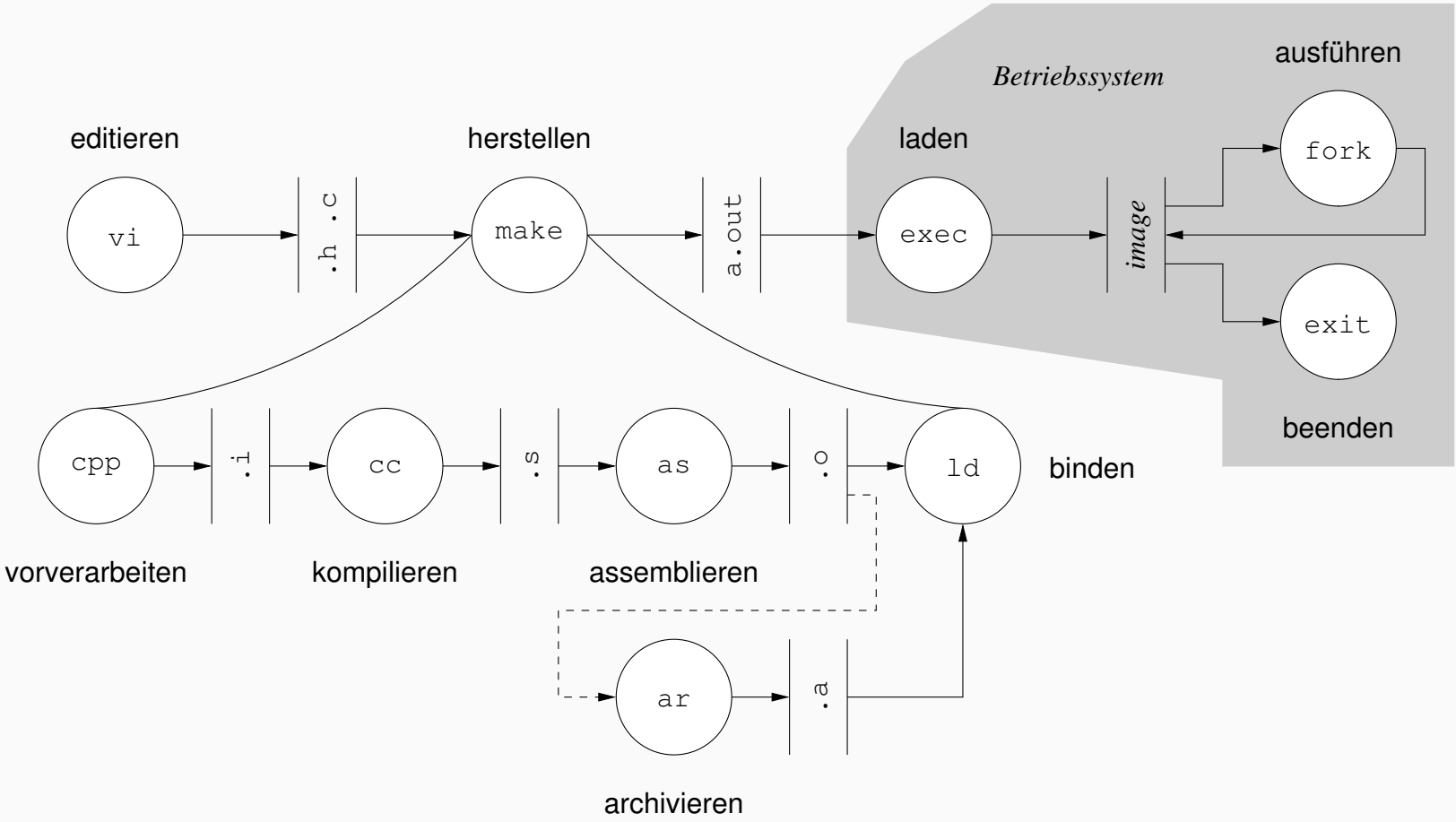
```
6  #include <stdio.h>
7  #include <string.h>
8
9  int getpid() {
10     char buffer[20];
11     gets(buffer);
12     return strlen(buffer);
13 }
```

SP1

UNIX

Quellprogramm $\mapsto$ Prozess

[2] und [3, S. 26*ff.*]



SP1

Einleitung

Ganzheitsmethode[1] [6]

Die Funktionsweise (auch) von Betriebssystemen zu verstehen, hilft bemerkenswerte Erscheinungen innerhalb eines Rechensystems zu begreifen und in ihrer Bedeutung besser einzuschätzen.

