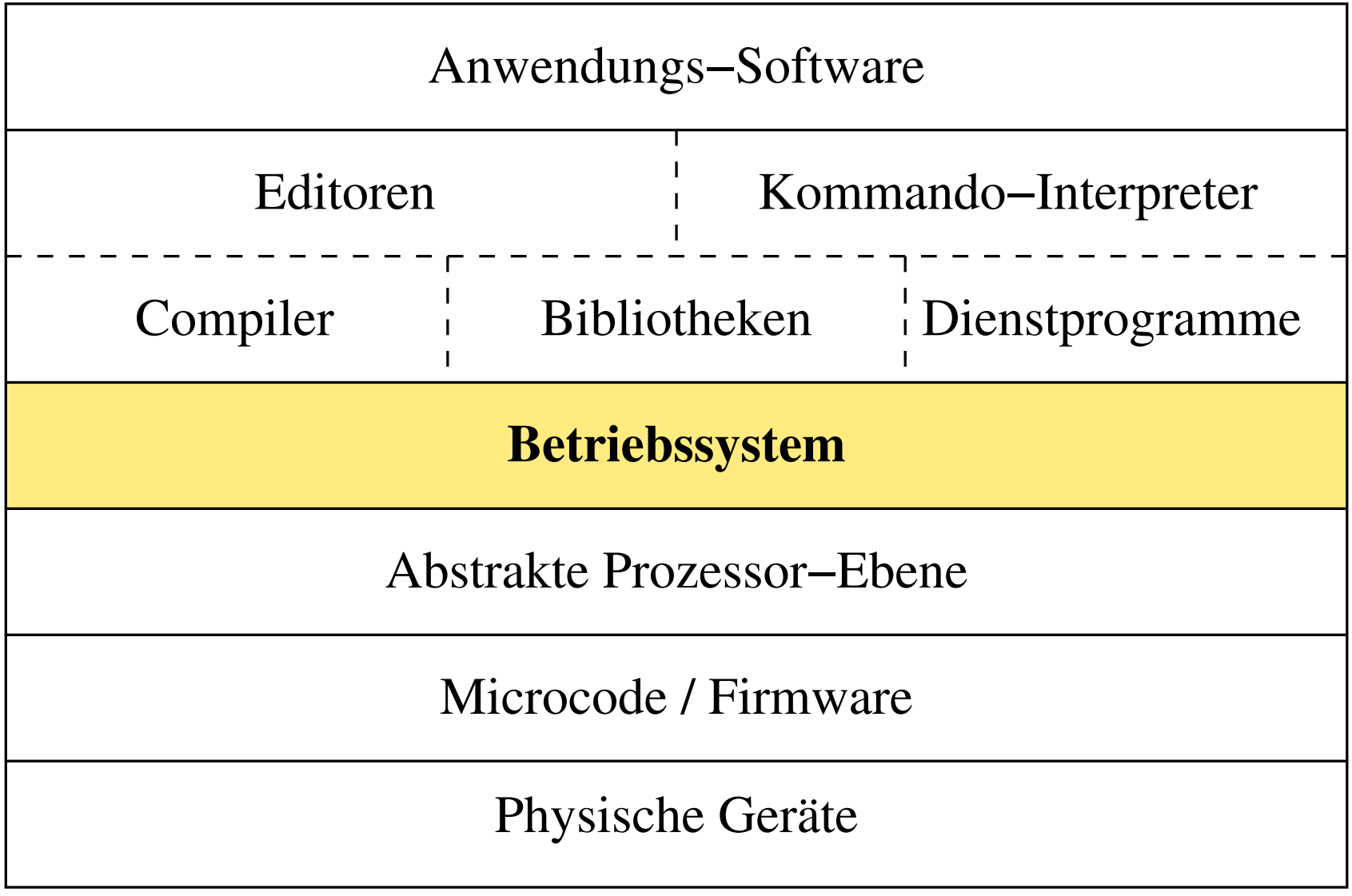
Die *DIN-Norm 44300* definiert ein Betriebssystem wie folgt:

> *Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.*

- Das Betriebssystem ist (abgesehen von der Firmware und einigen Zwischenstufen) das erste Programm, das von einem Rechner beim Hochfahren geladen wird.
- Das Betriebsystem läuft die gesamte Zeit, bis der Rechner wieder heruntergefahren wird.

