

Betriebssysteme

Robert Kaiser

(HTTP: <http://www.cs.hs-rm.de/~kaiser>

E-Mail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

10. Dateisysteme



<https://tagebuchdesdeutschen.wordpress.com/2016/08/06/neues-von-der-scheinbehoerden-front-strategiewechsel-der-firma-finanzamt/comment-page-1/>



Anforderungen an informationsverarbeitende Systeme:

- Speichern und Wiederauffinden sehr großer Mengen von Informationen
- Lebensdauer der Informationen länger als die der benutzenden Prozesse (**Persistenz**).
- Gemeinsame Nutzung von Informationen durch Prozesse (**Sharing**).

Definition: Datei, Dateisystem

- Eine **Datei** (*File*) ist eine logische Einheit zur Speicherung von Informationen auf externen Speichermedien.
- **Dateisysteme** (*File Systems*) sind die Teilsysteme eines Betriebssystems, die der Bereitstellung von Dateien dienen

Dateisysteme



- ① Dateien
- ② Verzeichnisse
- ③ Implementierung von Dateisystemen
- ④ Zusammenfassung

UNIX wird beispielhaft in den jeweiligen Abschnitten behandelt.

Dateien



Zunächst: Einführung von Dateien aus Benutzersicht

- 1 Benennung von Dateien
- 2 Dateistrukturen
- 3 Dateitypen
- 4 Dateizugriff
- 5 Dateiattribute
- 6 Dateioperationen
- 7 Memory-Mapped Files

Benennung von Dateien



- Datei als Abstraktion zur Speichern und Lesen von Information auf einem Hintergrundspeicher
- Benutzer muss **nicht** wissen, wie und wo die Information abgelegt wird, noch wie der Hintergrundspeicher (i.d.R. Platte) im Detail funktioniert.
- Dateinamen werden benutzt, um in Dateien abgelegte Information zu identifizieren und wiederaufzufinden:
 - ▶ Namensvergabe erfolgt bei Dateierzeugung durch den erzeugenden Prozess.
 - ▶ Datei und Dateiname bleiben bestehen, auch wenn der Prozess terminiert.
 - ▶ Dateiname kann von anderen Prozessen benutzt werden, um Zugang zu der gespeicherten Information zu bekommen.

Benennung von Dateien (2)



Definition: Dateinamensraum

Der **Dateinamensraum** definiert die Menge der zulässigen Dateinamen.

Regeln für zulässige Dateinamen sind stark systemabhängig:

- Unterscheidung zwischen Groß- und Kleinbuchstaben (ja: UNIX, nein: MS-DOS).
- Länge der zulässigen Dateinamen (UNIX: 255 Zeichen, MS-DOS: 8 Zeichen).
- Verwendung von Sonderzeichen.¹
- Verwendung von Namenserverweiterungen (*File Extensions*):
 - ▶ optional (Regelfall) oder erzwungen.
 - ▶ einfach (z.B. MS-DOS) oder mehrfach (UNIX).
 - ▶ Konventionen für die Verwendung
 - ★ z.B. .c für C-Quelldateien
 - ★ z.T. von verarbeitendem Programmen erzwungen z.B. make

¹UNIX: (fast) alles erlaubt, aber besser vermeiden: Leerzeichen, Umlaute. »

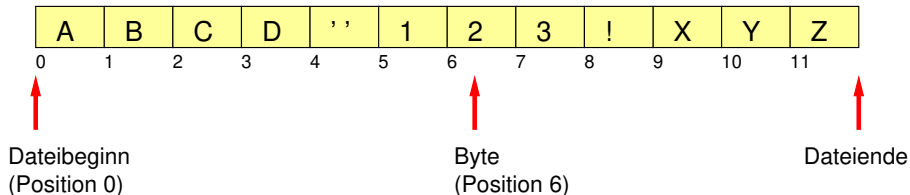
Typische Namenserverweiterungen



Erweiterung	Bedeutung
.bak	Sicherungsdatei
.BAT	MS-DOS „Batch“ Datei
.c	C-Quelltext
.gif	Bilddatei (Compuserve Graphical Image Format)
.html	Hypertext Markup Language
.jpg, .jpeg	Bilddatei nach dem JPEG Standard
.o	Objektdatei (Übersetzt, noch nicht gebunden)
.mp3	Audiodatei
.pdf	Portable Document Format
.sh	UNIX Shell Script
.tar	UNIX Tape Archive
.tar.gz, .tgz	Komprimiertes UNIX Tape Archive
.tex	Eingabe für das TeX-Satzsystem
.txt	ASCII-Textdatei
.zip	Komprimiertes Archiv

N.B.: UNIX: Lediglich Konvention

Dateistrukturen: Bytefolge



- Die Datei wird als **unstrukturierte Folge von Bytes** aufgefasst.
- Das Betriebssystem **weiß nichts** über den Inhalt oder dessen Struktur, sieht eine Datei ausschließlich als Container an.
- Interpretation der Bytefolge durch die zugreifenden Anwendungen
- Vorteil: Maximale Flexibilität.
- Sequenzielle Verarbeitung und/oder wahlfreier Zugriff durch Ansteuern einer Byte-Position

Dateistrukturen: Satz-Dateistruktur



Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Huber	Müller	Nöckel	Scholz	Panik	Hansl
25.1	19.5	11.0	16.2	17.3	18.0	19.2



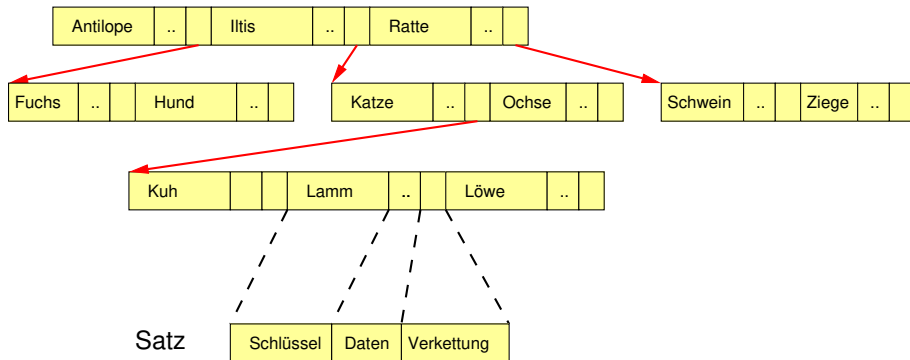
Dateianfang

Satz (Record)

Dateiende

- Die Datei wird als **Folge von Sätzen fester Länge** aufgefasst.
- Das Betriebssystem kennt die Satzlänge, nicht aber die innere Struktur.
- Lese / Schreib / Änderungsoperationen stets nur auf ganze Sätze anwendbar.
- Sätze sind durch ihre Satznummer (durch „Abzählen“) ansteuerbar.
- Ursprung: Dateien entsprechen Lochkartenstapel, d.h. Folge von Sätzen der Länge 80 Zeichen.

Dateistrukturen: Baum von Sätzen



- Die Datei wird als **Baum von Sätzen mit Schlüsselfeld** aufgefasst.
- Betriebssystem sieht die eventuell variabel langen Sätze, das Schlüsselfeld an fester Position im Satz und die Ordnung der Sätze.
- Operationen:
 - Suchen eines Satzes bei gegebenem Schlüsselfeld.
 - Ordnungserhaltendes Hinzufügen eines Satzes.

Dateitypen



UNIX: Zu unterscheiden² ...

- **Gewöhnliche Dateien** (*Regular Files*)

- ▶ ASCII- („Text-“) Dateien: Folge von Textzeilen variabler Länge, i.d.R. durch **betriebsystemabhängiges** Steuerzeichen getrennt (UNIX: „\n“, MacOS: „\r“, Windows: „\r\n“), mit Texteditor manipulierbar.
- ▶ Binärdateien: Alles Andere, verschiedene interne Formate, abhängig von Verwendung, häufig durch sog. „Magic Numbers“ gekennzeichnet, z.B. als ausführbare Dateien.
- ▶ (UNIX: **Keine** Unterscheidung auf Dateisystem-Ebene.)

- **Verzeichnisse** (*Directories*) (→ vgl. 10.2)

- **Zeichenorientierte Spezialdateien** (*Character Special Files*)

- ▶ Repräsentierung serieller E/A-Geräte, z.B. Terminals (z.B. Linux: „/dev/ttyS0“), Drucker, Netzwerk.

- **Blockorientierte Spezialdateien** (*Block Special Files*)

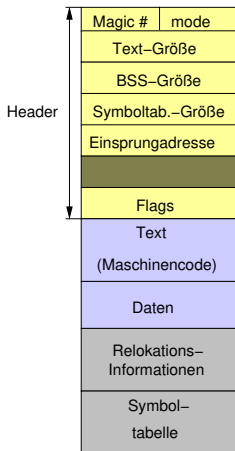
- ▶ Repräsentierung blockorientierter Hintergrundspeichermedien, z.B. Platte (z.B. Linux: „/dev/sda1“), CDROM.

²Tipp: Kommando „file“

Beispiel: UNIX Executable File



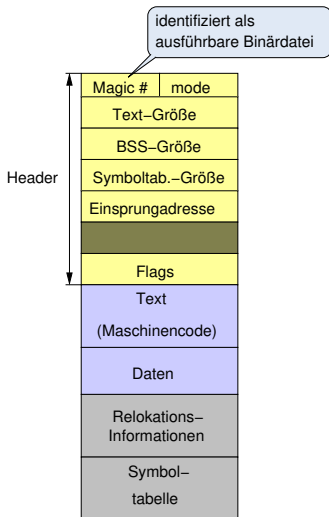
Format einer ausführbaren Datei (a.out, einfacher als ELF):



Beispiel: UNIX Executable File



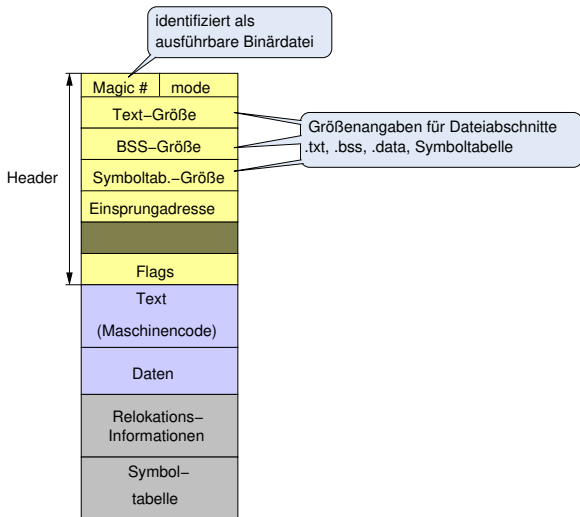
Format einer ausführbaren Datei (a.out, einfacher als ELF):



Beispiel: UNIX Executable File



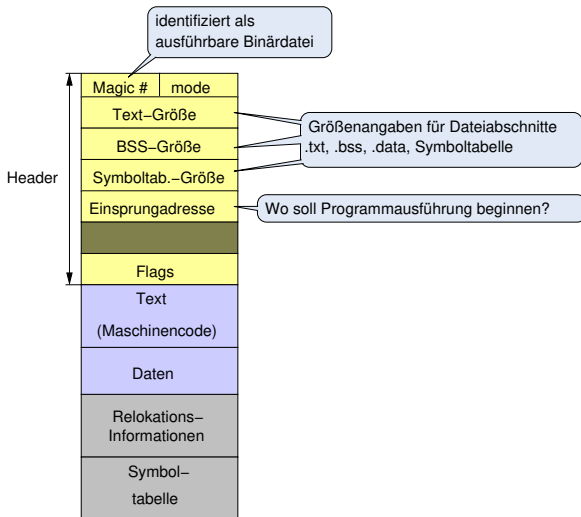
Format einer ausführbaren Datei (a.out, einfacher als ELF):



Beispiel: UNIX Executable File



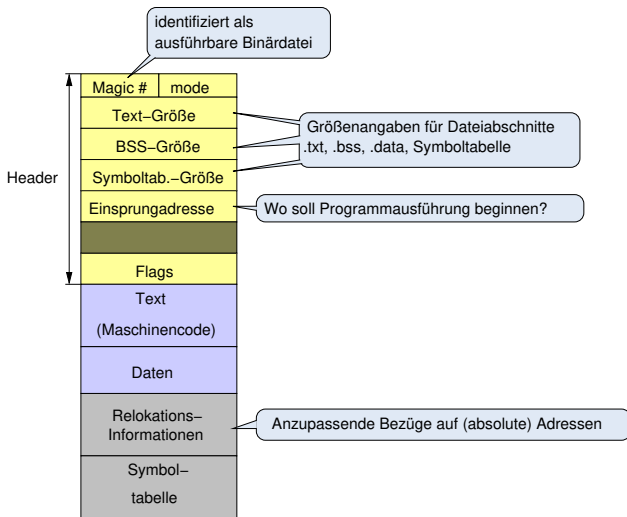
Format einer ausführbaren Datei (a.out, einfacher als ELF):



Beispiel: UNIX Executable File



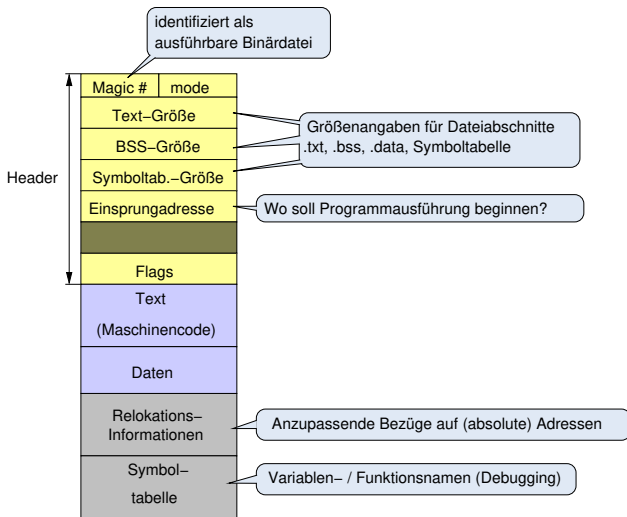
Format einer ausführbaren Datei (a.out, einfacher als ELF):



Beispiel: UNIX Executable File



Format einer ausführbaren Datei (a.out, einfacher als ELF):



Dateizugriff



Sequenzieller Zugriff:

- Verarbeitung beginnend am Dateianfang fortlaufend bis zum Dateiende.
 - „Zurückspulen“ auf den Dateianfang.
- Angemessene Abstraktion für Dateien auf Magnetbändern.