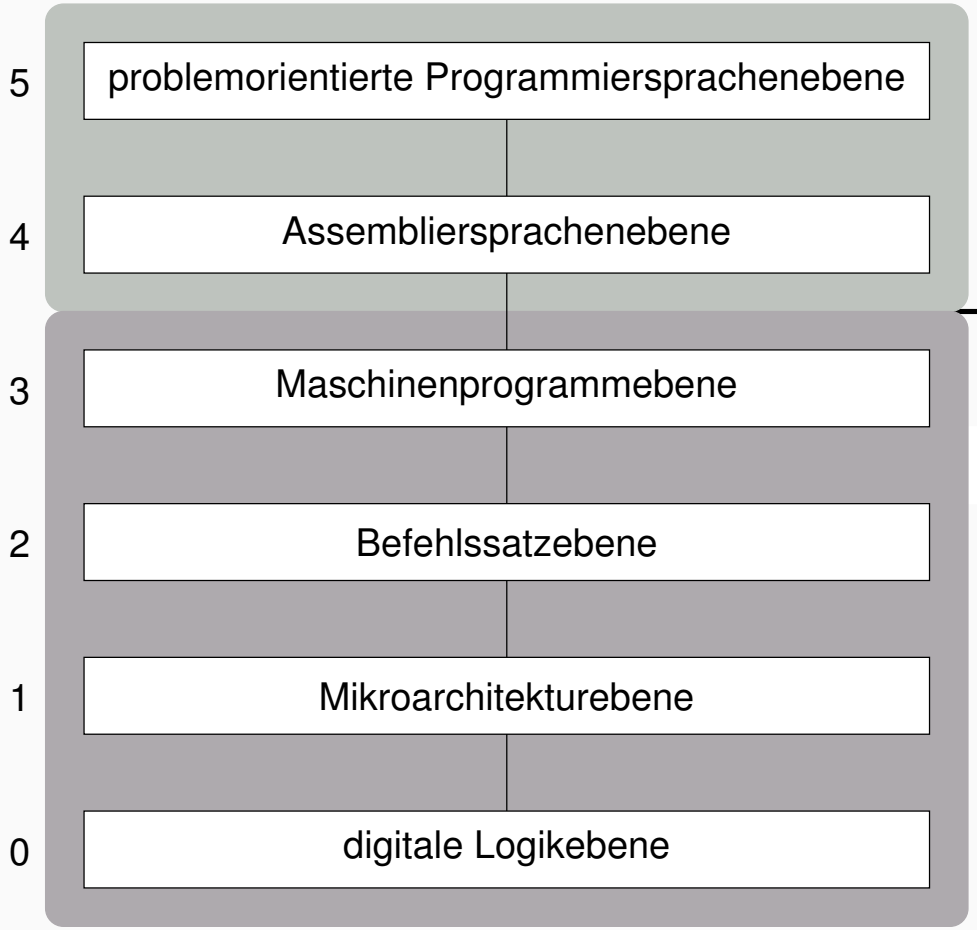
- **Eigenschaften** (*features*) von Betriebssystemen erkennen:
  - **funktionale**
    - Verwaltung der Betriebsmittel (Prozessor, Speicher, Peripherie) für eine Anwendungsdomäne
  - **nichtfunktionale**
    - dabei anfallender Zeit-, Speicher-, Energieverbrauch
    - d.h., **Gütemerkmale** einer Implementierung
- aus den funktionalen Eigenschaften resultierendes **Systemverhalten** unterscheiden von Fehlern (*bugs*) des Systems
  - um Fehler kann ggf. „herum programmiert“ werden
  - um zum Anwendungsfall unpassende Eigenschaften oft jedoch nicht

[1] Analytische Lernmethode, die die Vermittlung eines Stoffes als Gesamtheit in den Mittelpunkt stellt, um dann konstituierende Elemente weiter zu untersuchen.