Das Betriebssystem hat zwei zentrale *Aufgaben*:

- **Ressourcen-Management:** Das Betriebssystem verwaltet und kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie möglichst fair und effizient den einzelnen Nachfragern zu.
- **Erweiterte oder virtuelle Maschine:** Das Betriebssystem besteht aus einer (oder mehreren) Software-Schichten, die über der „nackten“ Hardware liegen. Diese erweiterte Maschine ist einfacher zu verstehen und zu programmieren, da sich komplizierte Zugriffe und Abhängigkeiten hinter einer einfacheren und einheitlichen Schnittstelle verbergen – den Systemaufrufen.

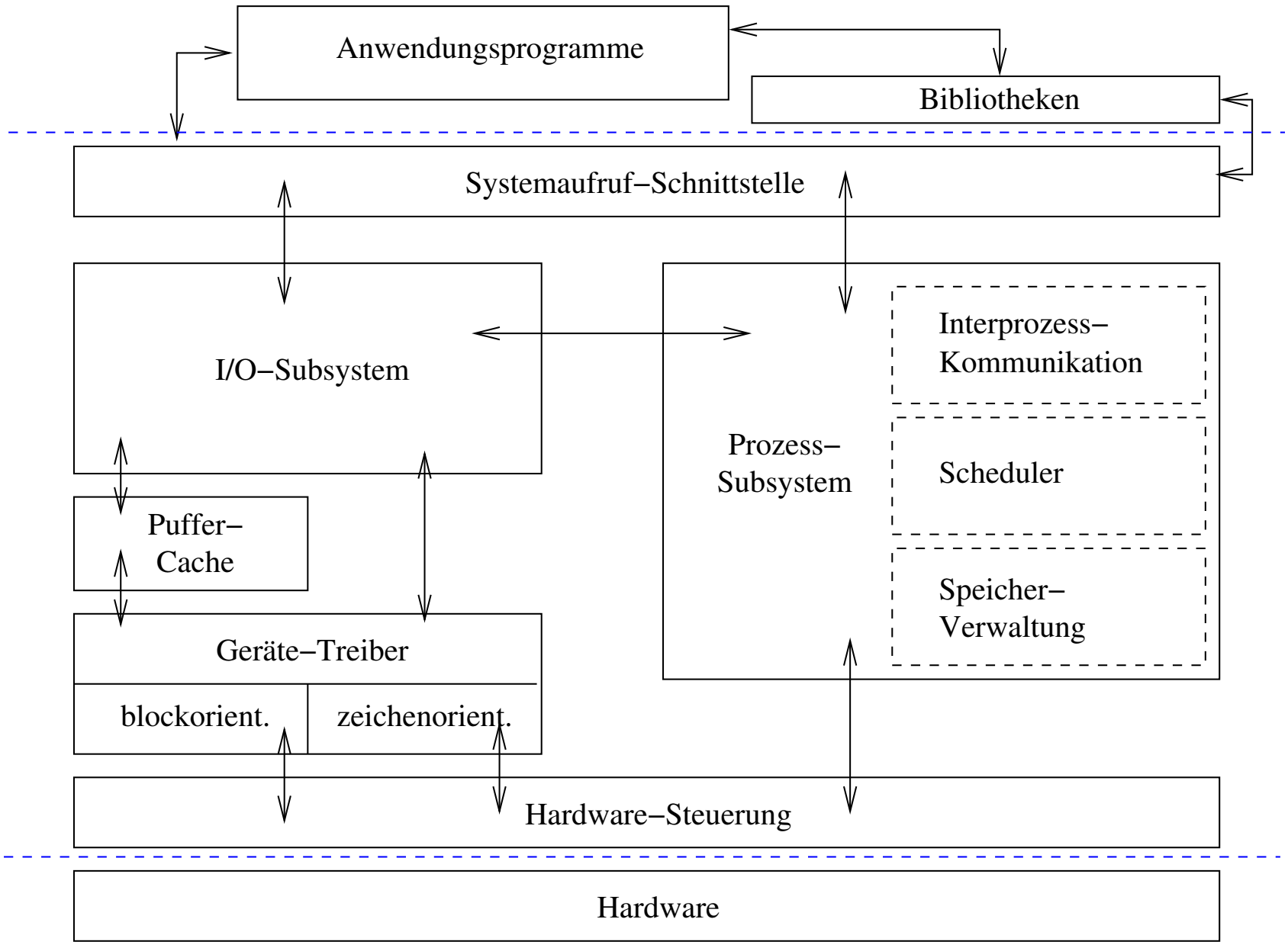

Anwendungs–Software
Editoren
Kommando–Interpreter
Compiler
Bibliotheken
Dienstprogramme
Betriebssystem
Abstrakte Prozessor–Ebene
Microcode / Firmware
Physische Geräte

- **Physische Geräte:**
  Prozessor, Festplatten, Grafikkarte, Stromversorgung, etc.
