Kapitel 10 - Betriebssysteme

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Aktueller Stand

- Was wir bis jetzt kennengelernt haben:
- Grundlagen von Programmiersprachen (grundlegende Konzepte, Kontrollfluss, Funktionen, Datentypen) ✅
- Interner Aufbau von Datentypen und Abbildung auf die Hardware ✅
- Speziell für hardwarenahe Sprachen:
  - Zeiger ✅
  - Speicherverwaltung ✅

Aktueller Stand

- Aber: Ein Programm läuft eher selten direkt auf der nackten Hardwareplattform
- Zoomen wir einmal raus:
  - **Betriebssystem** als Plattform für Programme → heute
  - **Compiler-Toolchain** für das Bauen eines Programms → nächstes Kapitel

Literatur

- Der Themenkomplex „Betriebssysteme“ ist **sehr** groß
- Nicht mit einer Vorlesung abzudecken
- Empfehlung: Besuch der Veranstaltung „Betriebssysteme“

  (oder selbstständig Themengebiet erarbeiten)
- Literatur: Andrew S. Tanenbaum - *Moderne Betriebssysteme* [1]
  - Eines der Standardwerke über Betriebssysteme
  - Aktuelle Version: 5. Auflage (2025)
  - In der Bibliothek verfügbar: 4. Auflage (ebenfalls gut geeignet)

Rückblick: Was ist ein Betriebssystem?

- Betriebssystem hat zwei Hauptaufgaben [1]:

1. Bereitstellung von Abstraktionen der Betriebsmittel
2. Verwaltung der Hardwareressourcen, die der Computer besitzt

Rückblick: Was ist ein Betriebssystem?

- Daraus folgt:

  Alles eine Anwendung, die auf dem Betriebssystem aufsetzt

- Dies alles ist nicht Teil eines Betriebssystems:
  - Graphische Oberflä
  - Anwendungen
  - Kommandozeile

Wir schauen jetzt mal unter die Haube!

- Beispiel GNU/Linux
  - Linux ist der Betriebssystemkern (*Kernel*)
  - GNU ist Sammlung von essenziellen Programmen (*GNU Core Utils*) aufsetzen

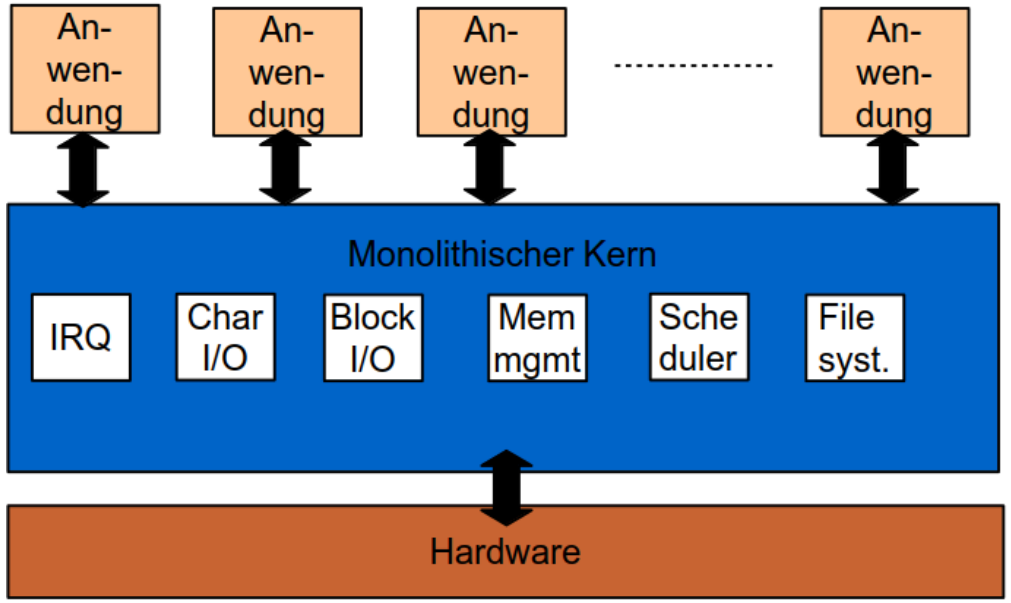
Der Maschinenraum

© Microsoft Imagine - mit KI generiert

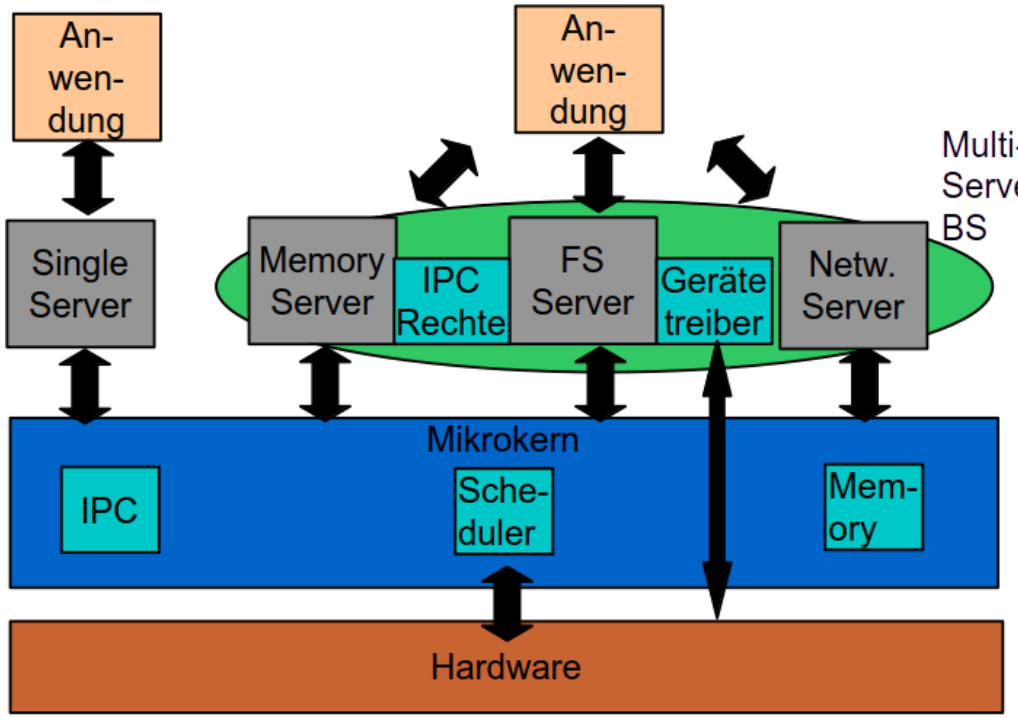
Struktur eines Betriebssystems

- Verschiedene Kategorien für den grundlegenden Aufbau eines Betriebssystems
- Zwei dominante Konzepte:

Monolithisches System



Mikrokern-System

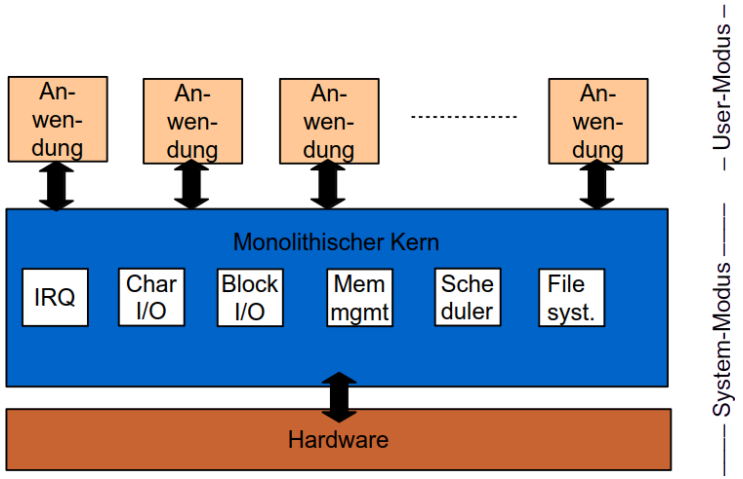


– User-Modus –

– System-Modus –

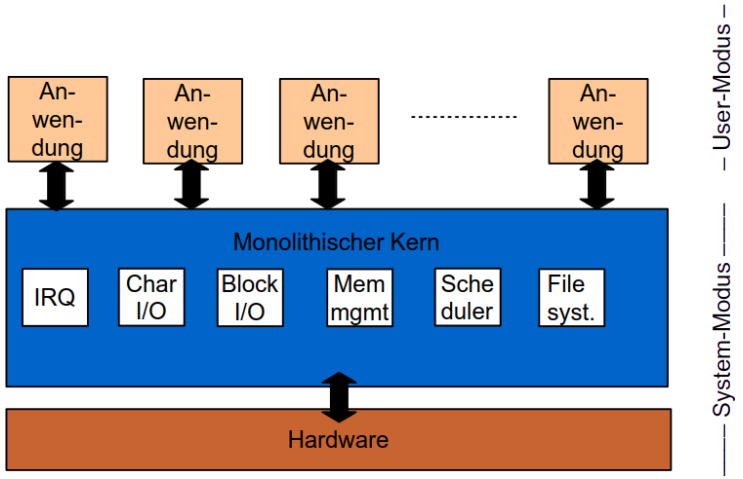
Monolithisches Betriebssystem

- Existieren schon lange: seit Anfang der 1970er Jahre
- Bekanntestes Beispiel: Linux
- Sind prinzipiell sehr einfach strukuriert
- Aufteilung in
  - Anwendungsebene (*User Mode*) und
  - Systemebene (*Kernel Mode*)



Anwendungsebene

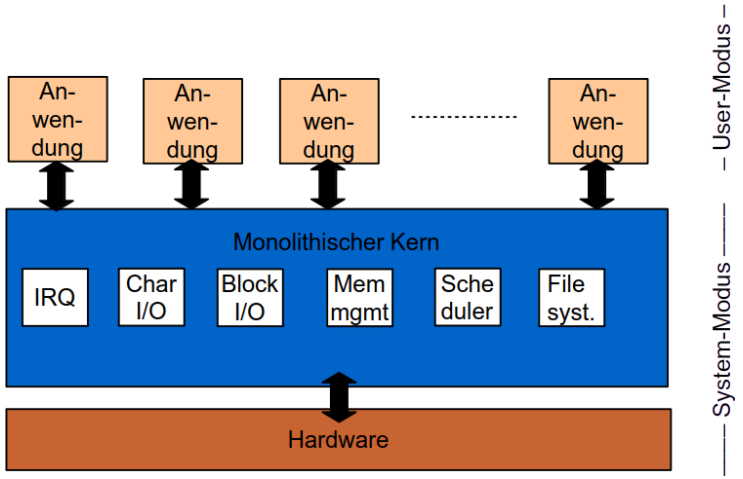
- Anwendungsebene ist nicht privilegiert
- Darf nur, was das Betriebssystemebene erlaubt
- Anwender:in ist auf diese Ebene beschränkt
- Interagiert mit darunterliegenden Betriebssystem
- Definierte Funktionen erlauben Anwendung Eintritt in Betriebssystem
  - Heißen *System Calls*
  - Wechsel auf Betriebssystemebene zur Ausführung notwendig
  - Beispiel: `read()` – „Lese etwas aus einer Datei“



Systemebene

- Systemebene ist das eigentliche Betriebssystem
- Direkter Zugriff auf die Hardware
- Stellt Treiber und Abstraktionen für Interaktion bereit
- Darf **alles** privilegiert ausführen

  → Große Verantwortung
- Fehlerhafte Programmcode kann gesamtes System zum Absturz bringen



Vor- und Nachteile eines Monolithen

Vorteile

- Unkomplizierte Struktur
- Relativ gute Performance
  - Ebenenwechsel kostet Zeit
  - Viel Kommunikation innerhalb einer Ebene

Nachteile

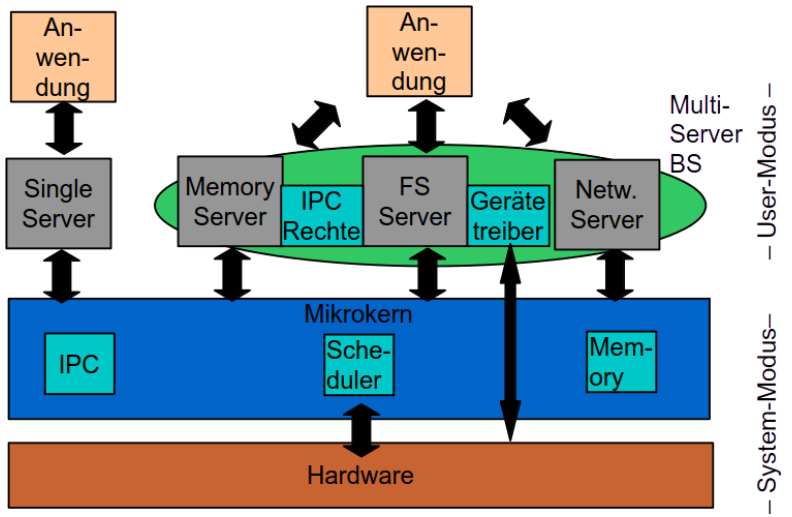
- Erweiterungen betreffen gesamten Kern

  → Neukompilierung notwendig
- Sicherheit: **Eine** Komponente „reißt kompletten Kern in den Tod“

Mikrokern-Betriebssystem

- Existieren ebenfalls schon lange
- Bekannteste Beispiele: MacOS
- Ziel: Möglichst kleine Systemebene

  → **Nur** absolut notwendige Komponenten
- Treiber auf Anwendungsebene verschieben



- **Viel** Kommunikation zwischen den Anwendungen: *Inter-Process Communication* (IPC)

  → Interaktion mit der jeweiligen System-Komponente auf Nutzerebene

Vor- und Nachteile eines Mikrokerns

Vorteile

- Flexibel erweiterbar
  - Neuer Treiber als Server auf Anwendungsebene
- Robuste Architektur
  - Fehler sind auf Komponente beschränkt
  - Komponente einfach neustarten →ist Anwendung

Nachteile

- Alles hängt von Kommunikationsmechanimus ab
- *IPC* **muss** performant sein

Abstraktionen

Übersicht

Was für Abstraktionen gibt es in einem Betriebssystem?