SP1

Rechnerorganisation

Schichtenfolge in Rechensystemen



Übersetzung

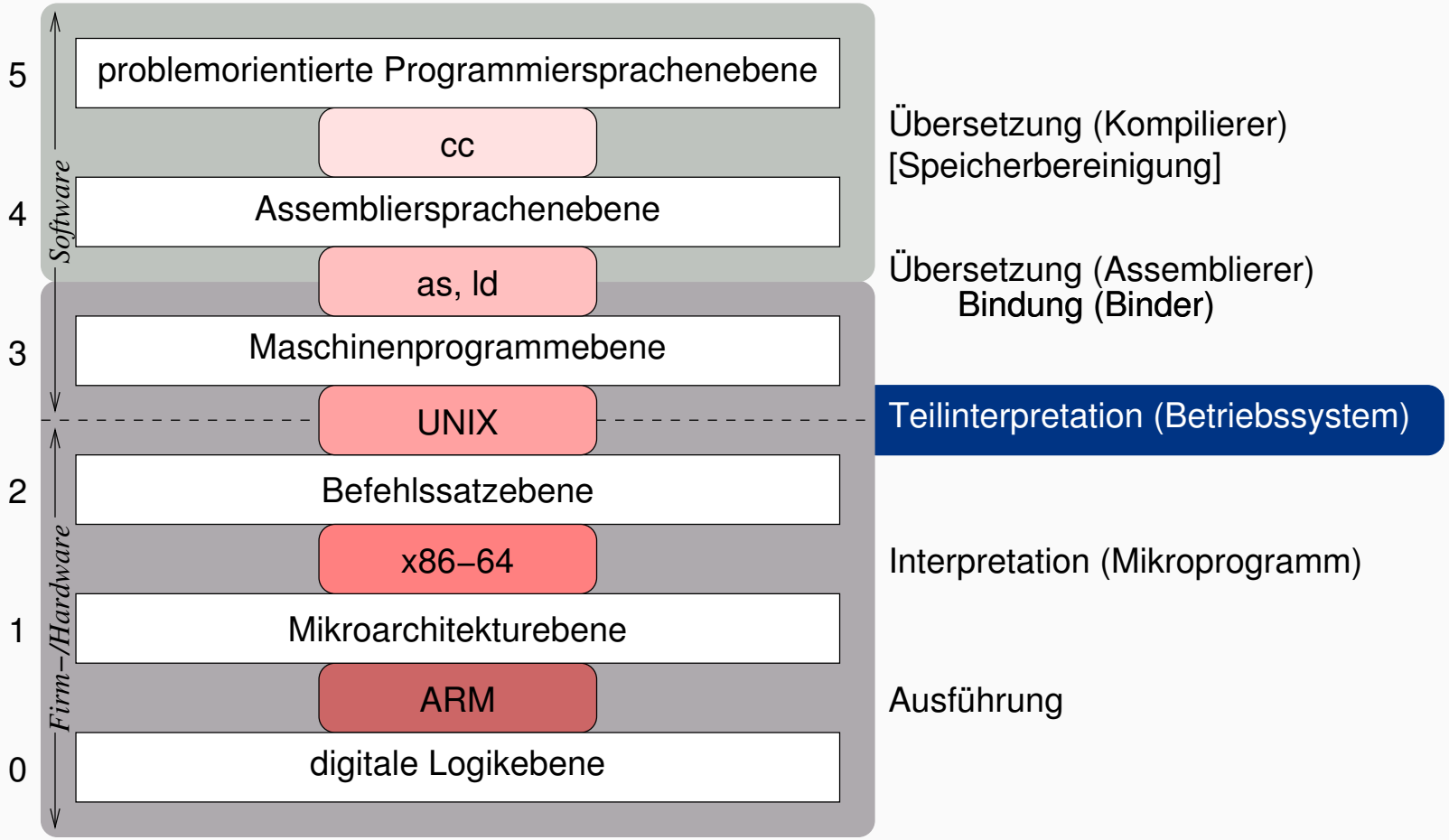
Interpretation

Demarkationslinie

- Methode der Abbildung
  - ≥ 4 Übersetzung vs.
  - ≤ 3 Interpretation
- Art der Programmierung
  - ≥ 4 Anwendung vs.
  - ≤ 3 System
- Natur der Sprache
  - ≥ 4 symbolisch vs.
  - ≤ 3 numerisch