- **Microcode / Firmware:**
  Software, die die physikalischen Geräte direkt kontrolliert und sich teilweise direkt auf den Geräten befindet. Diese bietet der nächsten Schicht eine einheitlichere Schnittstelle zu den physikalischen Geräten. Dabei werden einige Details der direkten Gerätesteuerung verborgen.

  Beispiel: Abbildung logischer Adressen auf physische Adressen bei Festplatten.
- **Abstrakte Prozessor-Ebene:**
  Schnittstelle zwischen Hard- und Software. Hierzu gehören nicht nur alle Instruktionen des Prozessors, sondern auch die Kommunikationsmöglichkeiten mit den Geräten und die Behandlung von Unterbrechungen.

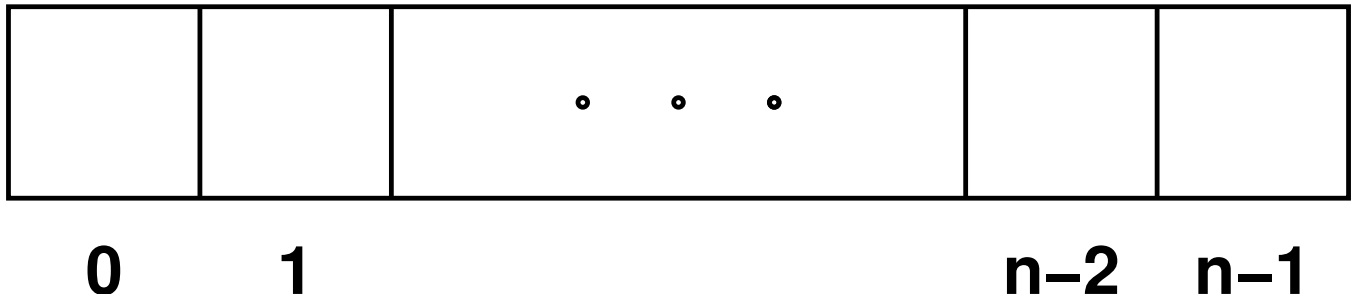
- **System-Software:**
  Software, die von der Schnittstelle des Betriebssystems abhängt und typischerweise vom Hersteller des Betriebssystems mit ausgeliefert wird.