- Prozessor
- (Ein-Ausgabe-)Geräte
  - Speichermedien (Festplatten, SSDs, ...)
  - USB-Geräte (Maus, Tastatur, Webcam, ...)
  - Viele weitere mehr: Grafikkarte, Soundkarte, Netzwerk, ...
- Speicher
  - Arbeitsspeicher
  - Hintergrundspeicher (Festplatten, SSD, ...)

(Dies ist keine abschließende Liste)

Prozessor

- Prozessor ist zentrale Recheneinheit eines Computers (*Central Processing Unit*, *CPU*)
- Besteht unter anderem aus:
  - Kontrolleinheit
    - Dekodiert Instruktionen und steuert Kontrollfluss
  - CPU-Register
    - Beinhalten Werte und (Zwischen-)Ergebnisse der Berechnungen
  - *Floating Point Unit* (*FPU*)
    - Spezialisierte Komponente für FP-Zahlen
  - *Arithmetic Logical Unit* (*ALU*)
    - Spezialisierte Komponente für Rechenoperationen und boolesche Algebra

© Andrew Dunn, CC BY-SA 2.0

Prozessor

- Interner Aufbau wird durch **Prozessorarchitektur** beschrieben
- Architekturen durchaus sehr verschieden
- Einige Komponenten grundsätzlich überall vorhanden
- Relevante Prozesssorarchitekturen: **x86**, **ARM** und **RISC-V**

© Andrew Dunn, CC BY-SA 2.0

Relevante Prozessorarchitekturen

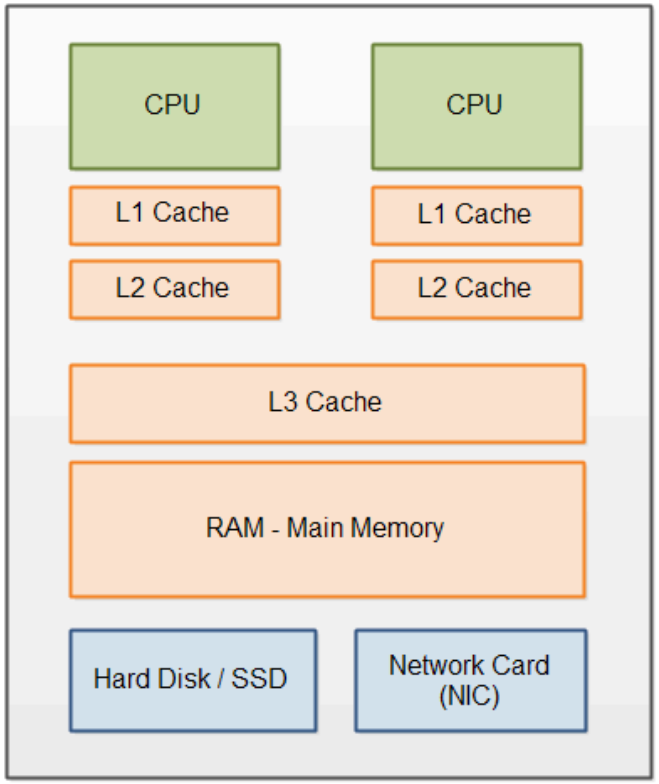
- x86 / amd64
  - Hauptsächlich Desktops und Laptops
  - Hersteller: Intel und AMD (Ursprung: Intel 80**86**)
  - Philosophie: Größtmögliche Abwärtskompatibilät
  - *CISC*-Vertreter
- ARM
  - Hauptsächlich Mobilgeräte
  - Designer: ***A**dvanced **R**ISC **M**achines*
  - Spezifiziert Designs und verkauft Lizenzen an verschiedene Hersteller
  - Designs beinhalten unter anderem Instruktionssatz/interne Komponenten

Relevante Prozessorarchitekturen

- RISC-V (*Reduced Instruction Set Computer 5*)
  - Typischer Einsatz: Mikrocontroller
  - Komplett offener Befehlssatz ohne kostenpflichtige Lizenzen
  - Jeder Hersteller kann frei entwickeln/anpassen und bauen
  - Seit ca. 2015 verfügbar (Vorgängerversionen waren Forschungsprojekte)
  - Noch nicht so dominant wie ARM und x86, aber bereits sehr beliebt bei Herstellern
- Ergeben sich zwei Fragen:
  - Aber wie wird ein Programm auf der CPU ausgeführt?
  - Wie Instruktionsströme von Firefox und Thunderbird unterscheiden? 🤔

Speichermodell – Caches

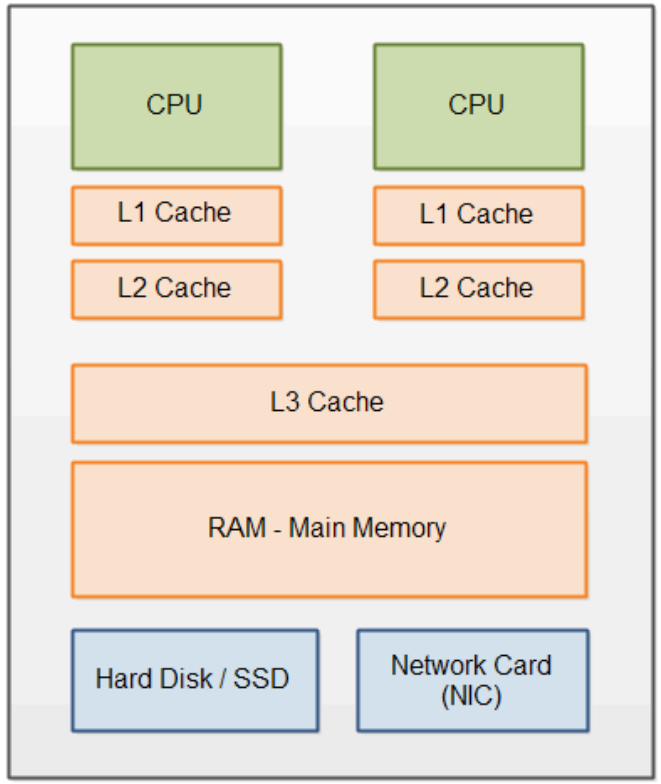
- Wichtiges Element von x86-CPUs sind Caches
- Schnelle, mehrstufige Zwischenspeicher
- Enthalten häufig benutzte/vor Kurzem benötigte Speicherblöcke
- Deutlich schneller als Zugriff auf RAM
- Werden mit Blöcken von je 64 Bytes (Cache Lines) befüllt