- Schichten der Ebene$_{[4,5]}$ sind nicht wirklich existent
  - sie werden via Übersetzung aufgelöst und auf tiefere Ebenen abgebildet
  - so dass am Ende nur ein Maschinenprogramm (Ebene$_3$) übrigbleibt

Schichten und abstrakte/reale Prozessoren [10]



- RISC auf Ebene$_1$ und gegebenenfalls (hier) CISC auf Ebene$_2$
  - nach außen „*complex*“, innen aber „*reduced instruction set computer*“
  - Intel *Core* oder *Haswell* $\leftrightarrow$ AMD *Bulldozer* oder *Zen* (ARM)

Systemaufrufschnittstelle (*system call interface*)[2] [7]

```
1  read:
2    push %ebx
3    movl 16(%esp),%edx
4    movl 12(%esp),%ecx
5    movl 8(%esp),%ebx
6    mov  $3,%eax
7    int  $0x80
8    pop  %ebx
9    cmp  $-4095,%eax
10   jae  __syscall_error
11   ret
```

- „Grenzübergangsstelle“ **Aufrufstumpf**
  - einerseits erscheint ein Systemaufruf als normaler Prozeduraufruf
  - andererseits bewirkt der Systemaufruf einen Moduswechsel
- sorgt für **Ortstransparenz** (funktional)
  - die Lokalität der aufgerufenen Systemfunktion muss nicht bekannt sein

- Systemaufrufe sind **Prozedurfernaufrufe**, um **Prozessdomänen** in kontrollierter Weise zu überwinden

3–5
- tatsächliche Parameter (Argumente) in Registern übergeben

6
- Systemaufrufnummer (Operationskode) in Register übergeben

7
- Domänenwechsel (Ebene$_3$ $\mapsto$ Ebene$_2$) auslösen
- Aufruf abfangen (*trap*) und dem Betriebssystem zustellen

9–10
- Status überprüfen und ggf. Fehlerbehandlung durchführen

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

Teilinterpretation [4]

- Befehle der Maschinenprogrammbene, also Ebene_3-Befehle sind…
  - „normale“ Befehle der Ebene_2, die die CPU direkt ausführen kann
    - **unprivilegierte Befehle**, die in jedem Arbeitsmodus ausführbar sind
  - „unnormale“ Befehle der Ebene_2, die das Betriebssystem ausführt
    - **privilegierte Befehle**, die nur im privilegierten Arbeitsmodus ausführbar sind
- die „aus der Reihe fallenden“ Befehle stellen Adressräume, Prozesse, Speicher, Dateien und Wege zur Ein-/Ausgabe bereit
  - Interpreter dieser Befehle ist das Betriebssystem
  - der dadurch definierte Prozessor ist die **Betriebssystemmaschine**
- demzufolge ist ein Betriebssystem immer nur **ausnahmsweise** aktiv
  - es muss von außerhalb aktiviert werden
    - programmiert im Falle eines Systemaufrufs (`CD80`: Linux/x86) oder einer sonstigen synchronen Programmunterbrechung (*trap*)
    - nicht programmiert, also nicht vorhergesehen, im Falle einer asynchronen Programmunterbrechung (*interrupt*)
  - es deaktiviert sich immer selbst, in beiden Fällen programmiert (`CF`: x86)

SP1

Betriebssystemkonzepte

Prozess

Betriebssysteme bringen Programme zur Ausführung, in dem dazu Prozesse erzeugt, bereitgestellt und begleitet werden

- im Informatikkontext ist ein Prozess ohne Programm nicht möglich
  - die als Programm kodierte Berechnungsvorschrift definiert den Prozess
  - das Programm legt damit den Prozess fest, gibt ihn vor
  - gegebenenfalls bewirkt, steuert, terminiert es gar andere Prozesse
    - wenn das Betriebssystem die dazu nötigen Befehle anbietet!
- ein Programm beschreibt die Art des Ablaufs eines Prozesses

  sequentiell
  - eine Folge von zeitlich nicht überlappenden Aktionen
  - verläuft deterministisch, das Ergebnis ist determiniert

  parallel
  - nicht sequentiell
  - in beiden Arten besteht ein Programmablauf aus **Aktionen**

Beachte: Programmablauf und Abstraktionsebene

Ein und derselbe Programmablauf kann auf einer Abstraktionsebene sequentiell, auf einer anderen parallel sein.