  Beispiele: Bibliotheken (libc.a), Kommandozeilen-Interpreter (Shells), graphische Benutzeroberflächen (X-Windows), systemnahe Werkzeuge, Netzwerkdienste (Web-Server)
- **Anwendungen:**
  Von Benutzern bzw. für Benutzer zur Lösung ihrer Probleme entwickelte Programme *Beispiel:* Textverarbeitungsprogramm


Anwendungsprogramme
Bibliotheken
Systemaufruf–Schnittstelle
I/O–Subsystem
Prozess–
Subsystem
Interprozess–
Kommunikation
Scheduler
Speicher–
Verwaltung
Puffer–
Cache
Geräte–Treiber
blockorient.
zeichenorient.
Hardware–Steuerung
Hardware

Aus IEEE Std 1003.1 (POSIX):

> *An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, symbolic link, socket, and directory. Other types of files may be supported by the implementation.*



- Eine gewöhnliche Datei entspricht einem Array aus Bytes.
- Wenn eine Datei eine Länge von n Bytes hat, sind diese über die Positionen 0 bis $n-1$ abrufbar.
- Eine Dateiverbindung hat eine aktuelle Position p.
- Wenn ein Byte über eine Verbindung gelesen oder geschrieben wird, dann erfolgt der Zugriff auf der aktuellen Position p, die anschließend, falls die Operation erfolgreich war, um eins erhöht wird.
- Lese-Operationen bei einer Position von n sind nicht erfolgreich.

- Unix verlangt und unterstellt bei regulären Dateien *keinerlei Struktur* und unterstützt auch keine.
- Die Konzepte variabel oder konstant langer Datensätze (Records) sind im Kernel von UNIX nicht implementiert.
- Entsprechend sind gewöhnliche Dateien ganz schlichte Byte-Arrays.
- Die einzige Besonderheit ist, dass Dateien unter Unix „Löcher“ haben dürfen, d.h. einzelne Indexbereiche des Arrays können unbelegt sein. Diese werden dann als Nullbytes ausgelesen.

Zu einer Datei gehören

- ein oder auch mehrere *Namen*,
- der *Inhalt* und *Aufbewahrungsort* (Menge von Blöcken auf der Platte, etc.) und
- *Verwaltungsinformationen* (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, etc.).

- Neben den gewöhnlichen Dateien gibt es unter Unix weitere Dateiformen.
- Neben den Verzeichnissen gibt es insbesondere Dateivarianten, die der Interprozess-Kommunikation oder direkten Schnittstelle zu Treibern des Betriebssystems dienen.
- Diese weichen in der Semantik von dem Byte-Array ab und bieten beispielsweise uni- oder bidirektionale Kommunikationskanäle.

- Gerätedateien erlauben die direkte Kommunikation mit einem (das jeweilige Gerät repräsentierenden) Treiber.
- Sie erlauben beispielsweise den direkten Zugriff auf eine Festplatte vorbei an dem Dateisystem.
- Für Gerätedateien gibt es zwei verschiedene Schnittstellen:
  - **Zeichenweise arbeitende Geräte** (*character devices* / *raw devices*):
    Diese Dateien erlauben einen ungepufferten zeichenweisen Lese- und/oder Schreibzugriff.
  - **Blockweise arbeitende Geräte** (*block devices*):
    Diese Dateien erlauben Lese- und Schreibzugriffe nur für vollständige Blöcke. Diese Zugriffe laufen implizit über den Puffer-Cache von Unix.

Auf eine Festplatte kann typischerweise auf drei verschiedene Weisen zugegriffen werden:

- Über ein Dateisystem.
- Über die zugehörige blockweise arbeitende Gerätedatei indirekt über den Puffer-Cache.
- Über die zugehörige zeichenweise arbeitende Gerätedatei.

Intern im Betriebsystem liegt die gleiche Schichtenstruktur der Schnittstellen vor: Zugriffe auf ein Dateisystem werden abgebildet auf Zugriffe auf einzelne Blöcke innerhalb des Puffer-Cache. Wenn der gewünschte Block zum Lesen nicht vorliegt oder ein verändeter Block im Cache zu schreiben ist, dann wird der zugehörige Treiber direkt kontaktiert.