© *Jakob Jenkov*

Speichermodell – Caches

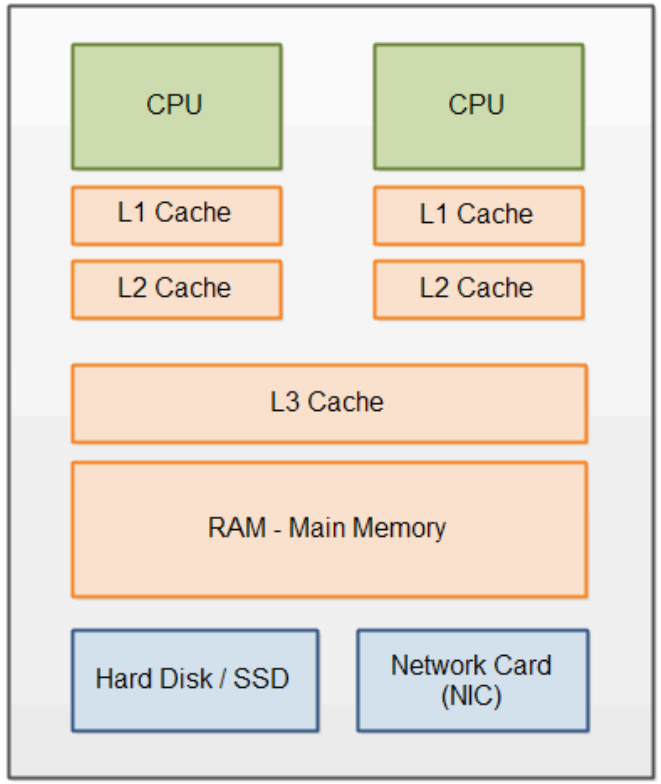
- Je näher am Kern, desto kleiner und schneller
- Ein anschaulicher Vergleich: Arbeit in einer Firma



© *Jakob Jenkov*

Speichermodell – Caches

- CPU-Kern ist ein Arbeitsplatz
- L1-Cache ist private Ablagefläche
  - Enthält gerade häufig benutzte Gegenstände
  - Beispiel: Zange (gerade wird geschraubt)
- L2-Cache ist Regal mit benötigten Gegenständen
  - Beispiel: Getränkeflasche
- L3-Cache ist angrenzender Lagerraum
  - Wird von mehreren Arbeitern parallel genutzt
- RAM ist Zentrallager am anderen Ende der Firma
  - Arbeiter läuft nur hin, wenn Bauteil benötigt wird → Zugriff dauert „lange“

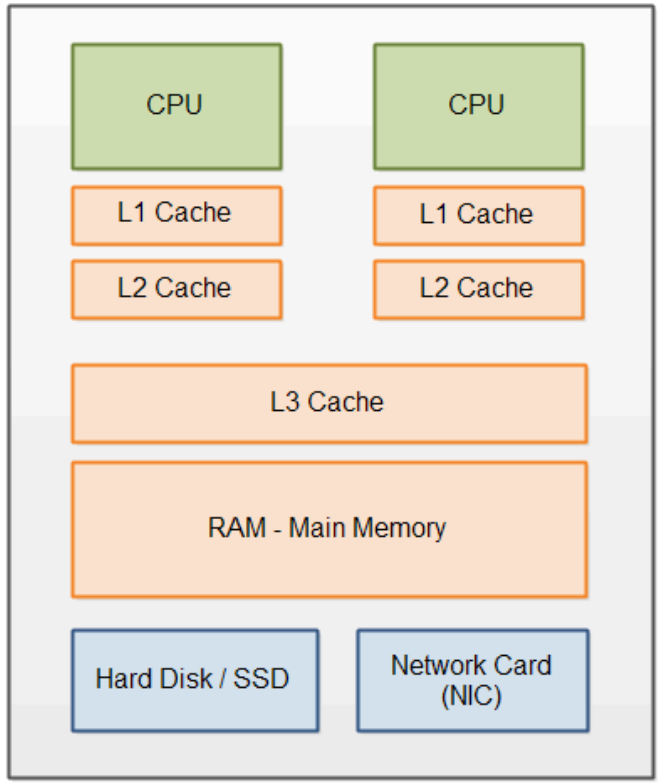


© *Rajiv Prabhakar*

Speichermodell – Caches

- Sind ein wichtiger Bestandteil der Performance moderner CPUs
- Zugriff auf RAM (statt L1)

  → Latenz ca. 100-fach höher
- Performance eines Programms abhängig von Cacheinhalt
  - Cacheanalyse ist schwierig
  - Gut, wenn Daten im Cache → *Cache Hit*
  - Schlecht, wenn Daten **nicht** im Cache → *Cache Miss*



© *Rajiv Prabhakar*

Speichermodell – CPU-Register

- Daten in CPU werden in Registern abgelegt
- Beispiel:

  `i + 4;` wird zu `add eax, 4` *(C → Assembly)*
- Einige interessante Register sind:
  - `eax` → per Konvention Ergebnis
  - `rsp`/`rbp` → Stackpointer/Basepointer
- Durch erweiterte Befehlssätze im Laufe der Zeit weitere Register eingeführt
  - Beispiel: **MMX** (*Multimedia Extension*)

x86-Register (Auszug)

eax
ebx
ecx
edx
esi
edi
esp
ebp

Prozessabstraktion

- Speicherstruktur eines Prozesses ist bekannt (Stack, Heap, BSS, ...)
- Aber was ist eigentlich ein Prozess?
- Zwei beispielhafte Definitionen [2]:

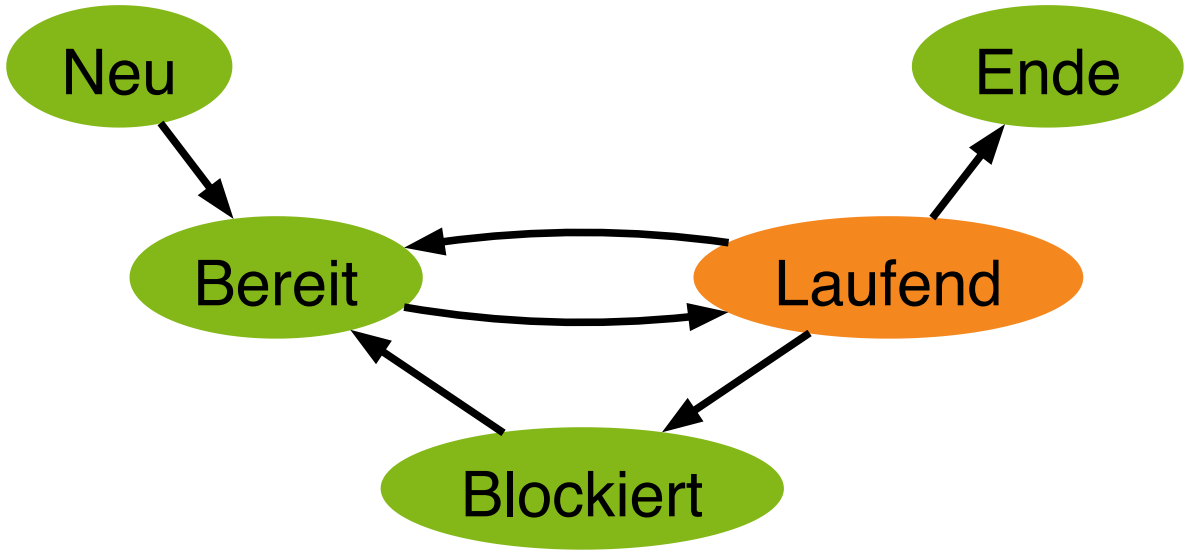
1. „An instance of a program running on a computer."
2. „A program in execution."

- Ein Prozess besitzt Verwaltungsinformationen (*Process Control Block*)
  - Kennung: *PID* = *Process ID* = eine fortlaufende Nummer
  - Zustand: z. B. *Running*
  - Priorität
  - Speicheradresse der nächsten Instruktion: *Program Counter*

Vorteile einer Prozess-Abstraktion

- Betriebssystem verwaltet Rechenzeit und Speicher eines Prozesses!
- Wichtig: Jeder Prozess „denkt“, er sei alleine in dem System:
  - Einem Prozess gehört der Prozessor
  - Einem Prozess gehört der gesamte Adressraum
- Vorteile dieser Abstraktion sind u. a.:
  - **räumliche Isolation:** Jeder Prozess agiert nur in seinem Speicher(gefängnis).
  - **zeitliche Isolation:** Ein Prozess kann **nicht** den Prozessor monopolisieren.

Prozesszustände



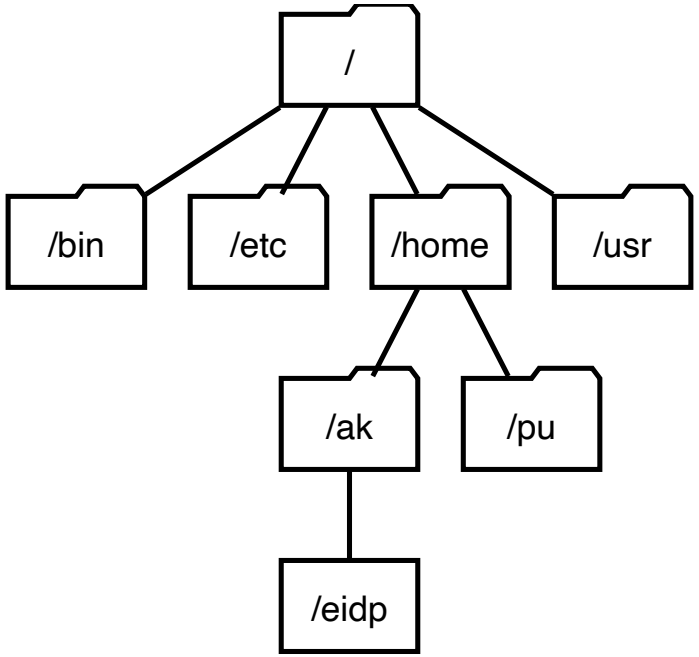
- Prozesse werden *Blockiert*, wenn sie auf ein Ereignis warten
- Prozesse wechseln von *Laufend* in *Bereit*, wenn die Zeitscheibe abgelaufen ist

(Vereinfachte Darstellung, für Details siehe Bücher von Silberschatz, Tanenbaum und anderen [1, 2])

Dateisysteme

- Dateisysteme strukturieren Inhalt von Geräten in Verzeichnisse(/Ordner) und Dateien
- Schaffen Abbildung von Dateien zu Sektoren (und Spuren)
- Beispiele: EXT4, NTFS, FAT32, …
- Kennt Dateien (und Verzeichnisse)
- Zugriff erfolgt Byte-weise:

  „Lese 30 Bytes an Stelle 42“



Dateisysteme – Journaling

- Moderne Dateisysteme notieren Änderungen separat in einer Art Logbuch (*Journal*)
  - Beispiele: NTFS (Windows), Ext4 (Linux)
- Anstehende Änderungen werden **erst** eingetragen und **anschließend** durchgeführt
- Ermöglicht eine schnelle Wiederherstellung des Dateisystems bei Absturz
- Überprüfung der Änderungshistorie auf Konsistenz mit Inhalt des Speichermediums

Einschub: UNIX – *Everything is a file*

- UNIX (und Linux) haben spezielle Philosophie: *Everything is a file*
- Alles wird als eine Datei im Dateisystem abgebildet
- Dazu zählt unter anderem:
  - Hardware-Komponenten und Geräte (in `/dev`)
  - IO-Geräte und Schnittstellen (in `/dev`)
  - Prozesse (in `/proc`)
- Abbildung auf Datei: Aufgabe des Betriebssystems
- Gilt eben so für Interpretation von in Datei geschriebenen Informationen