Direkter Zugriff:

- Bytes oder Sätze der Datei können in beliebiger Reihenfolge verarbeitet, gelesen oder geschrieben werden.
- Alternativen:
 - ▶ Jede Operation gibt die Position explizit an.
 - ▶ Positionierbefehl `seek` innerhalb einer Datei, danach sequenzielle Verarbeitung von dieser Position ausgehend.
- In allen neueren Systemen sind Dateien Direktzugriffsdateien.

Dateiattribute



Definition:

Dateiattribute (auch: Metadaten) sind Eigenschaften, die das Betriebssystem mit einer Datei verbindet (und im Dateikontrollblock speichert).

- Menge und Semantik der verwendeten Attribute ist systemabhängig.
- Beispiel UNIX:

Dateikontrollblock heißt *inode* (index node) und enthält u.a.:

- ▶ Dateityp
- ▶ Zugriffsrechte (rwx für owner, group, others (vgl. 10.5))
- ▶ Anzahl Referenzen auf diese Datei (Reference Counter)
- ▶ Eigentümer (user id und group id)
- ▶ Größe (Anzahl bytes)
- ▶ Zeitpunkt des letzten Zugriffs
- ▶ Zeitpunkt der letzten Modifikation
- ▶ Zeitpunkt des letzten Updates dieses Kontrollblocks

In UNIX wird der Dateiname selbst nicht im inode gespeichert, sondern im entsprechenden Verzeichnis (vgl. 10.2, 10.3).

Mögliche Dateiattribute



Attribute	Bedeutung
Schutz	Wer kann wie auf Datei zugreifen?
Passwort	Passwort für den Zugriff
Urheber	ID der Person, die die Datei erzeugt hat
Eigentümer	Aktueller Eigentümer
Read-only-Flag	0: Lesen/Schreiben, 1: nur Lesen
Hidden-Flag	0: normal, 1: in Listen nicht sichtbar
System-Flag	0: normale Datei, 1: Systemdatei
Archiv-Flag	0: wurde gesichert, 1: muss noch gesichert werden
ASCII/Binär-Flag	0: ASCII-Datei, 1: Binärdatei
Random-Access-Flag	0: nur sequenzieller Zugriff, 1: wahlfreier Zugriff
Temporary Flag	0: normal, 1: Datei bei Prozessende löschen
Sperr-Flags	0: nicht gesperrt, nicht null: gesperrt
Datensatzlänge	Anzahl der Bytes in einem Datensatz
Schlüsselposition	Offset des Schlüssels innerhalb des Datensatzes
Schlüssellänge	Anzahl der Bytes im Schlüsselfeld
Erstellungszeit	Datum und Zeitpunkt der Dateierstellung
Zeitpunkt des letzten Zugriffs	Datum und Zeitpunkt des letzten Zugriffs
Zeitpunkt der letzten Änderung	Datum und Zeitpunkt der letzten Dateiänderung
Aktuelle Größe	Anzahl der Bytes in der Datei
Maximale Größe	Anzahl der Bytes für maximale Größe der Datei

Dateioperationen



Typische Dateioperationen

- Create – (Leere) Datei erzeugen
- Open – Datei vor Benutzung öffnen
- Append – Daten am Ende der Datei anfügen
- Delete – Datei löschen
- Close – Datei nach Benutzung schließen (noch ausstehende Schreibvorgänge ausführen)
- Read – Daten ab aktueller Position lesen
- Write – Daten ab aktueller Position schreiben
- Seek – Aktuelle Position setzen
- Get Attributes – Dateiattribute lesen
- Set Attributes – Dateiattribute setzen
- Rename – Dateinamen ändern

UNIX: Create



```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int creat(const char *path, mode_t mode);
```

Oder (äquivalent):

```
int open(const char *path, O_CREAT|O_WRONLY|O_TRUNC, mode_t mode);
```

Erzeugt neue Datei mit Namen `path` und liefert einen Dateideskriptor als Ergebnis (-1 bei Fehler). Symbolische Werte für `mode`:

Symbol	Oktalzahl	Bedeutung
S_IRUSR	0400	Benutzer hat Leserechte
S_IWUSR	0200	Benutzer hat Schreibrechte
S_IXUSR	0100	Benutzer hat Ausführrechte
S_IRWXG	0070	Gruppe hat Lese-, Schreib- und Ausführrechte
S_IRGRP	0040	Gruppe hat Leserechte
S_IWGRP	0020	Gruppe hat Schreibrechte
S_IXGRP	0010	Gruppe hat Ausführrechte
S_IRWXO	0070	andere haben Lese-, Schreib- und Ausführrechte
S_IROTH	0004	andere haben Leserechte
S_IWOTH	0002	andere haben Schreibrechte
S_IXOTH	0001	andere haben Ausführrechte

`mode` wird mit der Prozess-Dateierzeugungsmaske `umask` zur Festlegung der Zugriffsrechte so verknüpft, dass Zugriffsrechte für gesetzte `umask`-Bits weggelassen werden.

UNIX: Open, Append



```
#include <fcntl.h>
```

```
int open(const char *path, int flags, mode_t mode);  
int open(const char *path, int flags);
```

Öffnet Datei mit Namen `path` und liefert einen Dateideskriptor als Ergebnis (-1 bei Fehler). `flags` – Wie soll die Datei geöffnet werden? (→ `fcntl.h`). Mögliche Werte:

- `O_RDONLY` = nur lesen
- `O_WRONLY` = nur schreiben
- `O_RDWR` = lesen und schreiben

Per Bit-ODER („|“) können verschiedene Flags hinzugefügt werden:

Flag	Bedeutung
<code>O_APPEND</code>	Schreibzugriffe: am Dateiende anhängen
<code>O_CREAT</code>	Datei anlegen, falls noch nicht vorhanden (in diesem Fall Argument <code>mode</code> erforderlich)
<code>O_EXCL</code>	(mit <code>O_CREAT</code>) Fehler, falls Datei schon da
<code>O_TRUNC</code>	löscht Dateiinhalt, falls schon vorhanden
<code>O_NDELAY</code>	erlaubt nicht-blockierende E/A-Operationen
<code>O_SYNC</code>	sofortiges Zurückschreiben bei Schreiboperationen

Behandlung von Argument `mode`: s.o. (`creat()`)

UNIX: Delete, Close



```
#include <unistd.h>

int unlink(const char *path);
```

Löscht den Namen `path` aus dem Dateisystem. Falls dieser Name der letzte Link auf eine Datei war und kein Prozess die Datei geöffnet hält, wird sie gelöscht.

```
#include <unistd.h>

int close(int fd);
```

Schließt den Dateideskriptor `fd`, so dass dieser nicht mehr zu einer Datei gehört und wieder verwendet werden kann.

UNIX: Read, Write



```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Liest **bis zu** count Byte aus dem Dateideskriptor fd in den bei buf beginnenden Puffer. Rückgabewert: Anzahl **tatsächlich gelesener** Bytes (0 bedeutet Dateieinde), oder -1 bei Fehler.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Schreibt **bis zu** count Byte aus dem bei buf beginnenden Puffer in die Datei, auf die der Dateideskriptor fd weist. Rückgabewert: Anzahl **tatsächlich geschriebener** Bytes, oder -1 bei Fehler.

UNIX: Seek



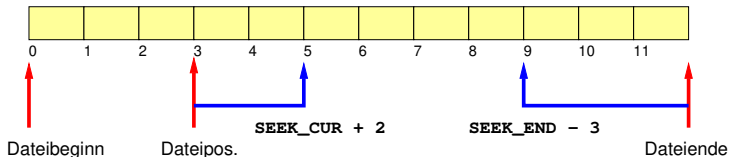
```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

`lseek()` positioniert die aktuelle Dateiposition von `fd` auf den Wert `offset` gemäß der Einstellung von `whence`. `offset` darf auch negativ sein. Mögliche Werte für `whence`:

Symbol	Bedeutung
SEEK_SET	neue Position wird auf <code>offset</code> gesetzt
SEEK_CUR	neue Position ist aktuelle Position + <code>offset</code>
SEEK_END	neue Position ist Dateiende + <code>offset</code>

Rückgabewert: neue Position (ab Anfang gezählt) oder -1 bei Fehler



UNIX: Get Attributes



```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Liefert Informationen zu der durch path angegebenen Datei in der angegebenen stat-Struktur (-1 bei Fehler). Identisch: fstat bei bereits geöffneter Datei.

```
struct stat {
    __dev_t      st_dev;           /* Geraet */
    __ino_t      st_ino;          /* Inode */
    __mode_t     st_mode;         /* Dateityp und -modus */
    __nlink_t    st_nlink;        /* Anzahl harter Links */
    __uid_t      st_uid;          /* UID des Besitzers */
    __gid_t      st_gid;          /* GID des Besitzers */
    __dev_t      st_rdev;         /* Typ (wenn Inode-Geraet) */
    __off_t      st_size;         /* Groesse in Bytes */
    __blksize_t  st_blksize;      /* Blockgroesse fuer Dateisystem-E/A */
    __blkcnt_t   st_blocks;       /* Anzahl der zugewiesenen 512B-Blocke */
    __time_t     st_atime;        /* Zeit des letzten Zugriffs */
    __time_t     st_mtime;        /* Zeit der letzten Veraenderung */
    __time_t     st_ctime;        /* Zeit der letzten Statusaenderung */
};
```

UNIX: Set Attributes



```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Ändert den Modus der angegebenen Datei, deren Pfadname in `path` übergeben wird, zu `mode`. Identisch: `fchmod` bei bereits geöffneter Datei.

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Ändert den Eigentümer und Gruppenzugehörigkeit der angegebenen Datei, deren Pfadname in `path` übergeben wird. Identisch: `fchown` bei bereits geöffneter Datei. ID-Nummern für Eigentümer (`uid`) und Gruppe (`gid`) stehen z.B. in der Datei `/etc/passwd`; Angabe von `-1`: keine Änderung.

```
#include <sys/types.h>
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *zeiten);
```

Zugriffs- und Modifikationszeitpunkte setzen.

UNIX: Rename



```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

Benennt eine Datei um und verschiebt sie in ein anderes Verzeichnis, wenn nötig. Alle anderen Hard Links sind nicht betroffen, ebenso offene Dateideskriptoren für oldpath.

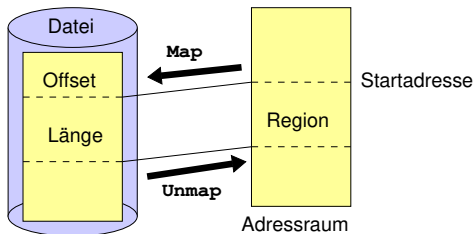
Memory-Mapped Files



Definition: Memory Mapping

Memory-Mapping von Dateien bezeichnet das Einblenden und Ausblenden von Dateien (oder Teil-Fenstern) in den Adressraum eines Prozesses.

- Abstrakte Operationen: Map und Unmap



- Vorteil: Zugriff auf Dateien mit normalen Speicherbefehlen, kein `read()` und `write()` erforderlich.
- In neueren Systemen in Zusammenhang mit Virtual Memory Management zum Transport der Seiten der Datei realisiert.

UNIX: Map, Unmap



```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Blendet `length` Bytes von Datei mit Dateideskriptor `fd` ab Position `offset` in den Adressraum des aktuellen Prozesses ab Adresse `addr` ein. Falls `addr == NULL` → System wählt Adresse selbst.

`prot` = Zugriffsart
mögliche Werte:

`flags`: weitere Optionen
Auswahl möglicher Werte:

Flag	Bedeutung
<code>PROT_EXEC</code>	Ausführen
<code>PROT_READ</code>	Lesen
<code>PROT_WRITE</code>	Schreiben
<code>PROT_NONE</code>	Kein Zugriff

Flag	Bedeutung
<code>MAP_SHARED</code>	Änderungen für andere Prozesse sichtbar
<code>MAP_PRIVATE</code>	Änderungen nicht sichtbar
<code>MAP_ANONYMOUS</code>	Keine Hintergrunddatei
<code>MAP_FIXED</code>	Keine Adressauswahl durch das BS
...	...

Rückgabewert: Virtuelle Adresse des Bereiches, `(void*)-1` bei Fehler.

```
int munmap(void *addr, size_t length);
```

Aufheben des Mappings

Beispiel: Direktzugriff




- struct-Typ beschreibt Messwerte-Datensatz:

```
typedef struct mw {  
    ^^Ichar ableser[20];  
    ^^Ifloat temperatur;  
} Messwert;  
^^I^^I^^I
```

Ziel:

- Speichern der Messwerte der jeweils letzten 7 Tage (rollierend)
- Dateiposition aus Wochentag (0=Sonntag, 1=Montag, ...)
- Direkter Zugriff über Wochentag-Nummer

```
enum { SONNTAG, MONTAG, .. SAMSTAG } wt;
```



Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Huber	Müller	Nöckel	Scholz	Panik	Hansl
25.1	19.5	11.0	16.2	17.3	18.0	19.2

Beispiel: Direktzugriff (2)



```
int speichern(int fd, Messwert *pm, int tag) {  
    ^^Iif(lseek(fd, tag*sizeof(Messwert), SEEK_SET) < 0) {  
    ^^I^^Ipperror("speichern (lseek)");  
    ^^I^^Ireturn -1;  
    ^^I}  
    ^^Iif(write(fd, pm, sizeof(Messwert)) < 0) {  
    ^^I^^Ipperror("speichern (write)");  
    ^^I^^Ireturn -1;  
    ^^I}  
    ^^Ireturn 0;  
}  
^^I^^I^^I
```

```
main() {  
    ^^IMesswert m;  
    ^^Ienum { SONNTAG, MONTAG, DIENSTAG, MITTWOCH, ...};  
    ^^I...  
    ^^Ifd = open("messwerte.dat", O_RDWR | O_CREAT, 0644);  
    ^^Ierr1 = speichern(fd, &m, SONNTAG);  
    ^^Ierr2 = lesen(fd, &m, MONTAG); /* wie speichern() */  
    ^^I...  
}
```

Beispiel: Mmap



```
...
int main(void) {
    int fd, laenge, i;
    Messwert *pmw; /* s.o.: "Direktzugriff" */
    fd = open("messwerte.dat", O_RDWR, 0644);
    laenge = lseek(fd, 0, SEEK_END);

    pmw = (Messwert*)mmap(0, laenge, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);
    if((Messwert*)-1 == pmw) {
        perror("mmap()");
        exit(EXIT_FAILURE);
    }
    for(i=0; i < 3; i++) {
        printf("Ableser %s: %f Grad\n", pmw[i].ableser, pmw[i].temperatur);
        pmw[i].temperatur = pmw[i].temperatur * 2;
    }
    munmap(pmw, laenge);
    return 0;
}
```

Verzeichnisse



Definition: Verzeichnis

Ein **Verzeichnis** (Ordner, Directory) dient der Strukturierung der Dateimenge eines Dateisystems und besteht aus einer Menge von Einträgen für Dateien und i.d.R. für weitere Verzeichnisse (Unterverzeichnisse).