Speicherverwaltung: Politiken

- Aufgabe ist es, über die **Speicherzuteilung** an einen Prozess Buch zu führen und seine Adressraumgröße dazu passend auszulegen

  Platzierungsstrategie (*placement policy*)
  - wo im Hauptspeicher ist noch Platz?
- zusätzliche Aufgabe kann die **Speichervirtualisierung** sein, um den Mehrprogrammbetrieb zu maximieren

  Ladestrategie (*fetch policy*)
  - wann muss ein Datum im Hauptspeicher liegen?

  Ersetzungsstrategie (*replacement policy*)
  - welches Datum im Hauptspeicher ist ersetzbar?
- die zur Durchführung dieser Aufgaben zu verfolgenden Strategien profitieren voneinander — oder bedingen einander
  - ein Datum kann ggf. erst platziert werden, wenn Platz freigemacht wurde
  - etwa indem das Datum den Inhalt eines belegten Speicherplatzes ersetzt
  - ggf. aber ist das so ersetzte Datum später erneut zu laden
  - bevor ein Datum geladen werden kann, ist Platz dafür bereitzustellen

Speicherverwaltung: „Reviere“ [11]

- normalerweise sind die **Verantwortlichkeiten** auf mehrere Ebenen innerhalb eines Rechensystems verteilt

  Speicherzuteilung
  - Maschinenprogramm und Betriebssystem
  - Haldenspeicher, Hauptspeicher

  Speichervirtualisierung
  - ist allein Aufgabe des Betriebssystems
  - Haupt-/Arbeitsspeicher, Ablage

- das Maschinenprogramm verwaltet den seinem Prozess (-adressraum) jeweils zugeteilten Speicher **lokal** eigenständig
  - stellt dabei **sprachenorientierte Kriterien** in den Vordergrund
    - typisch für den Haldenspeicher $\rightsquigarrow$ `malloc`/`free`
- das Betriebssystem verwaltet den gesamten Haupt-/Arbeitsspeicher **global** für alle Prozessexemplare bzw. -adressräume
  - stellt dabei **systemorientierte Kriterien** in den Vordergrund
    - hilft, einen Haldenspeicher zu verwalten $\rightsquigarrow$ z.B. `sbrk`/`mmap`
- Maschinenprogramm und Betriebssystem gehen somit eine **Symbiose** ein, sie nehmen eine **Arbeitsteilung** vor
  - genauer gesagt: das Laufzeitsystem (`libc`) im Maschinenprogramm

Prozessadressraum

- **Seitenadressierung** (*paging*) mittels **Seitentabelle** [11, S. 29–30]
  - jede von der CPU generierte Adresse wird gedeutet als $A_p = (p, o)$, wobei

    Versatz $o = [0, 2^w - 1]$, mit $9 \leq w \leq 30$ (*offset*)