E/A-Geräte

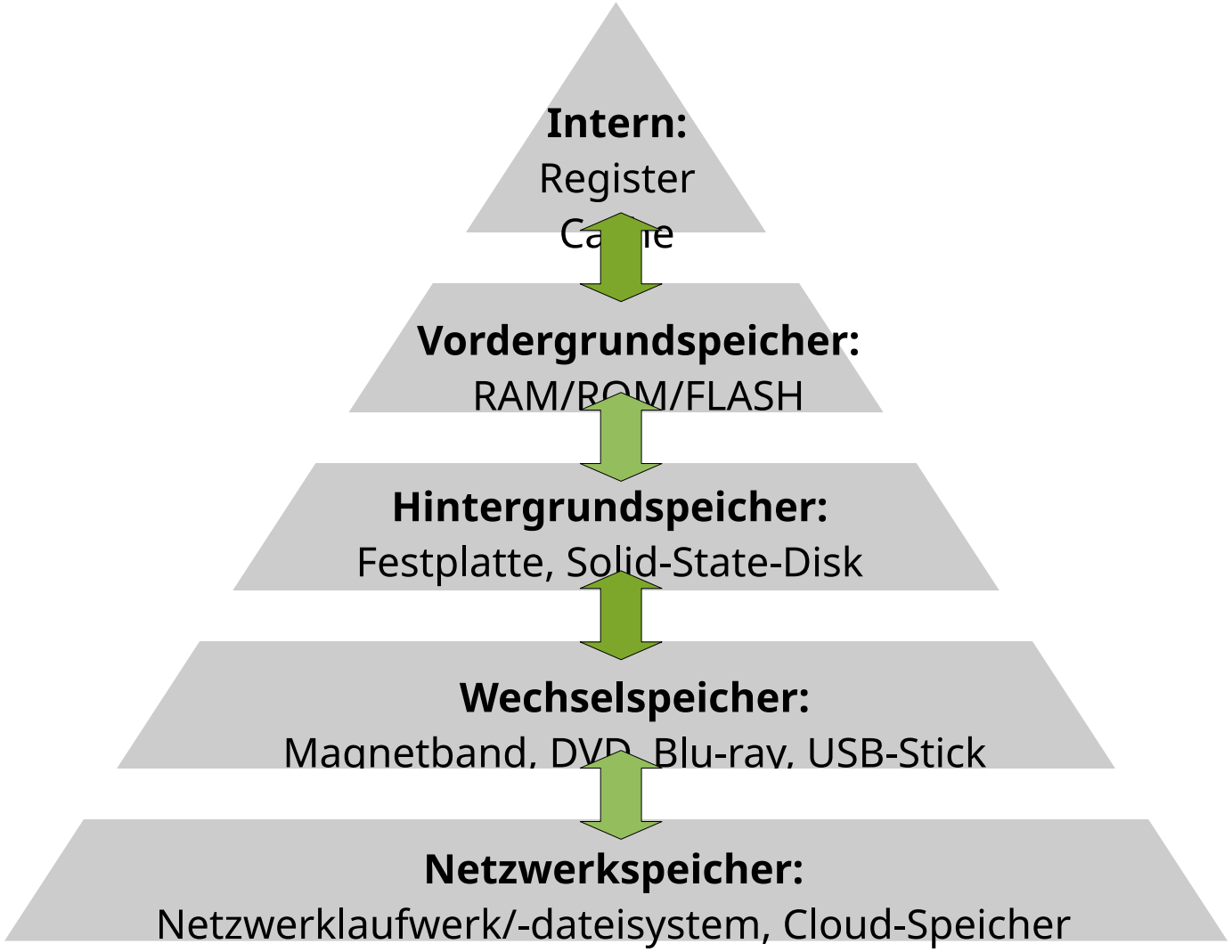
Zwei wesentliche Geräteklassen aus Sicht des Betriebssystems [1, 2]

- Zeichenorientierte Geräte
  - Zeichenorientierte Geräte (*Character Device*) sind interaktive Geräte
  - Beispiel: Tastatur, Maus, Drucker, ...
  - Meist sequentieller Zugriff: „Lese ein Byte nach dem anderen“
- Blockorientierte Geräte
  - Zeichenorientierte Geräte (*Character Device*) sind üblicherweise Speichermedien
  - Beispiel: Festplatte, Diskette, CD-ROM, DVD, BluRay, ...
  - Meist wahlfreier und blockweiser Zugriff: „Lese 512 Byte von Stelle 42“

(Nicht alle Geräte passen in diese Klassen.)

Speicherhierarchie

- Von oben nach unten
  - Zugriffszeiten steigen
  - Kapazität steigt
  - Preis sinkt



 © Folien zur Veranstaltung „Betriebssysteme“

Arbeitsspeicher

- Arbeitsspeicher wird auch als *Random Access Memory* (*RAM*) bezeichnet
- **Wahlfreier Zugriff** (*random access*) auf einzelne Speicherzellen
- Inhalt der Speicherzellen muss zyklisch aufgefrischt werden (sonst droht Ladungsverlust)
  - DDR5: Alle 32 Millisekunden (DDR4: 64 ms)
- Adressierbare Größe wird durch die Bitzahl bestimmt*
  - 64 Bits → 16 Exabytes adressierbare Speicherzellen

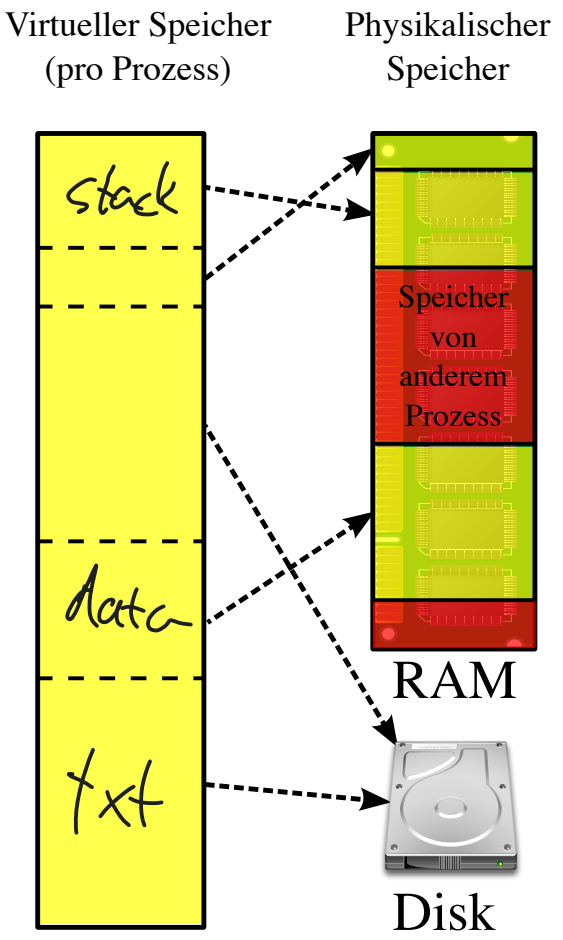
© Wikimedia

*Bei einigen Architekturen (insbesondere `amd64`) sind es in der Praxis „nur“ 48 Bits (256 Terabytes). Die CPU-Hersteller sparen so (noch) nicht benötigte Bits ein

Virtueller Speicher

- Jeder Prozess bekommt gedacht den vollen Adressraum zugeteilt
- Beispiel: 2^{32} Bytes bei 32-Bit-Maschine
- Über *Paging* blendet das Betriebssystem Speicher ein
- Abbildung auf „beliebige“ physikalische Adressen
- Typische Größe einer Seite (*Page*): 4 KiByte

(Wenn der Arbeitsspeicher voll ist, wird auf die Festplatte *ausgelagert*. Siehe *swap* unter Linux oder *page file* unter Windows.)