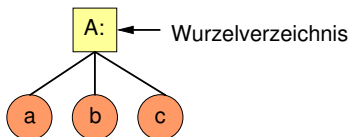
- 1 Verzeichnissysteme mit einer Ebene
- 2 Hierarchische Verzeichnissysteme
- 3 Pfadnamen
- 4 Verzeichnisoperationen
- 5 Montierbare Verzeichnisbäume

Verzeichnissysteme mit einer Ebene



Einfachste Form:

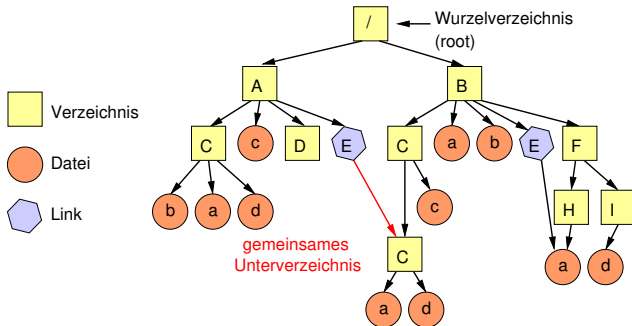
- Alle Dateien in einem (Wurzel-)Verzeichnis
- Varianten:
 - ▶ Ein Verzeichnis pro Laufwerk
 - ▶ Ein Verzeichnis pro Benutzer:In
- Früher in Single User Systemen (PCs) gebräuchlich
 - ▶ CP/M, erste MS-DOS Versionen
- Vorteil: Einfach
- Nachteil: Nur bei überschaubarer Anzahl Dateien brauchbar



Hierarchische Verzeichnissysteme



- Verzeichnisbaum für jeden Benutzer zur Strukturierung seiner Dateimenge (heute Standard).
- Darüber hinaus Einführung zusätzlicher **symbolischer Links** zur flexiblen gemeinsamen Benutzung von Dateimengen. Dateisystemstruktur wird damit zu einem **azyklischen Graphen**.



Pfadnamen



Definition: Pfadnamen

Für hierarchische Verzeichnissysteme dienen **Pfadnamen** (*Path Names*) der Benennung von Dateien und Verzeichnissen.

Alternativen

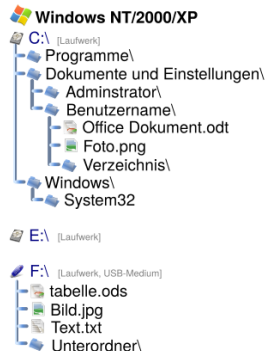
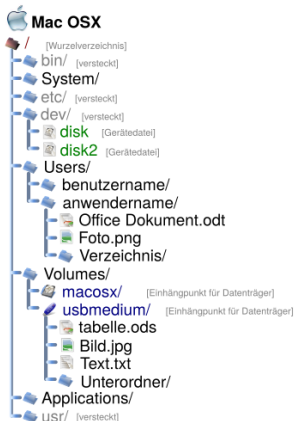
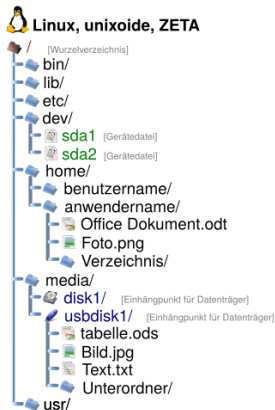
● Absolute Pfadnamen:

- ▶ UNIX: „/“ Name des Wurzelverzeichnisses
- ▶ Benennung vom Wurzelverzeichnis aus.
- ▶ UNIX: /usr/egon/uebung/mist.
- ▶ Beispiel MS-DOS: „\“ als Separator.

● Relative Pfadnamen:

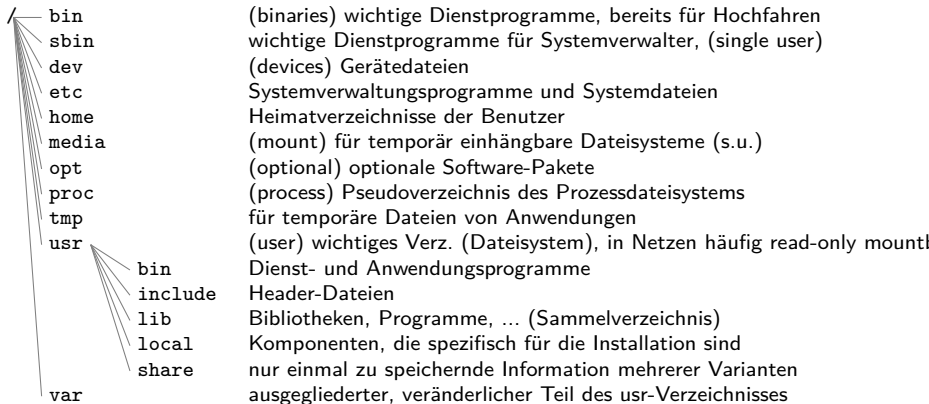
- ▶ Namen werden von einem Arbeitsverzeichnis ausgehend interpretiert (akt. Verzeichnis, Working oder Current Directory).
- ▶ „.“ bezeichnet häufig das aktuelle Verzeichnis selbst.
- ▶ „..“ bezeichnet häufig das Eltern-Verzeichnis (*Parent Directory*).

Typische System-Verzeichnisstrukturen



<https://de.wikipedia.org/w/index.php?curid=4098994>

Linux Filesystem Hierarchy Standard (FHS)



Verzeichnisoperationen



Typische Verzeichnisoperationen

- Create – Verzeichnis erzeugen
- Delete – Verzeichnis löschen
- Chgdir – Verzeichnis wechseln
- Opendir – Verzeichnis zum Lesen von Verzeichniseinträgen öffnen
- Closedir – Verzeichnis nach Benutzung schließen
- Readdir – Nächsten Verzeichniseintrag lesen
- Rename – Verzeichnis umbenennen
- Link – Verzeichniseintrag für bestehende Datei erzeugen
- Unlink – Verzeichniseintrag (und ggf Datei) löschen

UNIX: Create, Delete, Changedir



```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int mkdir(char *path, mode_t mode);
int rmdir(char *path);
int chdir(char *path);
```

- `mkdir()` legt das Verzeichnis `path` mit Zugriffsrechten `mode` und Einträgen für „.“ und „..“ an.
- `rmdir()` löscht das (bis auf die Einträge „.“ und „..“ leere!) Verzeichnis `path`
- `chdir()` setzt das aktuelle Verzeichnis **für den ausführenden Prozess** auf `path`

Wieso ist der Verweis-Zähler für ein Verzeichnis mindestens 2?

```
$ mkdir beispiel
```

```
$ ls -l
```

```
drwx----- 2 kaiser profs 4096 Apr 20 15:00 beispiel
```

UNIX: Opendir, Closedir, Readdir



```
#include <dirent.h>
#include <sys/types.h>

DIR *opendir(char *path);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
```

- opendir() öffnet das Verzeichnis path und gibt einen Zeiger auf DIR zurück (NULL bei Fehler)
- closedir() schließt ein Verzeichnis; Ergebnis ist 0 (ok) oder -1 (Fehler)
- readdir() liefert den jeweils nächsten Verzeichniseintrag. Bei Ende oder Fehler wird der NULL-Zeiger geliefert.

struct dirent enthält ein Feld `char d_name[]` mit dem Namen des betreffenden Verzeichniseintrags:

```
struct dirent {
    ^^ino_t          d_ino;          /* Inode-Nummer */
    ^^loff_t         d_off;          /* besser nicht verwenden, (s. manpage) */
    ^^unsigned short d_reclen;        /* Laenge dieses Datensatzes */
    ^^unsigned char  d_type;          /* Dateityp; nicht immer unterstuetzt */
    ^^Ichar          d_name[256];    /* Null-terminierter Dateiname */
};
```

Beispiel: mini-ls



Ziel: So etwas ...

```
$ ./a.out /etc
[/etc/sysconfig]
[/etc/X11]
/etc/fstab (1355 Bytes)
/etc/mtab (413 Bytes)
/etc/modules.conf (1049 Bytes)
/etc/csh.cshrc (561 Bytes)
/etc/bashrc (1497 Bytes)
/etc/gnome-vfs-mime-magic (8042 Bytes)
[/etc/profile.d]
/etc/csh.login (409 Bytes)
/etc/exports (2 Bytes)
/etc/filesystems (51 Bytes)
/etc/group (601 Bytes)
/etc/host.conf (17 Bytes)
/etc/hosts.allow (161 Bytes)
/etc/hosts.deny (347 Bytes)
...
^^I^^I^^I
```

Beispiel: mini-ls (2)



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

int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *eintrag;
    struct stat statbuf;
    char pfadpuffer[PATH_MAX], *pfadp;

    if (argc != 2) {
        printf("Aufruf: %s verzeichnis\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    dir = opendir(argv[1]);
    if (dir == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    ~I~I~I
```

Beispiel: mini-ls (3)



```
^^Istrcpy(pfadpuffer, argv[1]);
^^Istrcat(pfadpuffer, "/");
^^Ipfadp = pfadpuffer + strlen(pfadpuffer);
^^Iwhile (1) {
^^I^^Ieintrag = readdir(dir);
^^I^^Iif (eintrag == NULL) break;
^^I^^Iif (strcmp(eintrag->d_name, ".")==0 ||
^^I^^I^^Istrcmp(eintrag->d_name, "..")==0) continue;
^^I^^I^^Istrcpy(pfadp, eintrag->d_name);

^^I^^Iif (stat(pfadpuffer, &statbuf) == -1) {
^^I^^I^^Ierror("stat()");
^^I^^I} else if (S_ISDIR(statbuf.st_mode)) {
^^I^^I^^Iprintf("[%s]\n", pfadpuffer);
^^I^^I} else {
^^I^^I^^Iprintf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
^^I^^I}
^^I}
^^Iclosedir(dir);
^^Ireturn EXIT_SUCCESS;
}
^^I^^I^^I
```

Beispiel: mini-ls (3)

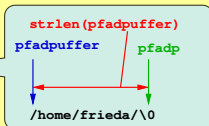


```

^^Istrcpy(pfadpuffer, argv[1]);
^^Istrcat(pfadpuffer, "/");
^^Ipfadp = pfadpuffer + strlen(pfadpuffer);
^^Iwhile (1) {
^^I^^Ieintrag = readdir(dir);
^^I^^Iif (eintrag == NULL) break;
^^I^^Iif (strcmp(eintrag->d_name, ".")==0 ||
^^I^^I^^Istrcmp(eintrag->d_name, "..")==0) continue;
^^I^^I^^Istrcpy(pfadp, eintrag->d_name);

^^I^^Iif (stat(pfadpuffer, &statbuf) == -1) {
^^I^^I^^Ierror("stat()");
^^I^^I} else if (S_ISDIR(statbuf.st_mode)) {
^^I^^I^^Iprintf("[%s]\n", pfadpuffer);
^^I^^I} else {
^^I^^I^^Iprintf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
^^I^^I}
^^I}
^^Iclosedir(dir);
^^Ireturn EXIT_SUCCESS;
}
^^I^^I^^I

```



Beispiel: mini-ls (3)

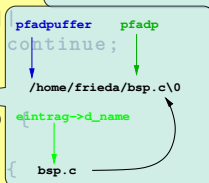
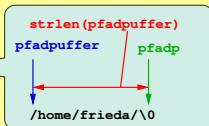


```

^^Istrcpy(pfadpuffer, argv[1]);
^^Istrcat(pfadpuffer, "/");
^^Ipfadp = pfadpuffer + strlen(pfadpuffer);
^^Iwhile (1) {
^^I^^Ieintrag = readdir(dir);
^^I^^Iif (eintrag == NULL) break;
^^I^^Iif (strcmp(eintrag->d_name, ".")==0 |
^^I^^I^^Istrcmp(eintrag->d_name, "..")==0)
^^I^^I^^Icontinue;
^^I^^I^^Istrcpy(pfadp, eintrag->d_name);

^^I^^Iif (stat(pfadpuffer, &statbuf) == -1)
^^I^^I^^Ierror("stat()");
^^I^^I} else if (S_ISDIR(statbuf.st_mode))
^^I^^I^^Iprintf("[%s]\n", pfadpuffer);
^^I^^I} else {
^^I^^I^^Iprintf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
^^I^^I}
^^I}
^^Iclosedir(dir);
^^Ireturn EXIT_SUCCESS;
}
^^I^^I^^I

```



Beispiel: mini-ls (3)



```

^^Istrcpy(pfadpuffer, argv[1]);
^^Istrcat(pfadpuffer, "/");
^^Ipfadp = pfadpuffer + strlen(pfadpuffer);
^^Iwhile (1) {
^^I^^Ieintrag = readdir(dir);
^^I^^Iif (eintrag == NULL) break;
^^I^^Iif (strcmp(eintrag->d_name, ".")==0 |
^^I^^I^^Istrcmp(eintrag->d_name, "..")==0)
^^I^^I^^Icontinue;
^^I^^I^^Istrcpy(pfadp, eintrag->d_name);

^^I^^Iif (stat(pfadpuffer, &statbuf) == -1)
^^I^^I^^Ierror("stat()");
^^I^^I} else if (S_ISDIR(statbuf.st_mode))
^^I^^I^^Iprintf("[%s]\n", pfadpuffer);
^^I^^I} else {
^^I^^I^^Iprintf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
^^I^^I}
^^I}
^^Iclosedir(dir);
^^Ireturn EXIT_SUCCESS;
}
^^I^^I^^I

```

Diagram 1: Calculation of the full path. A red arrow labeled `strlen(pfadpuffer)` points from the end of `pfadpuffer` to the end of `pfadp`. A green arrow labeled `pfadp` points to the resulting path `/home/frieda/\0`.

Diagram 2: Directory entry processing. A green arrow labeled `pfadp` points to `/home/frieda/bsp.c\0`. A green arrow labeled `eintrag->d_name` points to `bsp.c`. A curved arrow indicates the concatenation of the path and the directory name.

Diagram 3: File type check. A blue box contains the code `S_ISDIR(m)=TRUE` and `⇒ Ist Verzeichnis`, which is linked to the `S_ISDIR(statbuf.st_mode)` condition in the code.

UNIX: Rename, Link, Unlink