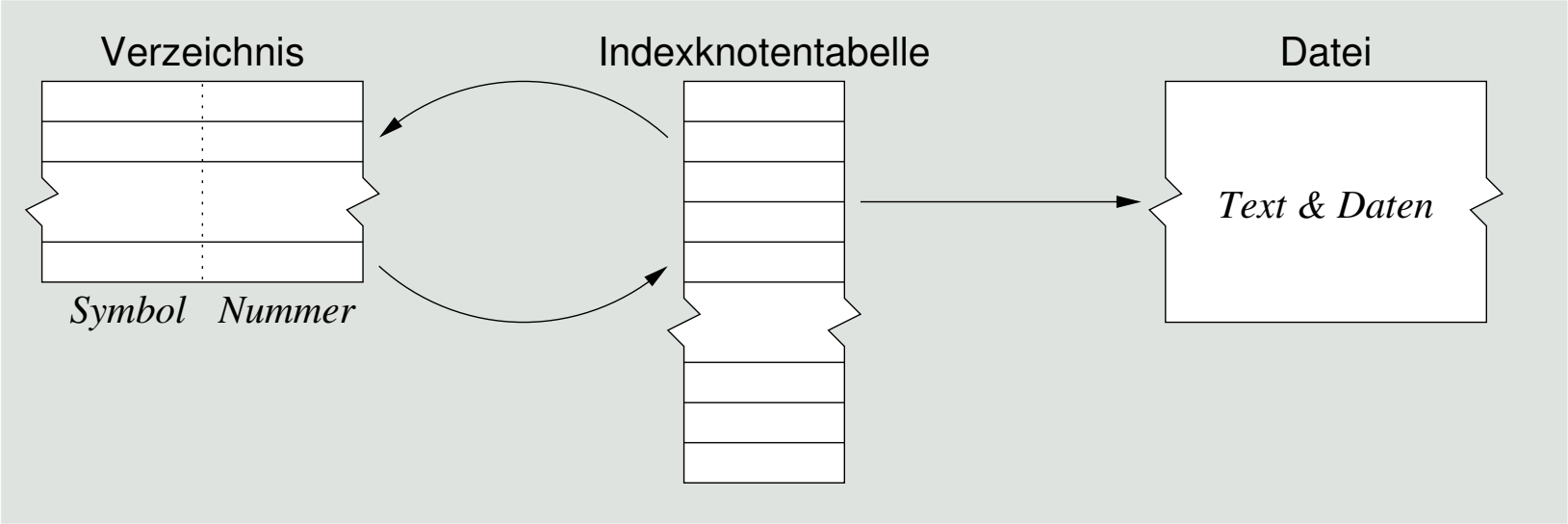
    Seitennummer $p = [0, 2^{n-w} - 1]$, mit $32 \leq n \leq 64$, Tabellenindex
  - eine gewöhnliche **lineare Adresse** $\rightsquigarrow$ **eindimensionaler Adressraum**
    - d.h., Oktetts oder Worte in einer Dimension aufgereiht
- **Segmentierung** (*segmentation*) mittels **Segmenttabelle** [3, S. 37]
  - jede Adresse ist repräsentiert als Zweitupel $A_s = \langle s, d \rangle$, wobei

    Segmentname $s = [0, 2^m - 1]$, mit $12 \leq m \leq 18$, Tabellenindex

    Verschiebung $d = [0, 2^n - 1]$, mit $32 \leq n \leq 64$ (*displacement*)
  - Zweikomponentenadresse $\rightsquigarrow$ **zweidimensionaler Adressraum**
    - d.h., Segmente in der ersten und Segmentinhalte in der zweiten Dimension

- Kombination (vgl. [3, S. 38–40]):
  - **segmentierte Seitenadressierung** (*segmented paging*)
    - die Seitentabellen sind segmentiert, d.h., $A_p = (p, o)$ mit $p = (s, d)$
  - **seitennummerierte Segmentierung** (*paged segmentation*)
    - die Segmente sind seitennummeriert, d.h., $A_s = \langle s, d \rangle$ mit $d = (p, o)$ oder die Segmenteinheit generiert eine lineare Adresse A_p für die Seiteneinheit

Organisation des Ablagespeichers



- die **Indexknotentabelle** (*inode table*) ist ein statisches Feld (*array*) von Indexknoten und die zentrale Datenstruktur
  - ein Indexknoten ist **Deskriptor** insb. eines Verzeichnisses oder einer Datei
- das **Verzeichnis** (*directory*) ist eine **Abbildungstabelle**, es übersetzt symbolisch repräsentierte Namen in Indexknotennummern
  - eine von der Namensverwaltung des Betriebssystems definierte Datei
- die **Datei** (*file*) ist eine abgeschlossene Einheit zusammenhängender Daten beliebiger Repräsentation, Struktur und Bedeutung

SP1

Betriebsarten

Stapelbetrieb [12]

- abgesetzter Betrieb: Satellitenrechner, Hauptrechner
  - Entlastung durch Spezialrechner
- überlappte Ein-/Ausgabe: DMA, *Interrupts*
  - nebenläufige Programmausführung
- überlappte Auftragsverarbeitung: Einplanung, Vorgriff
  - Verarbeitungsstrom von Aufträgen
- abgesetzte Ein-/Ausgabe: *Spooling*
  - Entkopplung durch Pufferbereiche
- Mehrprogrammbetrieb: *Multiprogramming*
  - Multiplexen der CPU
- dynamisches Laden: Überlagerung (*overlay*)
  - programmiertes Nachladen von Programmbestandteilen

Mehrzugangsbetrieb [5]