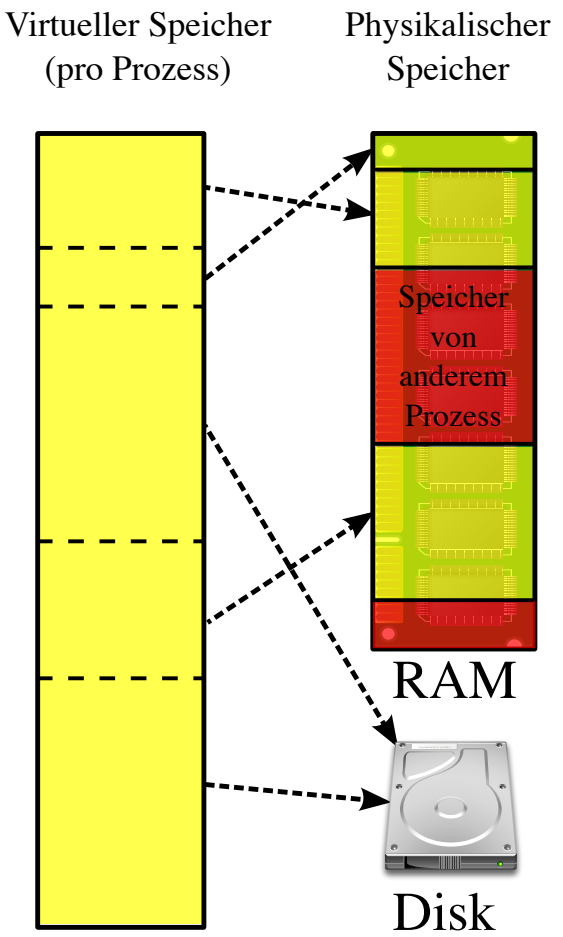
© Ehamberg, CC BY-SA 3.0

Speicherverwaltung

- `malloc()` / `free()` allozieren / geben Speicher frei
- Betriebssystem versucht einen Speicherbereich bereitzustellen
- Bei Erfolg neuen Speicher im virtuellen Adressraum des Prozesses einblenden

  → „Es wird ein neuer Pfeil gezogen.“ (siehe Abbildung)

(In Wirklichkeit geschieht die Anfordung mit `mmap()`-Syscall an das Betriebssystem. Die Funktionsaufrufe werden durch die `libc` bzw. `libstdcpp` verwaltet.)



© Ehamberg, CC BY-SA 3.0

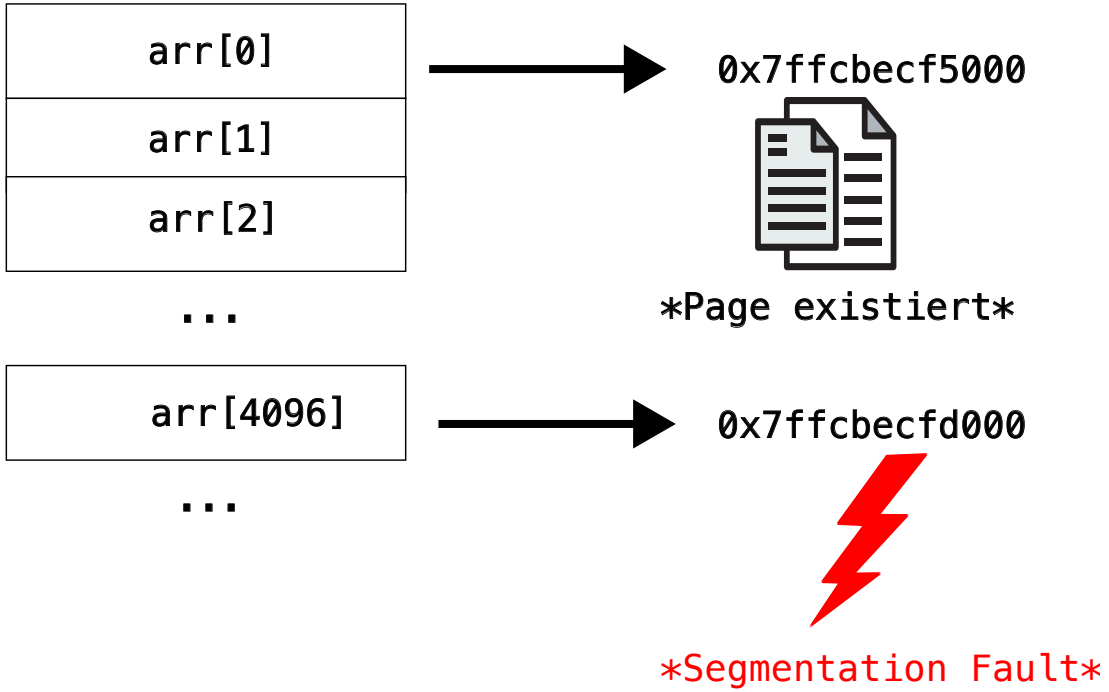
Beispiel – Paging

C Run

```c
#include <stdio.h>

#define OFFSET0 0
#define OFFSET1 512 * 1
#define OFFSET2 512 * 8

int main() {
    int arr[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    int* ptr = arr;
    printf("Adr0: %p\n", ptr + OFFSET0);
    printf("Adr1: %p\n", ptr + OFFSET1);
    printf("Adr2: %p\n", ptr + OFFSET2);
    printf("Val0: %d\n", ptr[OFFSET0]);
    printf("Val1: %d\n", ptr[OFFSET1]);
    // printf("Val2: %d\n", ptr[OFFSET2]);
}
```



Persistente Speicher

- Kein Auffrischen nötig
- Beispiele: Festplatten oder *Solid State Disks* (SSD)
- Dient als Ablage für großen Datenmengen
- Kennt nur Spuren und Sektoren
  - Typische Größen: 512 oder 4096 Bytes
- Bei Zugriff wird ganzer Sektor gelesen: „Lese Sektor 4711 und 4712“

© Hmvh (Wikimedia), CC BY-SA 4.0

Ablaufplanung – *Scheduling*

- Betriebssystem bestimmt Zuteilung von Ressourcen
- Ressourcen sind: CPU, Arbeitsspeicher, Festplatte, Netzwerk, …
- Strategie entscheidet über Zuteilung
  - Zeitpunkt
  - Menge
- Zuteilung erfolgt dynamisch oder statisch
- Statische Zuteilung erfolgt **vor** der Ausführung – *offline*
- Dynamische Zuteilung erfolgt **während** der Ausführung – *online*
- Beschränkung erst mal auf **CPU-Zuteilung**