```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);

#include <unistd.h>

int link(char *oldpath, char *newpath);
int unlink(char *path);
```

- `rename()` Benennt ein Verzeichnis um und verschiebt es in ein anderes Verzeichnis, wenn nötig.
 - ▶ kein Unterschied zu Dateioperation `rename()` (s.o.)
 - ▶ entsprechendes shell-Kommando: `mv oldpath newpath`
- `link()` legt einen neuen Verzeichniseintrag `newpath` an, der auf dieselbe Datei („inode“) wie der bestehende `oldpath` verweist, und erhöht den Referenzzähler im inode. (**Keine** Datei-Kopie!)
 - ▶ entsprechendes shell-Kommando: `ln oldpath newpath`
- `unlink()` erniedrigt den Referenzzähler im zugehörigen inode und löscht den Verzeichniseintrag, sowie die Datei, falls der Referenzzähler auf 0 gefallen ist.
 - ▶ kein Unterschied zu Dateioperation `unlink()` (s.o.)
 - ▶ entsprechendes shell-Kommando: `rm path`

Beispiel: ls -li



ls -l zeigt Anzahl der Verweise (Links) auf den inode einer Datei:

```
$ ls -l /usr/bin/  
...  
-rwxr-xr-x 2 root root 183136 Jul 30 2019 unzip  
...  
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zegrep  
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zfgrep  
-rwxr-xr-x 1 root root 2080 Jan 6 2019 zforce  
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zgrep  
-rwxr-xr-x 1 root root 213136 Aug 16 2015 zip  
-rwxr-xr-x 2 root root 183136 Jul 30 2019 zipinfo  
^^I^^I
```

Option „-li“ zeigt zusätzlich die inode-Nummer → so werden verschiedene Verweise auf denselben inode erkennbar:

```
$ ls -li /usr/bin/z*grep  
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zegrep  
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zfgrep  
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zgrep  
155628 -rwxr-xr-x 1 root root 2953 Jul 30 2019 /usr/bin/zipgrep  
^^I^^I
```

- zegrep, zfgrep und zegrep sind hier **dieselbe Datei**
- Unterscheidung der Funktion anhand argv[0]

UNIX: Symbolische Links



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

```
int symlink(const char *oldpath, const char *newpath);  
int readlink(const char *path, char *buf, size_t bufsiz);
```

- `symlink()` legt einen **symbolischen** Verweis `newpath` an, der auf `oldpath` verweist.
 - ▶ entsprechendes shell-Kommando: `ln -s oldpath newpath`
- `readlink()` liest den Verweis aus Symlink `path` in den Zeichenvektor `buf` (max. `bufsiz` Zeichen).
- (Für beide gilt: Ergebnis 0 für „ok“, -1 für Fehler.)

Unterschied zu bisher betrachteten *Hard Links*:

- Verweis erfolgt **per Namen**, nicht inode (ändert Referenzzähler nicht).
Konsequenzen:
 - ▶ Auflösung zur Laufzeit nötig, evtl. mehrstufig
 - ▶ Verweise über Filesystem- und Partitions Grenzen hinweg möglich (warum geht das mit harten Links nicht?)
 - ▶ Symbolische Links können „ins Leere“ verweisen (Ziel existiert nicht)

Beispiel: Symbolische Links: ls -li



```
$ ls -li /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -li /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -li /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -li /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -li /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
^^I
```

- Verweisziel wird bei symbolischen Links von ls angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf /usr/lib/cpp führt *letztlich* auf /usr/bin/x86_64-linux-gnu-cpp-8

Beispiel: Symbolische Links: ls -li



```
$ ls -li /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -li /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -li /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -li /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -li /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
^^I
```

- Verweisziel wird bei symbolischen Links von ls angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf /usr/lib/cpp führt *letztlich* auf /usr/bin/x86_64-linux-gnu-cpp-8

Beispiel: Symbolische Links: ls -li



```
$ ls -li /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -li /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -li /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -li /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -li /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
^^I
```

- Verweisziel wird bei symbolischen Links von ls angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf /usr/lib/cpp führt *letztlich* auf /usr/bin/x86_64-linux-gnu-cpp-8

Beispiel: Symbolische Links: ls -li



```
$ ls -li /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -li /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -li /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -li /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -li /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
^^I
```

- Verweisziel wird bei symbolischen Links von ls angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf /usr/lib/cpp führt *letztlich* auf /usr/bin/x86_64-linux-gnu-cpp-8

Beispiel: Symbolische Links: ls -li



```
$ ls -li /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -li /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -li /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -li /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -li /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
^^I
```

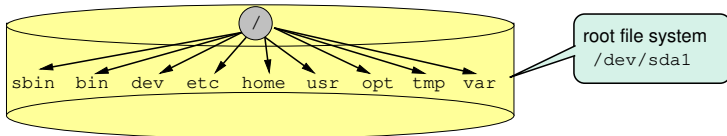
- Verweisziel wird bei symbolischen Links von ls angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf /usr/lib/cpp führt *letztlich* auf /usr/bin/x86_64-linux-gnu-cpp-8

Montierbare Verzeichnisbäume

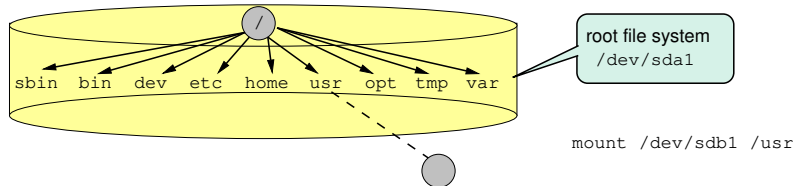


- Der sichtbare Dateiraum ist i.d.R. durch mehrere Dateisysteme auf mehreren Speichern (u.U. auf mehreren Rechnern) realisiert.
- ⇒ Mehr oder weniger sichtbare Auswirkungen auf den Dateinamensraum.
- Alternativen:
 - ▶ Dateinamen spiegeln die verschiedenen Speicher wider.
 - ★ Beispiel: Laufwerksbuchstaben in Windows:
C:\WINDOWS\system, H:\setup.exe
 - ▶ Transparenz im Dateinamensraum.
 - ★ Der gesamte Dateiraum ist aus mehreren Dateisystemen mit jeweils eigenem Verzeichnisbaum durch **Montieren** (*mount*) zusammengesetzt.
 - ★ Nach dem Montieren existiert ein einziger Dateinamensbaum.
 - ★ Beispiel: UNIX, Kommandos: `mount` / `umount`

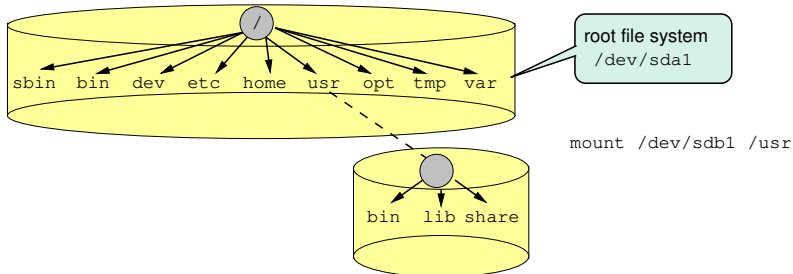
Beispiel:



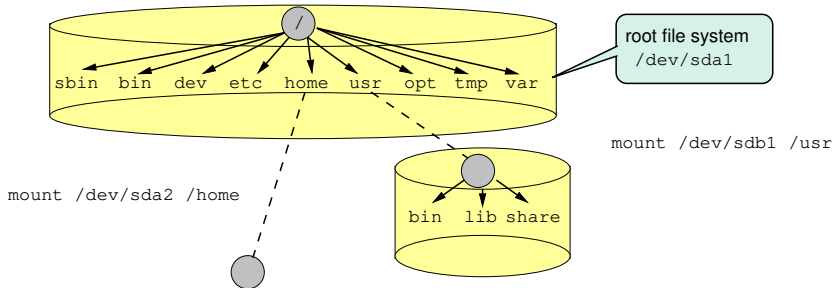
Beispiel:



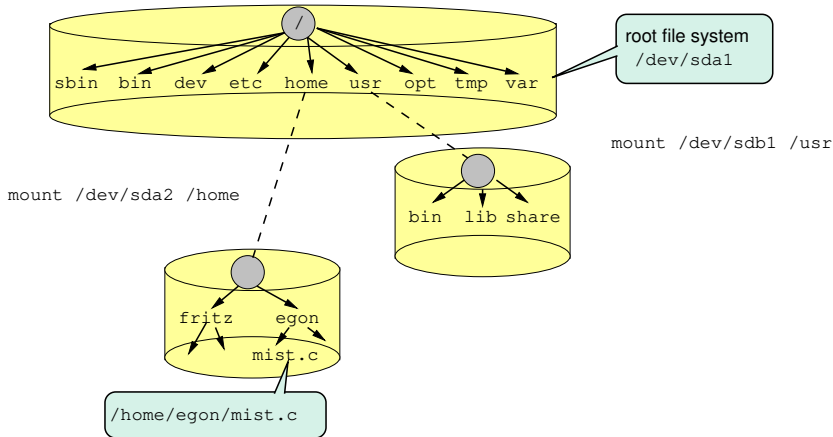
Beispiel:



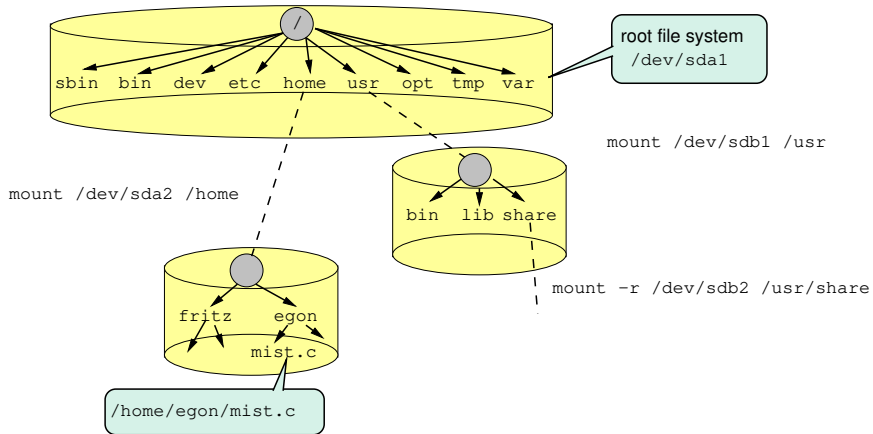
Beispiel:



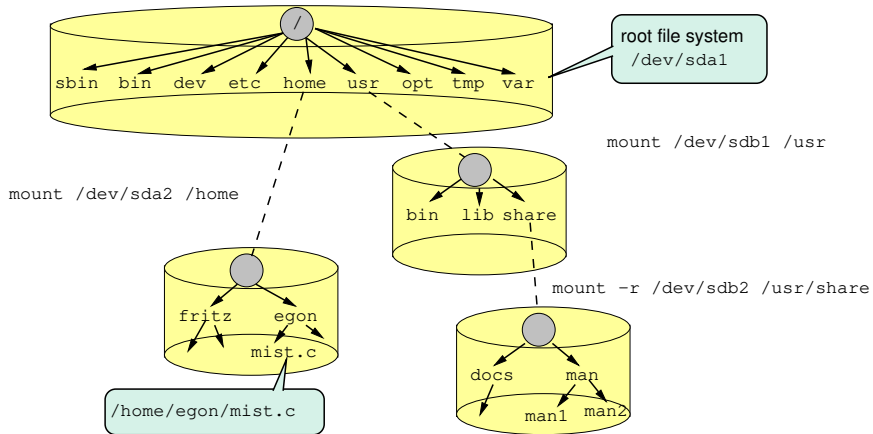
Beispiel:



Beispiel:



Beispiel:



Beispiel: /etc/fstab



zu mountendes Gerät

Mountpunkt

Dateisystem-Typ

Mount-Optionen

Informationen für

Datensicherung (dump)

Rangfolge bei der

Konsistenzprüfung