Prinzipiell lassen sich Dateisysteme in vier Gruppen unterteilen:

- *Plattenbasierte Dateisysteme:*
  Die Daten des Dateisystems liegen auf einer lokalen Platte.
- *Netzwerk-Dateisystem:*
  Das Dateisystem wird von einem anderen Rechner über das Netzwerk angeboten. Beispiele: NFS, AFS und Samba.
- *Meta-Dateisysteme:*
  Das Dateisystem ist eine Abbildungsvorschrift eines oder mehrerer anderer Dateisysteme. Beispiele: *tfs* und *unionfs*.
- *Pseudo-Dateisystem:*
  Das Dateisystem ist nicht mit persistenten Daten verbunden. Beispiel: Das *procfs* unter */proc*, das die einzelnen aktuell laufenden Prozesse repräsentiert.

- Gegeben ist die abstrakte Schnittstelle eines Arrays von Blöcken. (Dies kann eine vollständige Platte sein, eine Partition davon oder eine virtuelle Platte, wie sie etwa bei diversen RAID-Verfahren entsteht.)
- Zu den Aufgaben eines plattenbasierten Dateisystems gehört es, ein Array von Blöcken so zu verwalten, dass
  - über ein hierarchisches Namenssystem
  - Dateien (bis zu irgendeinem Maxium) frei wählbarer Länge
  - gespeichert und gelesen werden können.

Aus dem Werk von Marc J. Rochkind, Seite 29, zum Umgang mit einer Schreib-Operation:

> *I've taken note of your request, and rest assured that your file descriptor is OK,*
>
> *I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs.*
>
> *If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then).*
>
> *If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request.*
>
> *You may ask no further questions. Trust me. And thank me for the speedy reply.*

Was passiert, wenn dann mittendrin der Strom ausfällt?

- Blöcke einer Datei oder gar ein Verwaltungsblock sind nur teilweise beschrieben.
- Verwaltungsinformationen stimmen nicht mit den Dateiinhalten überein.

Im Laufe der Zeit gab es mehrere Entwicklungsstufen bei Dateisystemen in Bezug auf die Integrität:

- Im Falle eines Falles muss die Integrität mit speziellen Werkzeugen überprüft bzw. hergestellt werden. Beispiele: Alte Unix-Dateisysteme wie UFS (alt), ext2 oder aus der Windows-Welt die Familie der FAT-Dateisysteme.
- Ein Journalling erlaubt normalerweise die Rückkehr zu einem konsistenten Zustand. Beispiele: Neuere Versionen von UFS, ext3 und reiser3.
- Das Dateisystem ist immer im konsistenten Zustand und arbeitet entsprechend mit Transaktionen analog wie Datenbanken. Hinzu kommen Überprüfungssummen und Selbstheilungsmechanismen (bei redundanten RAID-Verfahren). Beispiele: ZFS, btrfs, ext4, reiser4 und NTFS.

Moderne Dateisysteme wie ZFS und btrfs werden als B-Bäume organisiert:

- B-Bäume sind sortierte und balancierte Mehrwege-Bäume, bei denen Knoten so dimensioniert werden, dass sie in einen physischen Block auf der Platte passen. (Wobei es Verfahren gibt, die mit dynamischen Blockgrößen arbeiten.)
- Entsprechend besteht jeder Block aus einer Folge aus Schlüsseln, Zeigern auf zugehörige Inhalte und Zeiger auf untergeordnete Teilbäume.
- Es werden nie bestehende Blöcke verändert. Stattdessen werden sie zunächst kopiert, angepasst, geschrieben und danach der neue statt dem alten Block verwendet (*copy on write*).
- Alte Versionen können so auch bei Bedarf problemlos erhalten bleiben (*snapshots*).