Parameter der CPU-Ablaufplanung

- Priorität: Ist ein Programm wichtiger als andere?
- Periodizität: Wird periodisch/nur sporadisch Rechenzeit benötigt?
- Präemptivität: Kann die Ausführung unterbrochen werden?
- Bisherige Rechenzeit: Hatte ein Programm schon viel Rechenzeit?
- ...
- Umsetzung über eine Zeitscheibe
  - Prozess bekommt Rechenzeit für feste Zeitspanne
  - Länge der Zeitscheibe hängt von genannten Faktoren ab

Aufbau der Ablaufplanung

- Für Prozesse ist das Scheduling in zwei Teile aufgeteilt:
  1. Dispatcher
  2. Scheduler
- Beide sind eng miteinander verzahnt
- Dispatcher übernimmt die „schmutzige“ Arbeit (korrektes Befüllen der CPU-Register)
  - Beispiel Firma: „Räumt Arbeitsplatz leer, packt Werkzeug in den Spind des Arbeiters”
  - Vorgang wird *Context Switch* genannt (immer an Vorgabe des Schedulers gebunden)
- Scheduler dirigiert Dispatcher und bestimmt nächsten Prozess (oder Arbeiter
  - Zeitscheibe wird durch einen Zeitgeber (*Timer*) vorgegeben
  - Beim „Klingeln“ des Timers → Wechsel des Prozesses

Application Binary Interface (ABI)

- *Application Binary Interface* (ABI) definiert u.a. Verwendung von Registern
  - Beispiel: In `eax` ist Rückgabewert einer Funktion/Ergebnis einer Berechnung
  - Funktionsaufrufe: In welchen Registern Funktionsparameter stehen
  - Welche Größen einzelne Datentypen haben (beeinflusst Register/Zeiger)
- Jedes Betriebssystem kann prinzipiell seine eigene ABI definieren
  - Linux/Unix verwenden *System V Release 4* (1983 erschienen)
  - Windows definiert seine eigene ABI
- Verwechselungsgefahr: *Application Programming Interface* (API)
  - Programmierschnittstelle
  - Beschreibt die Funktionen einer Bibliothek (Rückgabewert, Parameter) und ihre Verwendung

Zwischenstand

- Bisher: Expresstour durch einige wichtige Bereiche von Betriebssystemen
- CPUs und Betriebssysteme beinhalten aber natürlich noch viel mehr! 😃
- Abschließend noch zwei adjazente Bereiche
  - POSIX: Standardisierte Programmierschnittstelle für unixoide Betriebssysteme
  - Man-Pages: „Das Handbuch“ für unixoide Programme und Funktionen

Portable Operating System Interface (POSIX)

- Standard für Betriebssysteme (UNIX bzw. Linux)
- Definiert Funktionalität und Umfang von *API* und Programmen
  - Beispiele für Funktionen: `open()`, `mkdir()`
  - Beispiele für Programme: `ls`, `touch` (*Gnu Core Utils*)
- Ziel: Verfügbarkeit von Programmfunktionen mit
  - gleichen Namen,
  - Parametern und
  - Funktionalität
- Konsequenz: Programme müssen nur einmal geschrieben werden
- Motivation: Einheitliche Schnittstelle für große Zahl an UNIXen in 1990ern

Man Pages

- Kurzform für *Manual Pages* → Handbuchseiten
- Stellen Dokumentation für Programme und Programmierschnittstellen bereit
- Enthält üblicherweise Programmfunktionalität und Benutzungsbeispiele
- Für Funktionen: Parameter, Anforderungen an Parameter
- Lesen vor Verwendung von Programmen und Funktionen ist sehr empfehlenswert!
- Unter Linux von der Kommandozeile aufrufbar: `man man` (Man Page für `man`)
- Gibt es auch als Webseiten (a) oder (b)

Man Pages – Liste der Sektionen

- Sind in 8 relevante Abschnitte gegliedert
- Gliederung gemäß ihrer Funktionalität
- Grund: Einige Programme sind namensgleich zu Funktionen
- Unterscheidung über zusätzliche Angabe der Sektionsnummer
  - Beispiel: `man 1 write`

Abschnitte

1. User Commands: Typische Anwendungsprogramme *(wichtig)*
2. System Calls: Linux-spezifisch *(wichtig)*
3. Library Functions: Programmierfunktionen *(wichtig)*
4. Special Files: Spezielle Dateien (Bsp. `fbdev`)
5. File Formats: Dateiformate
6. Games
7. Conventions and Misc.: Verschiedenes („Wühlkiste“)
8. Administration: Werkzeuge für System-Administratoren

Außerdem: Eigenheiten von Linux

- Einige selbstdefinierte Sektionen (z.B. `0` für Header Files)
- Bei einigen Sektionen Zusatzbuchstaben (`1p` für POSIX-Programme)

Beispiel – Handbuchseite eines Programms

Auszug aus `man ls`

```
LS(1)                                                User Commands                                                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about the FILEs (the current directory by default).  Sort entries alphabetically if none of -cftu-
       vSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file
```

Beispiel – Handbuchseite einer Funktion

Auszug aus `man 3 printf`

```
printf(3)                                   Library Functions Manual                                   printf(3)

NAME
       printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf - formatted output con-
       version

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <stdio.h>

       int printf(const char *restrict format, ...);
        [...]

DESCRIPTION
       The functions in the printf() family produce output according to a format as described below.  The functions printf()
       and  vprintf() write output to stdout, the standard output stream; fprintf() and vfprintf() write output to the given
       output stream; sprintf(), snprintf(), vsprintf(), and vsnprintf() write to the character string str.

        [...]
Conversion specifiers
       A character that specifies the type of conversion to be applied.  The conversion specifiers and their meanings are:

       d, i   The  int argument is converted to signed decimal notation.  The precision, if any, gives the minimum number of
              digits that must appear; if the converted value requires fewer digits, it is padded on the  left  with  zeros.
              The default precision is 1.  When 0 is printed with an explicit precision 0, the output is empty.
        [...]
```

Zusammenfassung

- Grundlegene Betriebssystemarchitekturen: Mikroker vs. Monolith
- Abstraktionen in einem Betriebssystem
  - Speichermodell
  - Prozesse
  - Dateisysteme
  - Speicherhierarchie inkl. virtuellem Speicher
- Prozessverwaltung
- Nachschlagewerk in unixoiden Betriebssystemen wie Linux: *man pages*

Referenzen

[1] A. S. Tanenbaum und H. Bos, *Modern Operating Systems*, 4. Aufl. USA: Prentice Hall Press, 2014.

[2] A. Silberschatz, P. B. Galvin, und G. Gagne, *Operating System Concepts*, 10th Aufl. Wiley, 2018.