```
# <file system> <mount point> <type> <options> <dump/pass>
/dev/sda1      /          ext4      errors=remount-ro 0    1
/dev/sda2      /var       ext4      defaults           0    2
/dev/sda3      none       swap      sw                 0    0
/dev/sdb1      /home     ext4      defaults           0    2
/dev/sdb2      /opt      ext4      defaults           0    2
/dev/sr0       /media/cdrom0 udf,iso9660 user,noauto       0    0
^^I
```

Beispiel: /etc/fstab



zu mountendes Gerät

Mountpunkt

Dateisystem-Typ

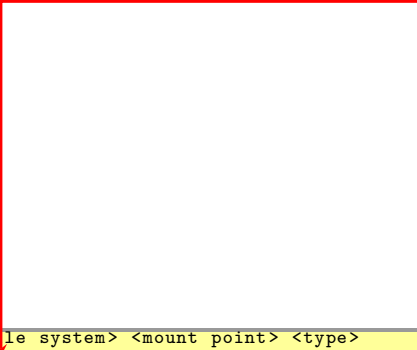
Mount-Optionen

Informationen für

Datensicherung (dump)

Rangfolge bei der

Konsistenzprüfung



# <file system>	<mount point>	<type>	<options>	<dump/pass>
/dev/sda1	/	ext4	errors=remount-ro	0 1
/dev/sda2	/var	ext4	defaults	0 2
/dev/sda3	none	swap	sw	0 0
/dev/sdb1	/home	ext4	defaults	0 2
/dev/sdb2	/opt	ext4	defaults	0 2
/dev/sr0	/media/cdrom0	udf,iso9660	user,noauto	0 0
^^I				

Beispiel: /etc/fstab



zu mountendes Gerät

Mountpunkt

Dateisystem-Typ

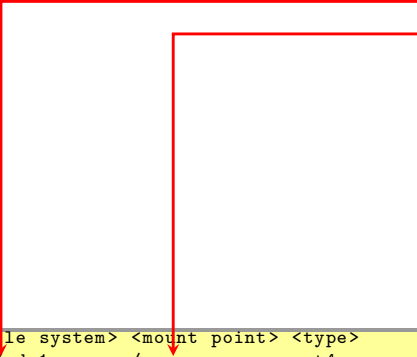
Mount-Optionen

Informationen für

Datensicherung (dump)

Rangfolge bei der

Konsistenzprüfung



#	<file system>	<mount point>	<type>	<options>	<dump/pass>
	/dev/sda1	/	ext4	errors=remount-ro	0 1
	/dev/sda2	/var	ext4	defaults	0 2
	/dev/sda3	none	swap	sw	0 0
	/dev/sdb1	/home	ext4	defaults	0 2
	/dev/sdb2	/opt	ext4	defaults	0 2
	/dev/sr0	/media/cdrom0	udf,iso9660	user,noauto	0 0
	^^I				

Beispiel: /etc/fstab



zu mountendes Gerät

Mountpunkt

Dateisystem-Typ

Mount-Optionen

Informationen für

Datensicherung (dump)

Rangfolge bei der

Konsistenzprüfung

# <file system>	<mount point>	<type>	<options>	<dump/pass>
/dev/sda1	/	ext4	errors=remount-ro	0 1
/dev/sda2	/var	ext4	defaults	0 2
/dev/sda3	none	swap	sw	0 0
/dev/sdb1	/home	ext4	defaults	0 2
/dev/sdb2	/opt	ext4	defaults	0 2
/dev/sr0	/media/cdrom0	udf,iso9660	user,noauto	0 0
^^I				

Beispiel: /etc/fstab



zu mountendes Gerät

Mountpunkt

Dateisystem-Typ

Mount-Optionen

Informationen für

Datensicherung (dump)

Rangfolge bei der

Konsistenzprüfung

# <file system>	<mount point>	<type>	<options>	<dump/pass>
/dev/sda1	/	ext4	errors=remount-ro	0 1
/dev/sda2	/var	ext4	defaults	0 2
/dev/sda3	none	swap	sw	0 0
/dev/sdb1	/home	ext4	defaults	0 2
/dev/sdb2	/opt	ext4	defaults	0 2
/dev/sr0	/media/cdrom0	udf,iso9660	user,noauto	0 0
^^I				

Beispiel: /etc/fstab



zu mountendes Gerät
Mountpunkt
Dateisystem-Typ
Mount-Optionen
Informationen für
Datensicherung (dump)
Rangfolge bei der
Konsistenzprüfung

# <file system>	<mount point>	<type>	<options>	<dump/pass>
/dev/sda1	/	ext4	errors=remount-ro	0 1
/dev/sda2	/var	ext4	defaults	0 2
/dev/sda3	none	swap	sw	0 0
/dev/sdb1	/home	ext4	defaults	0 2
/dev/sdb2	/opt	ext4	defaults	0 2
/dev/sr0	/media/cdrom0	udf,iso9660	user,noauto	0 0
^^I				

Implementierung von Dateisystemen



Im folgenden Sichtweise der Implementierung

- 1 Technische Gegebenheiten
- 2 Plattenplatz-Verwaltung
- 3 Implementierung von Dateien
- 4 Implementierung von Verzeichnissen
- 5 Dateisystemstruktur
- 6 Plattenplatz-Kontingierung
- 7 Wahl der Blockgröße
- 8 Repräsentierung im Hauptspeicher
- 9 Zuverlässigkeit des Dateisystems
- 10 Performance des Dateisystems

Speichermedium: Magnetplatte



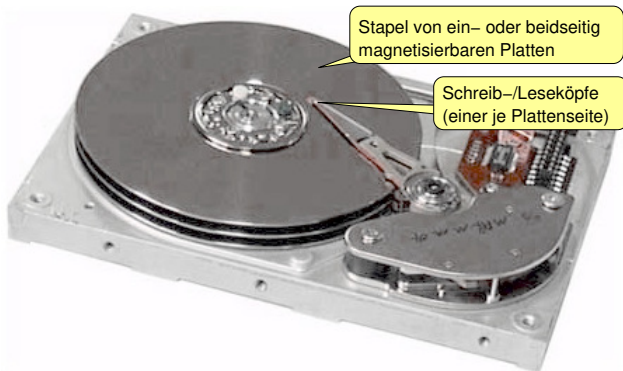
- 5000 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

Speichermedium: Magnetplatte



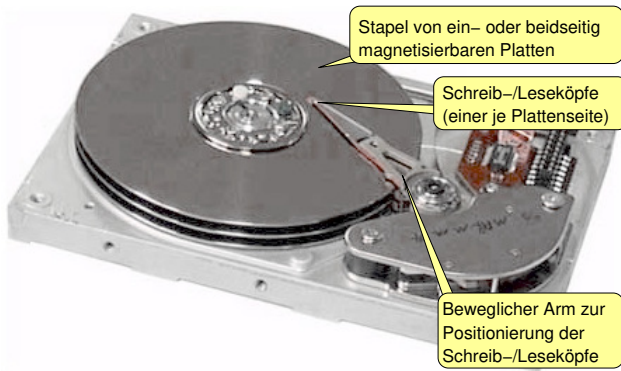
- 5000 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

Speichermedium: Magnetplatte



- 5000 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

Speichermedium: Magnetplatte



- 5000 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

Beispiel Festplatte: Technische Daten³



Hard Disk Model	WDC WD5000AAKX
Disk Family	Caviar Blue
Form Factor	3.5"
Capacity	500 GB (500 × 1 000 000 000 bytes)
Number Of Disks	1
Number Of Heads	2
Rotational Speed	7200 RPM
Rotation Time	8.33 ms
Average Rotational Latency	4.17 ms
Disk Interface	Serial-ATA/600
Buffer-Host Max. Rate	600 MB/seconds
Buffer Size	16384 KB
Average Seek Time	8.9 ms
Track To Track Seek Time	2 ms
Full Stroke Seek Time	21 ms

³Quelle: https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=WDC%20WD5000AAKX

Beispiel Festplatte: Technische Daten³



Hard Disk Model	WDC WD5000AAKX
Disk Family	Caviar Blue
Form Factor	3.5"
Capacity	500 GB (500 × 1 000 000 000 bytes)
Number Of Disks	1
Number Of Heads	2
Rotational Speed	7200 RPM
Rotation Time	8.33 ms
Average Rotational Latency	4.17 ms
Disk Interface	Serial-ATA/600
Buffer-Host Max. Rate	600 MB/seconds
Buffer Size	16384 KB
Average Seek Time	8.9 ms
Track To Track Seek Time	2 ms
Full Stroke Seek Time	21 ms

Maßgeblich für
Roh-Datenrate

³Quelle: https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=WDC%20WD5000AAKX

Beispiel Festplatte: Technische Daten³



Hard Disk Model	WDC WD5000AAKX
Disk Family	Caviar Blue
Form Factor	3.5"
Capacity	500 GB (500 × 1 000 000 000 bytes)
Number Of Disks	1
Number Of Heads	2
Rotational Speed	7200 RPM
Rotation Time	8.33 ms
Average Rotational Latency	4.17 ms
Disk Interface	Serial-ATA/600
Buffer-Host Max. Rate	600 MB/seconds
Buffer Size	16384 KB
Average Seek Time	8.9 ms
Track To Track Seek Time	2 ms
Full Stroke Seek Time	21 ms

Maßgeblich für
Roh-Datenrate

Spitzen-Datenrate
(Schnittstellen-
Eigenschaft)

³Quelle: https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=WDC%20WD5000AAKX

Beispiel Festplatte: Technische Daten³



Hard Disk Model	WDC WD5000AAKX
Disk Family	Caviar Blue
Form Factor	3.5"
Capacity	500 GB (500 × 1 000 000 000 bytes)
Number Of Disks	1
Number Of Heads	2
Rotational Speed	7200 RPM
Rotation Time	8.33 ms
Average Rotational Latency	4.17 ms
Disk Interface	Serial-ATA/600
Buffer-Host Max. Rate	600 MB/seconds
Buffer Size	16384 KB
Average Seek Time	8.9 ms
Track To Track Seek Time	2 ms
Full Stroke Seek Time	21 ms

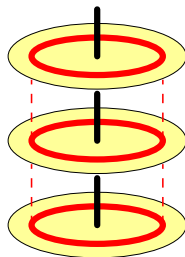
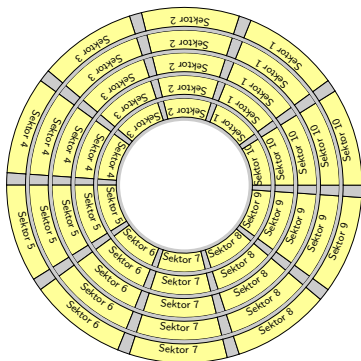
Maßgeblich für
Roh-Datenrate

Spitzen-Datenrate
(Schnittstellen-
Eigenschaft)

Bedingt durch
Kopfbewegung
→ Spitzenrate
wird nicht
dauerhaft erreicht

³Quelle: https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=WDC%20WD5000AAKX

Beispiel Festplatte: Geometrie

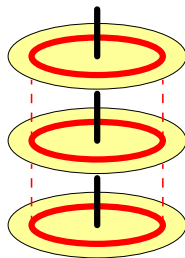
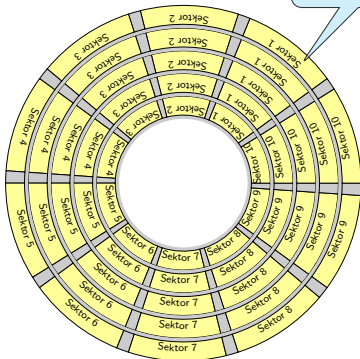


- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Beispiel Festplatte: Geometrie

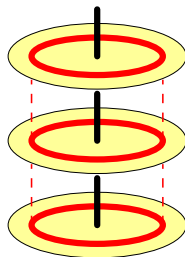
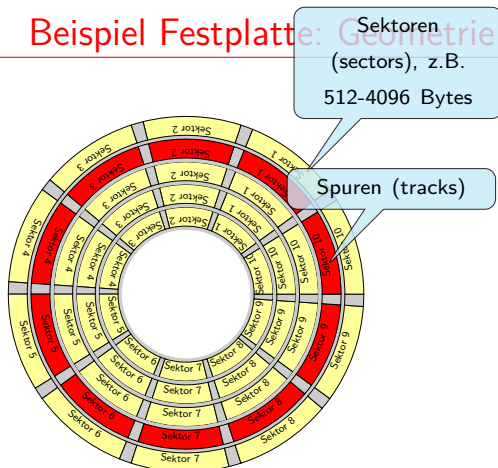


Sektoren
(sectors), z.B.
512-4096 Bytes



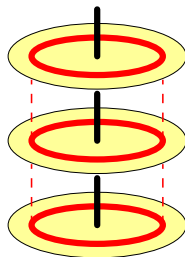
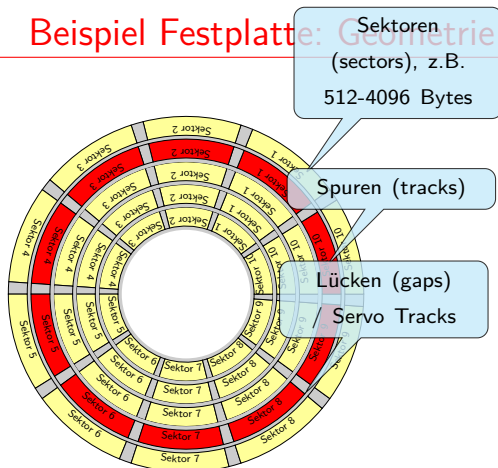
- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Beispiel Festplatte: Geometrie



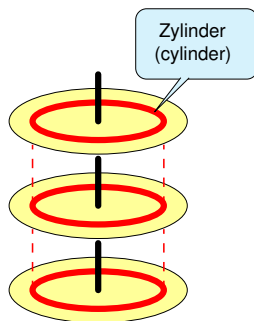
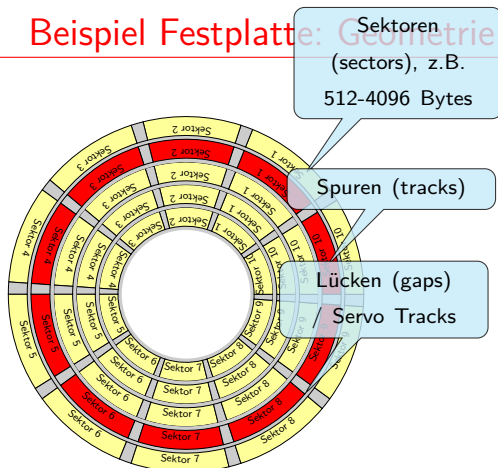
- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Beispiel Festplatte: Geometrie



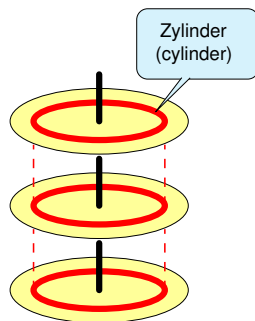
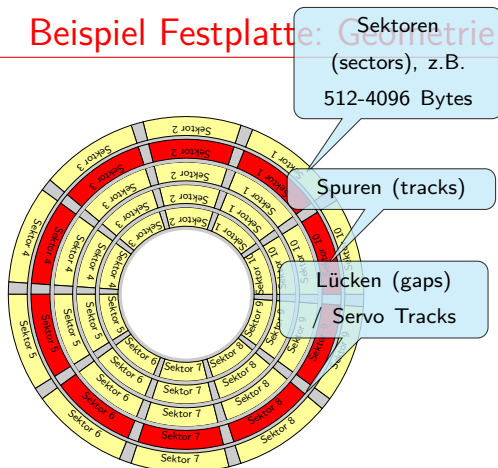
- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Beispiel Festplatte: Geometrie



- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Beispiel Festplatte: Geometrie



- Sektor: Kleinste adressierbare Einheit („Block“)
- Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.
- Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Eigenschaften von Festplatten



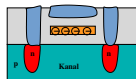
Beobachtungen

- Wahlfreiheit gilt nur bis zur Ebene der Blöcke (kleinste adressierbare Einheit), die Anwenderschnittstelle (s.o.) ermöglicht jedoch Byte-Adressierung
- Die Änderung eines einzelnen Bytes auf der Festplatte erfordert Lesen, Ändern und Zurückschreiben eines ganzen Blockes.
- Heutige Festplatten verwenden logische Blockadressen (LBA), die ursprünglich aus Zylinder-, Kopf- und Sektornummer gebildet wurden:

$$LBA = (C \cdot N_{Heads} + H) \cdot N_{BlocksPerTrack} + S$$

- Hierdurch können auch Geometriedaten der Festplatte, Eigenschaften wie z.B. eine variable Anzahl $N_{BlocksPerTrack}$, etc. abstrahiert werden → bei heutigen Festplatten haben diese Parameter -sofern sie überhaupt angegeben werden- keinen Bezug zur Realität mehr
- Dennoch gilt, dass benachbarte Blöcke mit hoher Wahrscheinlichkeit ohne Kopfbewegungen „in einem Zug“ schneller gelesen / geschrieben werden können.

Speichermedium: Halbleiterspeicher



Transistor

- Speichertechnologie: NAND-Flash, Speicherung von Ladungen auf Transistor-Gates. Typen:

NAND-Typ	Anzahl Zustände	Bits pro Transistor	Lebensdauer (Schreibzyklen)
SLC	1	1	~100.000
MLC	4	2	~5.000
TLC	8	3	~1.000
QLC	16	4	~1.000

- Blockgröße: 4KB (Lesen), >256KB (Löschen)
- Keine mechanischen Bewegungen notwendig → schnelles Lesen
- Änderung eines Bytes erfordert Lesen, Ändern, Löschen und Zurückschreiben eines Blockes
- Begrenzte Lebensdauer: Verschleiß durch Löschen
→ *Wear Leveling* zur Haltbarkeitssteigerung

(Platten-)platzverwaltung



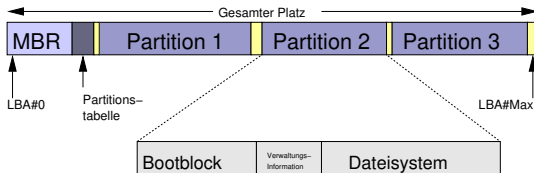
Hauptgesichtspunkte:

- 1 Partitionierung.
- 2 Logische Laufwerke.

Partitionierung