- Jedes Dateisystem enthält eine Hierarchie der Verzeichnisse.
- Darüber hinaus gibt es auch eine Hierarchie der Dateisysteme.
- Es beginnt mit der Wurzel / und dem die Wurzel repräsentierenden Wurzel-Dateisystem. (Dies ist das erste Dateisystem, das verwendet wird und das auch das Betriebssystem oder zumindest wesentliche Teile davon enthält.)
- Weitere Dateisysteme können bei einem bereits existierenden Verzeichnis eingehängt werden.
- So entsteht eine globale Hierarchie, die sich über mehrere Dateisysteme erstreckt.

```
doolin$ cd /
doolin$ df .
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0    8263277 3705376 4475269    46%    /
doolin$ cd /var
doolin$ df .
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s7    8263277 2002000 6178645    25%    /var
doolin$
doolin$ cd /
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s0      7.9G   3.5G   4.3G    46%    /
doolin$ cd var
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s7      7.9G   1.9G   5.9G    25%    /var
doolin$ cd run
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
swap                   1.6G    48K   1.6G     1%    /var/run
doolin$
```

- In den 70er Jahren (bis einschließlich UNIX Edition VII) hatte ein Unix-Dateisystem einen sehr einfachen Aufbau, bestehend aus
  - dem Boot-Block (reserviert für den Boot-Vorgang oder ohne Verwendung),
  - dem Super-Block (mit Verwaltungsinformationen für die gesamte Platte),
  - einem festdimensionierten Array von Inodes (Verwaltungsinformationen für einzelne Dateien) und
  - einem Array von Blöcken, die entweder für Dateiinhalte oder (im Falle sehr großer Dateien) für Verweise auf weitere Blöcke einer Datei verwendet werden.

- Das heutige UFS (*UNIX file system*) geht zurück auf das von Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler und Robert S. Fabry Anfang der 80er Jahre entwickelte Berkeley Fast File System.
- Gegenüber dem historischen Aufbau enthält es einige wesentliche Veränderungen:
  - Die Verwaltungsinformationen einer Datei und der Dateiinhalt werden so auf der Platte abgelegt, dass sie möglichst schnell hintereinander gelesen werden können.
  - Dazu wird die Platte entsprechend ihrer Geometrie in Zylindergruppen aufgeteilt. Zusammenhängende Inodes und Datenblöcke liegen dann möglichst in der gleichen oder der benachbarten Zylindergruppe.
  - Die Blockgröße wurde vergrößert (von 1k auf 8k) und gleichzeitig wurden für kleine Dateien fragmentierte Blöcke eingeführt.
  - Damit der Verlust des Super-Blocks nicht katastrophal ist, gibt es zahlreiche Sicherungskopien des Super-Blocks an Orten, die sich durch die Geometrie ableiten lassen.
- Das unter Linux lange Zeit populäre ext2-Dateisystem hatte UFS als Vorbild.

- Eine Inode enthält sämtliche Verwaltungsinformationen, die zu einer Datei gehören.
- Jede Inode ist (innerhalb eines Dateisystems) eindeutig über die Inode-Nummer identifizierbar.
- Die Namen einer Datei sind *nicht* Bestandteil der Inode. Stattdessen bilden Verzeichnisse Namen in Inode-Nummern ab.
- U.a. finden sich folgende Informationen in einer Inode:
  - Eigentümer und Gruppe
  - Dateityp (etwa gewöhnliche Datei oder Verzeichnis oder einer der speziellen Dateien)
  - Zeitstempel: Letzter Lesezugriff, letzter Schreibzugriff und letzte Änderung der Inode.
  - Anzahl der Verweise aus Verzeichnissen
  - Länge der Datei in Bytes (bei gewöhnlichen Dateien und Verzeichnissen)
  - Blockadressen (bei gewöhnlichen Dateien und Verzeichnissen)