- Dialogbetrieb: Dialogstationen
  - mehrere Benutzer gleichzeitig bedienen können
- Hintergrundbetrieb: Mischbetrieb
  - Programme im Vordergrund starten
- Teilnehmerbetrieb: Zeitscheibe, *Timesharing*
  - eigene Dialogprozesse absetzen können
- Teilhaberbetrieb: residente Dialogprozesse
  - sich gemeinsame Dialogprozesse teilen können
- Multiprozessorbetrieb: Parallelrechner, SMP
  - Parallelverarbeitung von Programmen
- Speicheraustausch: *Swapping*, *Paging*
  - von ganzen Prozessadressräumen oder einzelnen Bestandteilen

Echtzeitbetrieb: Terminvorgaben [5, S. 12*ff.*]

- externe (physikalische) Prozesse definieren, was genau bei einer nicht termingerecht geleisteten Berechnung zu geschehen hat:
  weich (*soft*) auch „schwach“
    - das Ergebnis ist weiterhin von Nutzen, verliert jedoch mit jedem weiteren Zeitverzug des internen Prozesses zunehmend an Wert
    - die Terminverletzung ist tolerierbar

  fest (*firm*) auch „stark“
    - das Ergebnis ist wertlos, wird verworfen, der interne Prozess wird abgebrochen und erneut bereitgestellt
    - die Terminverletzung ist tolerierbar

  hart (*hard*) auch „strikt“
    - Verspätung des Ergebnisses kann zur „Katastrophe“ führen, dem int. Prozess wird eine **Ausnahmesituation** zugestellt
    - Terminverletzung ist keinesfalls tolerierbar — aber möglich…
- ggf. zusätzlich geforderte Randbedingung ist die Termineinhaltung unter allen Last- und Fehlerbedingungen

Gliederung

SP2

Ausblick

- Prozessverwaltung
  - Einplanung (klassisch, Fallstudien)
  - Koroutinen, Programmfäden, Einlastung
- Synchronisation
  - ein-/mehrseitig, blockierend/nicht-blockierend
  - Verklemmungen (Gegenmaßnahmen, Auflösung)
- Speicherverwaltung
  - Adressräume, MMU (Pentium)
  - Disziplinen, virtueller Speicher, Arbeitsmenge
- Dateiverwaltung
  - Organisation des Hintergrundspeichers
  - Datenverfügbarkeit (RAID)

Literaturverzeichnis (1)

[1] Kleinöder, J. :
Kurzeinführung in C.
In: [13], Kapitel 2

[2] Kleinöder, J. :
Vom C-Programm zum UNIX-Prozess.
In: [13], Kapitel 3

[3] Kleinöder, J. ; Schröder-Preikschat, W. :
Adressbindung.
In: [13], Kapitel 6.3

[4] Kleinöder, J. ; Schröder-Preikschat, W. :
Betriebssystemmaschine.
In: [13], Kapitel 5.3

Literaturverzeichnis (2)

[5] Kleinöder, J. ; Schröder-Preikschat, W. :
Dialog- und Echtzeitverarbeitung.
In: [13], Kapitel 7.2

[6] Kleinöder, J. ; Schröder-Preikschat, W. :
Einführung.
In: [13], Kapitel 4

[7] Kleinöder, J. ; Schröder-Preikschat, W. :
Maschinenprogramme.
In: [13], Kapitel 5.2

[8] Kleinöder, J. ; Schröder-Preikschat, W. :
Organisation.
In: [13], Kapitel 1

Literaturverzeichnis (3)

[9] Kleinöder, J. ; Schröder-Preikschat, W. :
Prozesse.
In: [13], Kapitel 6.1

[10] Kleinöder, J. ; Schröder-Preikschat, W. :
Rechnerorganisation.
In: [13], Kapitel 5.1

[11] Kleinöder, J. ; Schröder-Preikschat, W. :
Speicher.
In: [13], Kapitel 6.2

[12] Kleinöder, J. ; Schröder-Preikschat, W. :
Stapelverarbeitung.
In: [13], Kapitel 7.1

Literaturverzeichnis (4)

[13] Kleinöder, J. ; Schröder-Preikschat, W. ; Lehrstuhl Informatik 4 (Hrsg.):
Systemprogrammierung.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)