In der Regel wird der gesamte Platz auf mehrere *Partitionen* verteilt



- **MBR** *master boot record*: enthält ausführbaren Code, der beim Systemstart vom BIOS (basic input/output system) geladen und gestartet wird.
- Darin enthalten: **Partitionstabelle**: beschreibt die Aufteilung der Platte in bis zu 4 Partitionen (Anfang, Länge, Typ, ggf. „bootbar“-Flag)
- Der MBR-Code identifiziert eine **Startpartition**, lädt und startet deren ersten Block (**Bootblock**), der seinerseits ggf. das Laden und Starten des Betriebssystems auslöst.
- Der Bootblock muß das Dateisystem des zu startenden Betriebssystems (zumindest eingeschränkt) verstehen, der Code im MBR kann unabhängig davon sein.

Logische Laufwerke

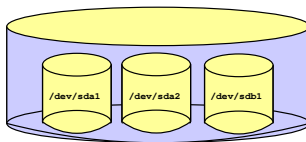


Definition: Logisches Laufwerk

Ein **logisches Laufwerk** (*logical volume*) ist eine dynamisch veränderbare Partition, die sich auch über mehrere physische Datenträger hinweg erstrecken kann.

Beispiel: Linux LVM

- Physische Speichergeräte (physical volumes) werden zu Laufwerksgruppen (volume groups) zusammengefaßt
- Auf einer Laufwerksgruppe können logische Laufwerke eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.



Logische Laufwerke

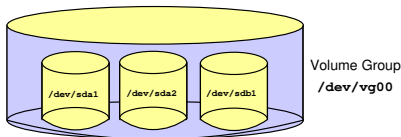


Definition: Logisches Laufwerk

Ein **logisches Laufwerk** (*logical volume*) ist eine dynamisch veränderbare Partition, die sich auch über mehrere physische Datenträger hinweg erstrecken kann.

Beispiel: Linux LVM

- Physische Speichergeräte (physical volumes) werden zu Laufwerksgruppen (volume groups) zusammengefaßt
- Auf einer Laufwerksgruppe können logische Laufwerke eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.



Logische Laufwerke

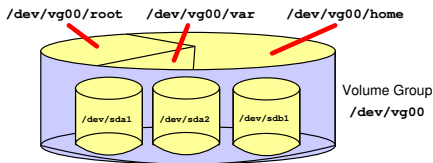


Definition: Logisches Laufwerk

Ein **logisches Laufwerk** (*logical volume*) ist eine dynamisch veränderbare Partition, die sich auch über mehrere physische Datenträger hinweg erstrecken kann.

Beispiel: Linux LVM

- Physische Speichergeräte (physical volumes) werden zu Laufwerksgruppen (volume groups) zusammengefaßt
- Auf einer Laufwerksgruppe können logische Laufwerke eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.



LVM: Nutzen



Im laufenden Betrieb (!) ...

- kann die Kapazität der Laufwerksgruppe durch **Hinzufügen weiterer physische Volumes** vergrößert werden.
- können **Daten** von alten Laufwerken auf neue **verlagert** und die alten Laufwerke außer Betrieb genommen werden.
- kann logischen Laufwerken mehr **Speicherplatz zugeordnet** werden oder Speicherplatz **entzogen** werden.

LVM unterstützt „Filesystem Snapshots“

- Beim Anlegen eines Snapshots wird ein neues logisches Laufwerk angelegt, das den **momentanen Zustand** seines zugehörigen Ursprungs-Laufwerks enthält (eingefrorene Sicht, keine Kopie)
- Ermöglicht **konsistente Backups** über Snapshot-Laufwerk **trotz weiterlaufenden Betriebs** auf dem ursprünglichen Laufwerk

RAID



Weitere Möglichkeit: RAID:

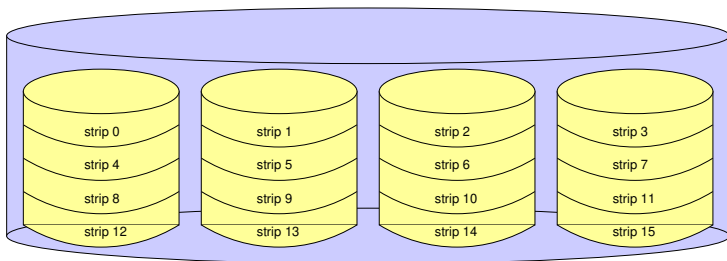
Redundant Array of Independent (Inexpensive) Disks

- **Mehrere Platten zusammengeschaltet**, sehen für den Rechner wie **eine große Platte** aus.
- Realisierungen:
 - ▶ Hardware-RAID (spezieller Festplatten-Controller)
 - ▶ Software-RAID (Betriebssystem verwaltet mehrere angeschlossenen Platten als RAID, Beispiel: Linux mdadm)
- Ziele:
 - ▶ **Erhöhung der Datensicherheit** durch geschickte redundante Speicherung.
 - ▶ Austausch defekter Platten im laufenden Betrieb ohne Unterbrechung (oft auch "hot standby"-Platte)
 - ▶ Verteilung der Daten auf die einzelnen Platten wird durch „RAID level“ (RAID level 0 ... RAID level 6) definiert.

RAID0 - „striping“



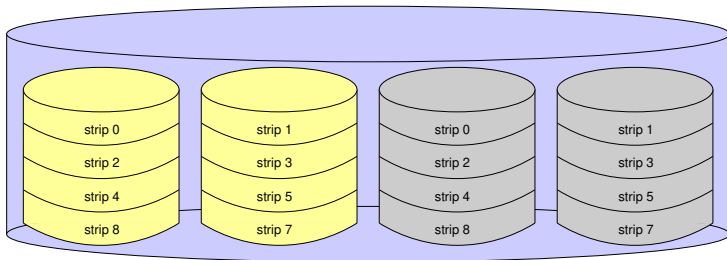
- RAID-Platte wird in „Streifen“ mit k Blöcken eingeteilt
 - Streifen werden reihum auf den angeschlossenen Platten abgelegt.
- **keine Redundanz**, damit keine höhere Fehlertoleranz
- **schneller Zugriff** besonders bei großen Dateien, da Platten parallel arbeiten können
- RAID-Kapazität: Summe der Plattenkapazitäten



RAID1 - „mirroring“



- Zu jeder Platte gibt es eine Spiegelplatte gleichen Inhalts
- **Fehlertoleranz:** Wenn eine Platte ausfällt, kann andere sofort einspringen (übernimmt Controller automatisch)
- **Schreiben:** etwas langsamer; **Lesen:** schneller durch Parallelzugriff auf beide zuständigen Platten
- RAID-Kapazität: Hälfte der Summe der Plattenkapazitäten



RAID5 - parity

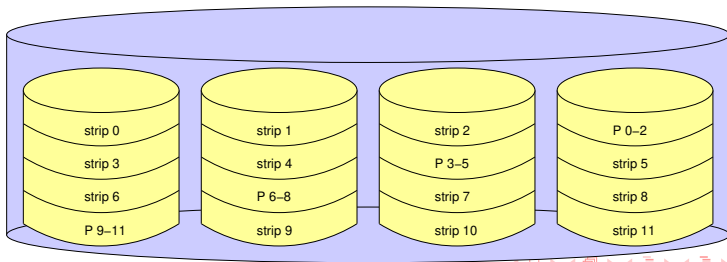


- Paritätsinformation (XOR) auf alle Platten verteilt
Beispiel: P 0-2 enthält XOR-Verknüpfung über die Streifen 0, 1, 2
- XOR Verknüpfung ist „selbstinvers“:

Wenn gilt: $P = A \oplus B \oplus C$
dann ist: $A = B \oplus C \oplus P$
und: $B = A \oplus C \oplus P$
und: $C = A \oplus B \oplus P$

+ Fehlertolerant bei guter
Kapazitätsnutzung
+ Leseoperationen schnell
- Schreiben aufwändiger

→ Solange maximal eine (beliebige) Platte ausfällt, kann ihr Inhalt aus den Übrigen (im lfd. Betrieb) rekonstruiert werden (wieder per XOR)



Aufgaben des Dateisystems



Wollen Benutzende Blöcke, Sektoren, Spuren etc. selbst ansteuern, ihre Daten in 512-Byte-Blöcke aufteilen müssen usw? Wohl kaum. Sie wollen:

- Mithilfe der in 10.1 beschriebenen Funktionen Dateien und Verzeichnisse bearbeiten und verwalten
- Optimale Hardwarenutzung⁴
- **Einheitlichen Zugang** zu vielen (verschiedenen) Speichergerätearten (standardisierte Schnittstelle)

⁴(natürlich ohne eigene Hardware-Kenntnisse ...)

Implementierung von Dateien

Hauptproblem

- Verwaltung der Plattenblöcke und ihrer Zugehörigkeit zu einer Datei.
- Alternativen:
 - 1 Kontinuierliche Allokation.
 - 2 Allokation mittels einer verketteten Liste.
 - 3 Allokation mittels einer verketteten Liste und einem Index.
 - 4 Allokation mittels Index Nodes (*inodes*).

Kontinuierliche Allokation

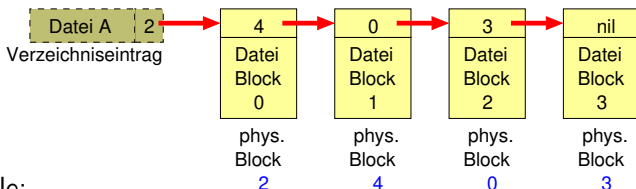


- Jeder Datei wird eine Menge zusammenhängender Blöcke zugeordnet (beim **Anlegen reserviert**).
- Vorteile:
 - ▶ Einfach zu implementieren (nur die Adresse des ersten Blocks ist zu speichern).
 - ▶ Sehr gute Performance beim Lesen und Schreiben der Datei (minimale Kopfbewegungen).
- Nachteile:
 - ▶ Maximalgröße der Datei muss zum Erzeugungszeitpunkt bekannt sein, Wachsen (Append) ist nicht möglich
 - ▶ Externe Fragmentierung durch Löschen / Überschreiben
 - ▶ Verdichtung extrem aufwändig / langwierig
- Einsatzgebiete:
 - ▶ Echtzeit-Anwendungen (zusammenhängende Dateien (*contiguous files*) für kalkulierbare Zugriffszeiten)
 - ▶ Write-Once Dateisysteme (CD/DVD, Logs, Backups, Versionierung).

Allokation mittels verketteter Liste



- Idee: Speicherblöcke einer Datei werden durch Verweise miteinander verkettet.
- Jeder Block hat einen **Verweis auf Nachfolger-Block**
- Verweis z.B. direkt am Beginn jedes Speicherblocks
- Verzeichniseintrag verweist auf ersten Block der Datei

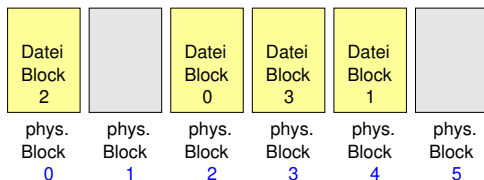
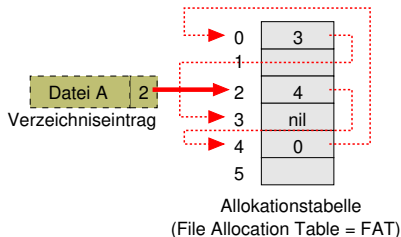


- Vorteile:
 - ▶ Keine Externe Fragmentierung
- Nachteile:
 - ▶ Wahlfreier Zugriff ist seeehr langsam.
 - ▶ Es steht nicht der gesamte Datenblock für Daten zur Verfügung.

Allokation mittels verketteter Liste und Index



- Speicherung der Verkettungsinformation in einer separaten Tabelle (Index oder **F**ile **A**llocation **T**able = FAT).



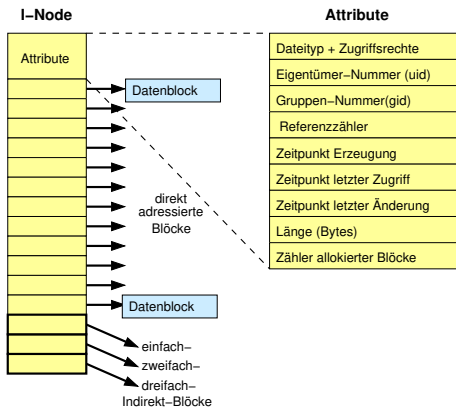
- Vorteile:**
 - ▶ Gesamter Datenblock steht für Daten zur Verfügung.
 - ▶ Akzeptable Performance bei direktem Zugriff, da der Index im Arbeitsspeicher gehalten werden kann.
- Nachteile:**
 - ▶ Die gesamte Tabelle muss im Arbeitsspeicher gehalten werden. Kann bei großer Platte sehr speicherplatzaufwändig sein.
- Beispiel: MS-DOS FAT File System**

Allokation mittels Index Nodes (1)



Definition: I-Node

Ein **I-Node** (UNIX: *inode*) oder Index Node ist ein Dateikontrollblock



- enthält neben Attributen der Datei eine Tabelle mit Adressen von zugeordneten Plattenblöcken.
- Ursprung Beispiel: BSD UNIX Fast File System (ufs)

Allokation mittels Index Nodes (2)



Logische Blocknummern einer Datei ...

- ...werden fortlaufend vergeben,
- beginnend bei den direkt adressierten Blöcken.

Einige wenige (~12) Blockadressen sind im Inode selbst gespeichert

- Nach Öffnen einer Datei und damit verbundenem Einlagern des Inodes in den Hauptspeicher stehen diese Adressen sofort zur Verfügung.
- Schneller Zugriff bei kleinen Dateien.

Für größere bis sehr große Dateien werden nach und nach einfach, zweifach und dreifach indirekte Blöcke zur Speicherung verwendet.

- Zugriffsgeschwindigkeit sinkt bei größeren Dateien.

Beispiel: Linux ext2, ext3

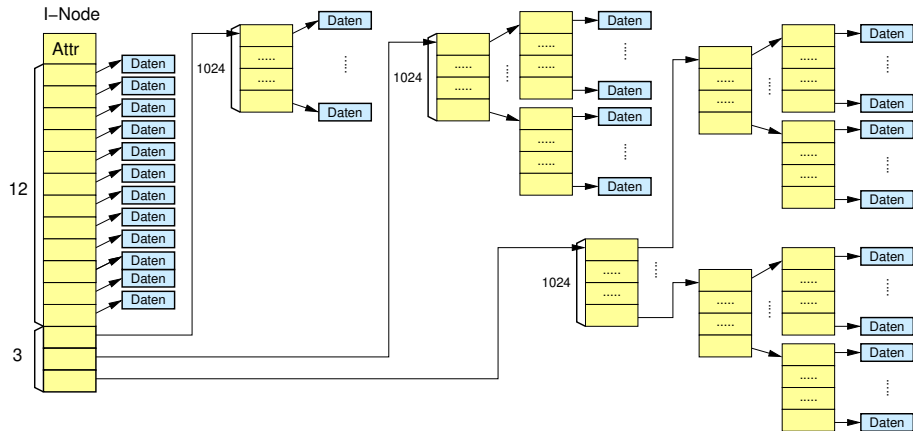
- Blockgröße: 4KB
 - 12 direkt adressierte Blöcke → 48KB
 - Es sind 1-fach, 2-fach und 3-fach indirekte Blöcke vorgesehen
 - Indirekte Blöcke (4KB) speichern bis zu 1024 ($=2^{10}$) weitere Verweise
- ⇒ Maximale Dateigröße:

$$(12 + 2^{10} + 2^{30} + 2^{30}) \cdot 4KB = 48KB + 4MB + 4GB + 4TB \approx 4TB$$

Allokation mittels Index Nodes (3)



Nutzung von Indirekt-Blöcken