- In der Unix-Welt gibt es keine standardisierten Systemaufrufe, die ein Auslesen eines Verzeichnisses ermöglichen.
- Der Standard IEEE Std 1003.1 bietet jedoch die Funktionen *opendir*, *readdir* und *closedir* als portable Schnittstelle oberhalb der (nicht portablen) Systemaufrufe an.
- Alle anderen Funktionalitäten (Auslesen des öffentlichen Teils einer Inode, Wechseln des Verzeichnisses und sonstige Zugriffe auf Dateien) sind auch auf der Ebene der Systemaufrufe standardisiert.

dir.c

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   char usage[] = "Usage: %s [directory]\n";
   if (argc > 1) {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
   char* dirname;
   if (argc > 0) {
      dirname = *argv;
   } else {
      dirname = ".";
   }
   /**** Auslesen von dirname ****/
}
```

dir.c

```
if (chdir(dirname) < 0) {
   perror(dirname);
   exit(1);
}
DIR* dir = opendir(".");
if (!dir) {
   perror(dirname);
   exit(1);
}
```

- Mit *chdir()* ist es möglich, das aktuelle Verzeichnis zu wechseln. Dies betrifft den aufrufenden Prozess (und wird später an neu erzeugte Prozesse weiter vererbt).
- *chdir()* wird hier verwendet, um im weiteren Verlauf den Zusammenbau zusammengesetzter Pfade aus dem Verzeichnisnamen und dem darin enthaltenen Dateinamen zu vermeiden.
- Nach dem Aufruf von *chdir()* ist das gewünschte (dann aktuelle) Verzeichnis unter dem Namen . erreichbar.

Auslesen eines Verzeichnisses

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
   struct stat statbuf;
   if (lstat(entry->d_name, &statbuf) < 0) {
      perror(entry->d_name); exit(1);
   }
   if (S_ISREG(statbuf.st_mode)) {
      printf("regular file with %jd bytes\n",
         (intmax_t) statbuf.st_size);
   } else if (S_ISDIR(statbuf.st_mode)) {
      puts("directory");
   } else if (S_ISLNK(statbuf.st_mode)) {
      char buf[1024];
      ssize_t len = readlink(entry->d_name, buf, sizeof buf);
      if (len < 0) {
         perror(entry->d_name); exit(1);
      }
      printf("symbolic link pointing to %.*s\n", len, buf);
   } else {
      puts("special");
   }
}
closedir(dir);
```

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
```

- *readdir* liefert einen Zeiger auf eine (statische) Struktur mit Informationen über die nächste Datei aus dem Verzeichnis.
- Die Struktur mag mehrere systemabhängige Komponenten haben. Relevant und portabel ist jedoch nur der Dateiname in dem Feld *d_name*

dir.c

```
struct stat statbuf;
if (lstat(entry->d_name, &statbuf) < 0) {
   perror(entry->d_name); exit(1);
}
```

- Es gibt mehrere Systemaufrufe, die den öffentlichen Teil einer Inode auslesen können.
- Dazu gehört *lstat*, das einen Dateinamen erhält und dann in dem per Zeiger referenzierten Struktur die gewünschten Informationen aus der Inode ablegt.
- Im Unterschied zu *stat*, das genauso aufgerufen wird, folgt *lstat* nicht implizit symbolischen Links, so dass wir hier die Chance haben, diese als solche zu erkennen.

dir.c

```
    if (S_ISREG(statbuf.st_mode)) {
       printf("regular file with %jd bytes\n",
          (intmax_t) statbuf.st_size);
```

- Das Feld *st_mode* aus der von *lstat()* gefüllten Datenstruktur enthält in kombinierter Form mehrere Informationen über eine Datei:
  - den Dateityp,
  - die Zugriffsrechte (rwx) für Besitzer, Gruppe und den Rest der Welt und
  - eventuelle weitere besondere Attribute wie etwa das Setuid-Bit oder das Sticky-Bit.
- Damit der Zugriff weniger kompliziert ist, gibt es standardisierte Makros im Umgang mit *st_mode*. So liefert etwa *S_ISREG* den Wert **true**, falls es sich um eine gewöhnliche Datei handelt.

dir.c