Implementierung von Verzeichnissen



Hauptaufgabe des Verzeichnissystems:

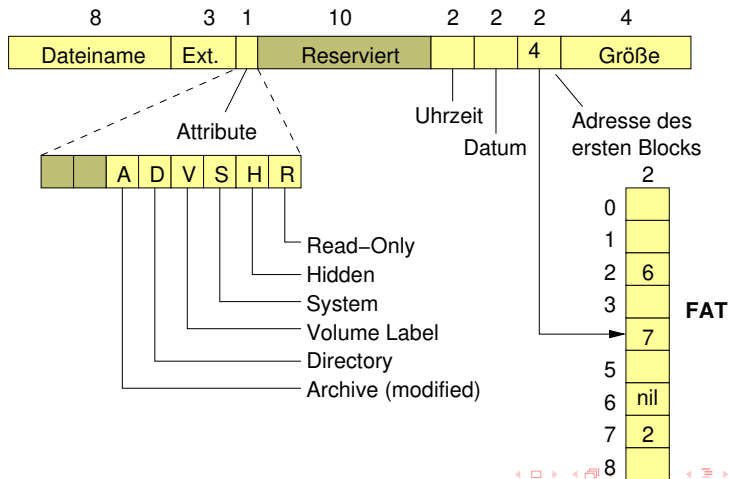
Abbildung der Zeichenketten-Namen von Dateien in Informationen zur Lokalisierung der zugeordneten Plattenblöcke.

- Bei Pfadnamen werden die Teilnamen zwischen Separatoren schrittweise über eine Folge von Verzeichnissen umgewandelt.
- Verzeichniseintrag liefert bei gegebenem Namen (Teilnamen) die Information zum Auffinden der Plattenblöcke:
 - ▶ bei kontinuierlicher Allokation: die Plattenadresse der gesamten Datei oder des Unterverzeichnisses.
 - ▶ bei Allokation mit verketteter Liste mit und ohne Index: die Plattenadresse des ersten Blocks der Datei oder des Unterverzeichnisses.
 - ▶ bei Allokation mit Index Nodes: die Nummer des Inodes der Datei oder des Unterverzeichnisses.

Beispiel: MS-DOS



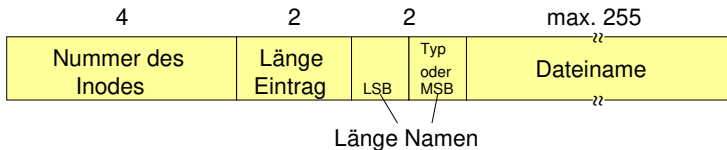
- (s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels verketteter Liste und Index.
- Verzeichniseintrag:



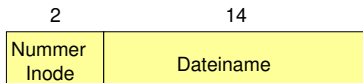
Beispiel: UNIX (1)



- (s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels Index Nodes (inodes).
- Verzeichniseintrag Linux (ext2:)



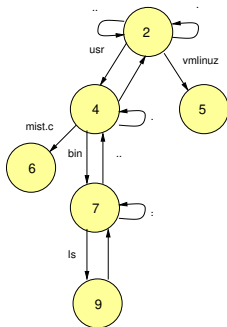
- Verzeichniseintrag klassisches UNIX System V (s5):



Beispiel: UNIX (2)

- Prinzip der Umsetzung eines Pfadnamens

Logische Dateisystemstruktur



Inode-Liste

The diagram illustrates the mapping of file system permissions from a table to directory and file structures. The table on the left lists permissions for various files and directories, with arrows pointing to their corresponding structures on the right.

	root	staff
2	drwxr-xr-x	
	andere Attr	
	Blockliste	
3	frei	
	root	staff
4	drwxr-xr-x	
	andere Attr	
	Blockliste	
	root	source
5	-rwxr--r--	
	andere Attr	
	Blockliste	
	kaiser	prof
6	-rw-----	
	andere Attr	
	Blockliste	
	root	source
7	drwxr-xr-x	
	andere Attr	
	Blockliste	
8	frei	
	root	staff
9	-rwxr--r--	
	andere Attr	
	Blockliste	

Arrows indicate the mapping of permissions to specific file system structures:

- Permissions 2, 3, and 4 map to the root directory structure (Datenblöcke).
- Permissions 5 and 6 map to the /usr directory structure (Directory /usr).
- Permissions 7 and 8 map to the /usr/bin directory structure (Directory /usr/bin).
- Permissions 9 and 10 map to the /usr/bin/ls file structure (Datei /usr/bin/ls).

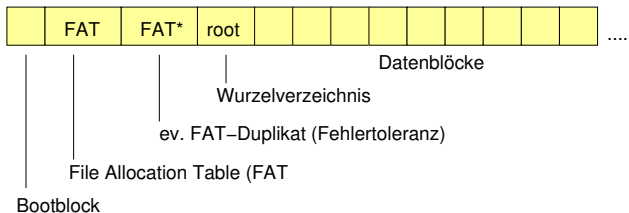
The structures on the right are:

- Datenblöcke:** A table with columns for permissions and values. It shows the mapping of permissions to specific file system structures.
- Directory /usr:** A table with columns for permissions and values. It shows the mapping of permissions to specific file system structures.
- Directory /usr/bin:** A table with columns for permissions and values. It shows the mapping of permissions to specific file system structures.
- Datei /usr/bin/ls:** A table with columns for permissions and values. It shows the mapping of permissions to specific file system structures.

Dateisystemstruktur



- Die Struktur eines Dateisystems wird beim Erzeugen auf die Blockmenge eines logischen Laufwerks (Partition) aufgeprägt.
- Dienstprogramme zum Erzeugen:
 - ▶ MS-DIS: `format`⁵
 - ▶ UNIX: `mkfs`, `(newfs)`
- Beispiel: MS-DOS

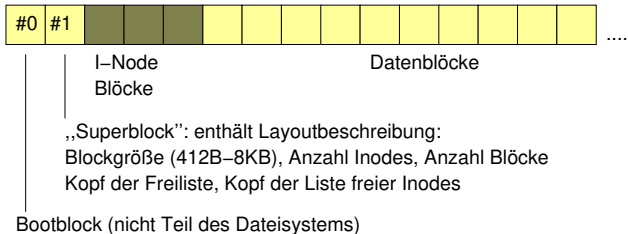


⁵nicht zu verwechseln mit dem Formatieren eines Mediums, in MS-DOS low-level Formatierung genannt

Dateisystemstruktur (2)



- Beispiel: Klassisches UNIX System V (s5)



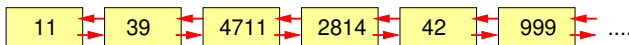
Verwaltung freier Blöcke



Angewendete Methoden:

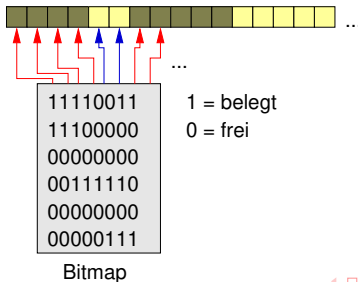
- Verkettete Liste.

- ▶ Vorteil: Größere freie Bereiche einfacher erkennbar
- ▶ Beispiele: MS-DOS FAT, UNIX System V



- Bitmap

- ▶ Vorteil: Größere freie Bereiche einfacher erkennbar.
- ▶ Beispiel: Linux ext2

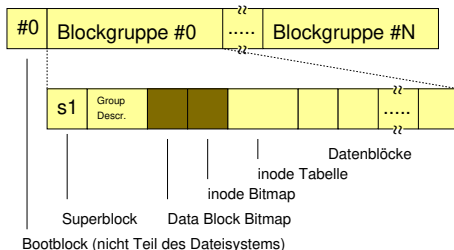


Dateisystemstruktur (3)



Beispiel: Linux ext2: Besonderheiten

- Einführung sogenannter **Blockgruppen**, d.h. Mengen von aufeinander folgenden Blöcken innerhalb eines Dateisystems mit jeweils eigenen Verwaltungsstrukturen.
- I-Node und zugehörige Datenblöcke sollen möglichst dicht beisammen bleiben (Performance).



Redundante Kopien s1, .. des Superblocks in jeder Blockgruppe an verschiedenen Stellen (Kopfpositionen) zur Verbesserung der Verfügbarkeit der Layoutinformation auch bei Plattenfehlern.

Dateisystemstruktur (4)



Weitere Ansätze (hier nicht im Detail besprochen)

- **Log-basierte** Dateisysteme:

- ▶ Für Halbleiter-Speichermedien konzipiert (keine seek-Zeiten)
- ▶ Schreibaufträge puffern, in regelmäßigen Zeitabständen als ganzes Segment schreiben.
- ▶ Position der inodes über in-Memory Map verwalten.
- ▶ Platte wird als „Ringpuffer“ betrieben.
- ▶ „Cleaner“ Thread gibt regelmäßig unbenutzte Blöcke frei.

- **Journaling**-Dateisysteme:

- ▶ Erst: Geplante Operationen (idempotent) in Log schreiben
- ▶ Dann: Operationen ausführen
- ▶ Bei Absturz: Geplante, aber nicht mehr zur Ausführung gekommene Operationen nachholen.

Quotas (Plattenplatz-Kontingentierung)



Ziel:

Vermeidung der Monopolisierung von Plattenplatz durch einzelne Benutzer in Mehrbenutzersystemen.

- Systemadministrator kann jedem Benutzer **Schranken** (Quotas) zuordnen für
 - ▶ maximale Anzahl von eigenen **Dateien** und
 - ▶ Anzahl der von diesen benutzten **Plattenblöcke**
- Betriebssystem stellt sicher, dass diese Schranken nicht überschritten werden.

Beispiel: UNIX Quotas

- Je Benutzer und Dateisystem werden verwaltet:
 - ▶ Weiche Schranke (Soft Limit) für die Anzahl der benutzten Blöcke (kurzfristige Überschreitung möglich).
 - ▶ Harte Schranke (Hard Limit) für die Anzahl der benutzten Blöcke (kann nicht überschritten werden).
 - ▶ Anzahl der aktuell insgesamt zugeordneten Blöcke.
 - ▶ Restanzahl von Warnungen. Diese werden bei Überschreitung des Soft Limits beim Login beschränkt oft wiederholt, danach ist kein Login mehr möglich.
 - ▶ Gleiche Information für die Anzahl der benutzten Dateien (Inodes).

Wahl der Blockgröße



Fast alle Dateisysteme bilden Dateien aus Blöcken fester Länge

(Block umfasst zusammenhängende Folge von Sektoren.)

Problem: Welches ist die optimale Blockgröße?

- Kandidaten aufgrund der Plattenorganisation sind:
 - ▶ Sektor (512 B - 4KB)
 - ▶ Spur (z.B. 256 KB)
- Untersuchungen an Dateisystemen in UNIX-Systemen zeigen:
 - ▶ Die meisten Dateien sind klein (< 10 KB) aber mit wachsender Tendenz.
 - ▶ In Hochschul-Umgebung im Mittel 1 KB [Tanenbaum]
- Eine große Allokationseinheit (z.B. Spur) verschwendet daher zuviel Platz.
- Beispielrechnung: Verschwendeter Platz

(Daten basieren auf realen Dateien, Quelle [Leffler et al].)

Gesamt [MB]	Overhead [%]	Organisation
775.2	0.0	nur Daten, byte-variabel lange Segmente
807.8	4.2	nur Daten, Blockgröße 512 B, int. Fragmentierung
828.7	6.9	Daten und Inodes, UNIX System V, Blockgröße 512 B
866.5	11.8	Daten und Inodes, UNIX System V, Blockgröße 1 KB
948.5	22.4	Daten und Inodes, UNIX System V, Blockgröße 2 KB
1128.3	45.6	Daten und Inodes, UNIX System V, Blockgröße 4 KB

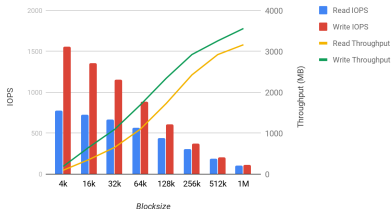
Wahl der Blockgröße (2)



- Eine kleine Allokationseinheit (z.B. Sektor) führt zu schlechter zeitlicher Performance (viele Blöcke = viele Kopfbewegungen).

Mechanische Festplatte

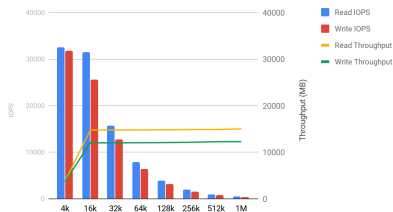
Disk perf vs block size (1TB STD-PD, 32vCPU)



<https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647>

zum Vergleich: SSD N.B.: andere Skalierung

Disk perf vs Block size (1TB PD-SSD, 32vCPU)



<https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647>

Wahl der Blockgröße (3)



Kompromiss:

- Wahl einer mittleren Blockgröße, z.B. 4 KB oder 8 KB.
- Bei Sektorgröße 512 B entspricht ein Block von 4 KB Größe dann 8 aufeinanderfolgenden Sektoren.
- Für Lesen oder Schreiben eines Blockes wird die entsprechende Folge von Sektoren als Einheit gelesen oder geschrieben.
- Ab ca. 2010 für PC-Systeme und Notebooks vermehrt Festplatten mit 4 KB-Sektorgröße (*Advanced Format*)
- Ca. 9% Kapazitätsgewinn (da weniger Gaps)
- Kompatibilitätsprobleme (Lösung durch Emulation) können Performance-Nachteile haben



Wahl der Blockgröße (3)



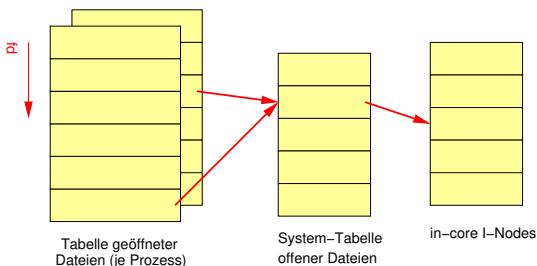
Tricks ...

- Beispiel: BSD UNIX Fast File System (heute ähnlich auch bei ext2)
 - Einführung **zweier** Blockgrößen, genannt **Block** und **Fragment** als Teil eines Blocks (typ. heute 8 KB / 1 KB).
 - Eine Datei besteht aus ganzen Blöcken (falls nötig) sowie ein oder mehreren Fragmenten am Ende der Datei.
- Transfer großer Dateien wird effizient.
- Speicherplatz für kleine Dateien wird gut genutzt.
- Empirisch wurde ein ähnlicher Overhead für ein 4KB/1KB BSD File System beobachtet wie für das 1 KB System V File System.

Repräsentierung im Hauptspeicher



- Bisher wurde die Repräsentierung von Dateien und Verzeichnissen auf dem Hintergrundspeicher betrachtet.
- Geöffnete Dateien besitzen eine Repräsentierung im Arbeitsspeicher

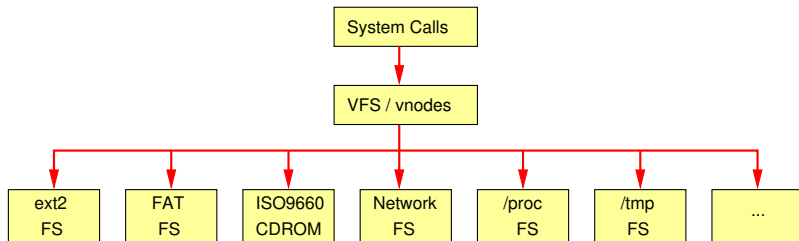


N.B.: Die System-Tabelle enthält insbesondere die Datei-Position jeder geöffneten Datei. Da geöffnete Dateien bei `fork()` vererbt werden, ist die einmalige Verarbeitung des Dateiinhalts durch Eltern- und Kindprozess oder durch mehrere Kindprozesse möglich. Erneutes Öffnen derselben Datei resultiert dagegen in einem neuen Eintrag mit unabhängigem Positionszeiger.

Virtuelles Dateisystem



- Innerhalb des BS-Kerns wurde neue Schnittstelle des Dateisystems eingezeichnet, das sogenannte vnode-Interface (virtual inode) des VFS (Virtual File System).
- Das vnode-Interface umfasst generische Operationen zum Umgang mit Dateien und Verzeichnissen bzw. Dateisystemen als ganzes.
- Die virtuelle Schnittstelle wird für jeden Dateisystemtyp implementiert.
- Das Interface ist auch auf Pseudo-Dateisysteme anwendbar, wie etwa das /proc-Prozessdateisystem (jedem aktiven Adressraum entspricht eine Datei) oder RAM-Disk.



Zuverlässigkeit des Dateisystems



Hauptgesichtspunkte:

- Behandlung fehlerhafter Blöcke.
- Dateisystemkonsistenz.
- Erzeugen und Verwalten von Backups.

Behandlung fehlerhafter Blöcke



- Sowohl Festplatten als auch Flash-Speicher haben i.d.R. von Anfang an **fehlerhafte Blöcke** (z.B. aufgrund ungleichmäßiger Magnetisierung der Oberflächen, Toleranzen in der Chip-Fertigung, etc).
- Bei Festplatten werden fehlerhafte Sektoren beim Formatieren des Mediums (Aufbau der Sektoren) festgestellt. (Im PC-Umfeld wird das Formatieren auch low-level-Formatierung genannt).
- Bei Flash-Speichern kommen aufgrund des Verschleißes im lfd. Betrieb weitere defekte Blöcke hinzu.
- Verzeichnis der fehlerhaften Sektoren wird **Media Defect List** genannt.
- Hardware-Lösung (heute üblich):
 - ▶ Media Defect List wird vom Gerätecontroller selbst geführt.
 - ▶ Jedem defekten Block wird ein Ersatzblock (i.d.R. aus einem dafür reservierten Bereich) zugeordnet, der statt des defekten Blocks benutzt wird.
 - ▶ Bei Festplatten Problem: evtl. unvorhersehbare Kopfbewegungen.
 - ▶ Bei Flash-Speichern kann der reservierte Bereich irgendwann erschöpft sein.
- Software-Lösung (heute eher unüblich):
 - ▶ Es wird eine Datei konstruiert, die nie gelesen oder geschrieben wird und der alle defekten Blöcke zugeordnet werden.

Konsistenz des Dateisystems

Definition: Dateisystem-Konsistenz

Konsistenz eines Dateisystems meint Korrektheit der inneren Struktur des Dateisystems.

- d.h. aller mit der Aufprägung der Dateisystemstruktur auf die Blockmenge verbundenen Informationen (Meta-Information des Dateisystems, UNIX: z.B. Superblock, Freiliste oder Bitmap).
- Beispiel einer Konsistenzregel:
 - ▶ Jeder Block ist entweder Bestandteil genau einer Datei oder eines Verzeichnisses, oder er ist genau einmal als freier Block bekannt.
- Verletzung der Konsistenz:
 - ▶ I.d.R. durch Systemzusammenbruch (z.B. aufgrund eines Stromausfalls) vor Abspeicherung aller modifizierten Blöcke eines Dateisystems.
- Überprüfung der Konsistenz:
 - ▶ Betriebssysteme besitzen Hilfsprogramme zur Überprüfung und evtl. Wiederherstellung der Konsistenz bei eventuell auftretendem Datenverlust.

Beispiel: UNIX



UNIX war traditionell schwach in Bezug auf Sicherstellung der Konsistenz von Dateisystemen:

- Dateisystem (z.B. ufs) wird **nicht in atomaren Schritten** von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Eine solche Veränderung verlangt i.d.R. mehrere Schreibzugriffe.
- Modifizierte Datenblöcke bleiben im Pufferspeicher (Block Buffer Cache) und werden durch einen Dämonprozess spätestens nach 30 sec zurückgeschrieben.
- Modifizierte Blöcke mit Meta-Informationen werden zur Verringerung der Gefahr der Inkonsistenz sofort zurückgeschrieben.
- System Call `sync` existiert zur Einleitung eines sofortigen Zurückschreibens aller veränderten Blöcke (*forced write*).

Beispiel: UNIX (2)



Durch Einführung von Journaling-Dateisystemen⁶ hat sich die Situation deutlich gebessert.

- Meta-Informationen des Dateisystems werden vorab per Write-Ahead-Logging (analog Datenbanken) gespeichert.
- Konsistenz ist gewährleistet, z.B. bei Systemzusammenbruch.
- Dennoch können plötzliche Stromausfälle während laufender physischer Schreibvorgänge Schäden verursachen.
- Gilt für Festplatten wie für Flash-Speicher.
- Deshalb: Medien vor dem Entfernen immer erst „unmounten“.

Zur Verbesserung der Verfügbarkeit von Daten im Falle von Plattenfehlern werden z.B. eingesetzt:

- Spiegelplattenbetrieb (RAID1 - Disk mirroring, vgl. 10.3.2)
- Paritätsinformation speichern (RAID5, vgl. 10.3.2)

⁶ext3, ext4, ReiserFS, Btrfs, XFS, ..., vgl. 10.3.5

Überprüfung und Reparatur



- Dienstprogramme zur Überprüfung / Reparatur der Konsistenz eines **nicht in Benutzung** befindlichen Dateisystems **bei möglichem Datenverlust**: fsck, z.T. auch ncheck
- fsck (*file system check*) führt Konsistenzüberprüfungen durch:
- **Blocküberprüfung**:

- ▶ zwei Tabellen mit jeweils einem Zähler je Block
- ▶ anfangs alle Zähler mit 0 initialisiert

0	5					10					15					
1	1	0	1	1	1	0	0	0	0	1	1	0	1	0	belegte Blöcke	
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	freie Blöcke	

- ▶ erste Tabelle: wie oft tritt jeder **Block in einer Datei** auf?
 - ★ alle inodes lesen.
 - ★ für jeden verwendeten Block Zähler in erster Tabelle inkrementieren.
- ▶ zweite Tabelle: **freie Blöcke**
 - ★ Für Blöcke in der Liste / Bitmap der freien Blöcke Zähler in zweiter Tabelle inkrementieren.
- **Konsistenz**: Für jeden Block muss der Zählerstand aus Tab 1 und Tab 2 zusammen „1“ sein.

Überprüfung und Reparatur (2)



- Fehlender Block: Block₅ 4 ist weder belegt noch frei?₁₀ ₁₅

1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1

belegte Blöcke

freie Blöcke

→ Maßnahme: Block zu freien Blöcken hinzunehmen

- Doppelter Block in Freiliste (Block 8)₀ ₅ ₁₀ ₁₅

1	1	0	1	1	1	0	0	0	0	0	1	1	0	1	0
0	0	1	0	0	0	1	1	2	1	1	0	0	1	0	1

belegte Blöcke

freie Blöcke

→ Maßnahme: Freiliste neu aufbauen

- Doppelter belegter Block (Block 11)₀ ₅ ₁₀ ₁₅

1	1	0	1	1	1	0	0	0	0	0	2	1	0	1	0
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1

belegte Blöcke

freie Blöcke

→ Maßnahme: Block kopieren, Kopie-Block in eine der beiden betroffenen Dateien statt Block 11 einbauen

Überprüfung und Reparatur (3)



- Darüber hinaus **Überprüfung der Verzeichniseinträge**:
 - ▶ Vergleiche Anzahl aller Verzeichniseinträge mit Verweis auf einen inode mit dem darin gespeicherten Referenzzähler.
 - Maßnahme: ggf. inode-Referenzzähler der durch Zählung festgestellten Zahl von Referenzen anpassen.
(N.B.: Es kann „beliebig viele“ (> 0) Referenzen auf einen inode geben (aufgrund harter Links!))
- **Problem**: Für große Platten kann ein fsck-Lauf sehr lange (Stunden!) dauern. In dieser Zeit ist das System möglicherweise nicht verfügbar (Kosten!)

Performance des Dateisystems



Maßnahmen zur Performance-Steigerung:

- Blockgruppen:
 - ▶ Ziel: Vermeiden von weiten Kopfbewegungen.
 - ▶ Beispiel: Linux ext2 File System (vgl. 10.3.5).
 - ▶ Kein Effekt (auch kein negativer) bei Flash-Speicher
- Block Buffer Cache.
 - ▶ Ziel: Reduzierung der Anzahl der Plattenzugriffe.
 - ▶ Ein Teil des Arbeitsspeichers, **(Block) Buffer Cache** genannt, wird als Cache für Dateisystemblöcke organisiert.
 - ▶ Typische Hitrate: 85%
 - ▶ Modifizierter LRU-Algorithmus zur Auswahl zu verdrängender Blöcke unter Berücksichtigung der Forderungen zur Verringerung der Gefahr von Inkonsistenz
 - ▶ Blöcke verbleiben im Cache, auch wenn die entsprechenden Dateien geschlossen sind.

Performance des Dateisystems (2)



• Dateinamens-Cache:

- ▶ Ziel: Verringerung der Anzahl der Schritte bei der Abbildung von Datei-Pfadnamen auf Blockadressen.
- ▶ Beispiel: BSD UNIX namei-Cache:
 - ★ `namei()`-Routine
zur Umsetzung von Pfadnamen auf inode-Nummern.
benötigte vor Einführung von Caching ca. 25% der Zeit eines Prozesses im BS-Kern, auf <10% gesenkt.
 - ★ Cache der n letzten übersetzten Teilnamen:
Typische Hitrate: 70-80%.
Höchste Bedeutung für Gesamtperformance.
 - ★ Cache des letzten benutzten Directory-Offsets:
Wenn ein Name im selben Verzeichnis gesucht wird, beginnt die Suche am gespeicherten offset und nicht am Anfang des Verzeichnisses (sequentielles Lesen eines Verzeichnisses ist häufig, z.B. durch das `ls`-Kommando).
Typische Hitrate: 5-15 %.
 - ★ Gesamthitrate von ca. 85% wird erreicht.

Zusammenfassung



Was haben wir in Kap. 10 gemacht?

- Konzepte von Dateisystemen.
- Verwalten von Mengen von Dateien und Verzeichnissen.
 - ▶ Zunächst eine äußere Benutzersicht
 - ★ Benennung von Dateien
 - ★ Dateistrukturen und -Typen
 - ★ Zugriffsarten
 - ★ Dateiattribute
 - ★ Hierarchische Strukturierung.
 - ▶ Dateisysteme aus der Sicht der Implementierung
 - ★ Technische Gegebenheiten
 - ★ Plattenplatz-Verwaltung
 - ★ Implementierung von Dateien
 - ★ Implementierung von Verzeichnissen
 - ★ Dateisystemstruktur
 - ★ Plattenplatz-Kontingentierung
 - ★ Wahl der Blockgröße
 - ★ Repräsentierung im Hauptspeicher
 - ★ Zuverlässigkeit des Dateisystems
 - ★ Performance des Dateisystems