```
    } else if (S_ISLNK(statbuf.st_mode)) {
       char buf[1024];
       ssize_t len = readlink(entry->d_name, buf, sizeof buf);
       if (len < 0) {
          perror(entry->d_name); exit(1);
       }
       printf("symbolic link pointing to %.*s\n", len, buf);
```

- Wäre *stat()* an Stelle von *lstat()* verwendet worden, würde dieser Fall nie erreicht werden, da normalerweise symbolischen Links implizit gefolgt wird.
- Mit *readlink()* kann der Link selbst ausgelesen werden.
- Das Ziel eines symbolischen Links muss nicht notwendigerweise existieren. Falls das Ziel nicht existiert, liefert *stat()* einen Fehler, während *lstat()* uns unabhängig von der Existenz das Ziel nennt.

- Bei Systemaufrufen sind, soweit sie von Privilegien und/oder einem Zugriffsschutz abhängig sind, folgende u.a. folgende vier Identitäten von Belang:

effektive Benutzernummer	*geteuid()*
effektive Gruppennummer	*getegid()*
reale Benutzernummer	*getuid()*
reale Gruppennummer	*getgid()*

- Normalerweise gleichen sich die effektiven und realen Nummern. Im Falle von Programmen mit dem s-bit werden die effektiven Identitätsnummern von dem Besitzer des Programmes übernommen, während die realen Nummern gleichbleiben.
- In Bezug auf Zugriffe im Dateisystem sind die effektiven Nummern von Belang.

- Zu jeder Inode gehören die elementaren Zugriffsrechte die Lese-, Schreib- und Ausführungsrechte angeben für den Besitzer, die Gruppe und den Rest der Welt.
- Wenn die effektive Benutzernummer die 0 ist, dann ist alles erlaubt (Super-User-Privilegien).
- Falls die effektive Benutzernummer mit der der Datei übereinstimmt, dann sind die Zugriffsrechte für den Besitzer relevant.
- Falls nur die effektive Gruppennummer mit der Gruppenzugehörigkeit der Datei übereinstimmt, dann sind die Zugriffsrechte für die Gruppe relevant.
- Andernfalls gelten die Zugriffsrechte für den Rest der Welt.

- Lese-, Schreib- und Ausführungsrechte haben bei Verzeichnissen besondere Bedeutungen.
- Das Leserecht gibt die Möglichkeit, das Verzeichnis mit *opendir* und *readdir* anzusehen, aber noch *nicht* das Recht, *stat* für eine darin enthaltene Datei aufzurufen.
- Das Ausführungsrecht lässt die Verwendung des Verzeichnisses in einem Pfad zu, der an einem Systemaufruf weitergereicht wird.
- Das Schreibrecht gewährt die Möglichkeit, Dateien in dem Verzeichnis zu entfernen (*unlink*), umzutaufen (*rename*) oder neu anzulegen. Das Ausführungsrecht ist aber eine Voraussetzung dafür.

- Zusätzlich gibt es noch drei weitere Bits:

Set-UID-Bit	Bei einer Ausführung wird die effektive Benutzernummer (UID) gesetzt auf die Benutzernummer des Besitzers.
Set-GID-Bit	Entsprechend wird auch die effektive Gruppennummer (GID) gesetzt. Bei Verzeichnissen bedeutet dies, dass neu angelegte Dateien die Gruppe des Verzeichnisses erben.
Sticky-Bit	Programme mit dem Sticky-Bit bleiben im Speicher. Verzeichnisse mit diesem Bit schränken die Schreibrechte für fremde Dateien ein – nützlich für gemeinsam genutzte Verzeichnisse wie etwa */tmp*.