



# **Kap. 10: Dateisysteme**

## 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).

### Def

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.

- Datenhaltungssysteme**

	<b>Dateisysteme</b>	<b>Datenbank-systeme</b>	<b>Objekt-management systeme</b>
<b>Inhalt</b>	universell	Massendaten weniger struktureller Typen	Beziehungen stehen im Vordergrund
<b>Gespeicherte Information</b>	passiv	passiv	aktiv
<b>Semantik aufprägender Code</b>	extern	extern	intern (Typen)
<b>Zugriff</b>	über Namen, einfache Navigation	komplexe assoziative Suchfunktionen	komplexe Such- und Navigations- funktionen

10.1 Dateien

10.2 Verzeichnisse

10.3 Implementierung von Dateisystemen

10.4 Sicherheit

10.5 Schutz

10.6 Zusammenfassung

UNIX wird beispielhaft in den jeweiligen Abschnitten behandelt.

Zunächst Einführung von Dateien aus Benutzersicht.

## Gliederung

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

# 10.1.1 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.

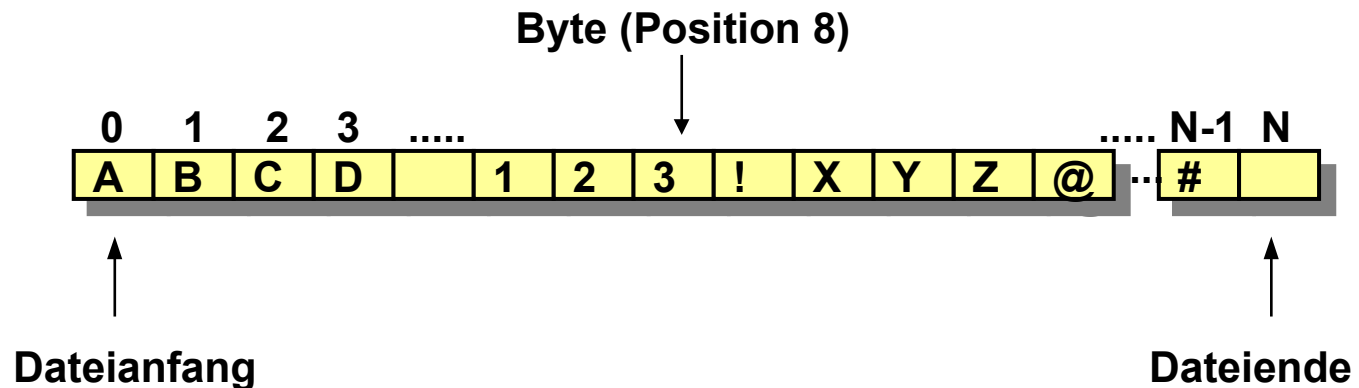
## Def

- Der Dateinamenraum definiert die Menge der zulässigen Dateinamen.
- Regeln für die Konstruktion zulässiger Dateinamen stark systemabhängig:
  - Unterscheidung zwischen Groß- und Kleinbuchstaben (ja: UNIX, nein: MS-DOS).
  - Länge der zulässigen Dateinamen (BSD UNIX: 255 Zeichen, MS-DOS: 8 Zeichen).
  - Verwendung von Sonderzeichen.
  - 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 verarbeitenden Programmen erzwungen

# 10.1.2 Dateistrukturen



Die Datei als unstrukturierte Folge von Bytes:



- Das Betriebssystem **weiß nichts** über den Inhalt oder dessen Struktur, sieht eine Datei ausschließlich als Container an.
- Vorteil: Maximale Flexibilität.
- Beispiele: UNIX, MS-DOS.

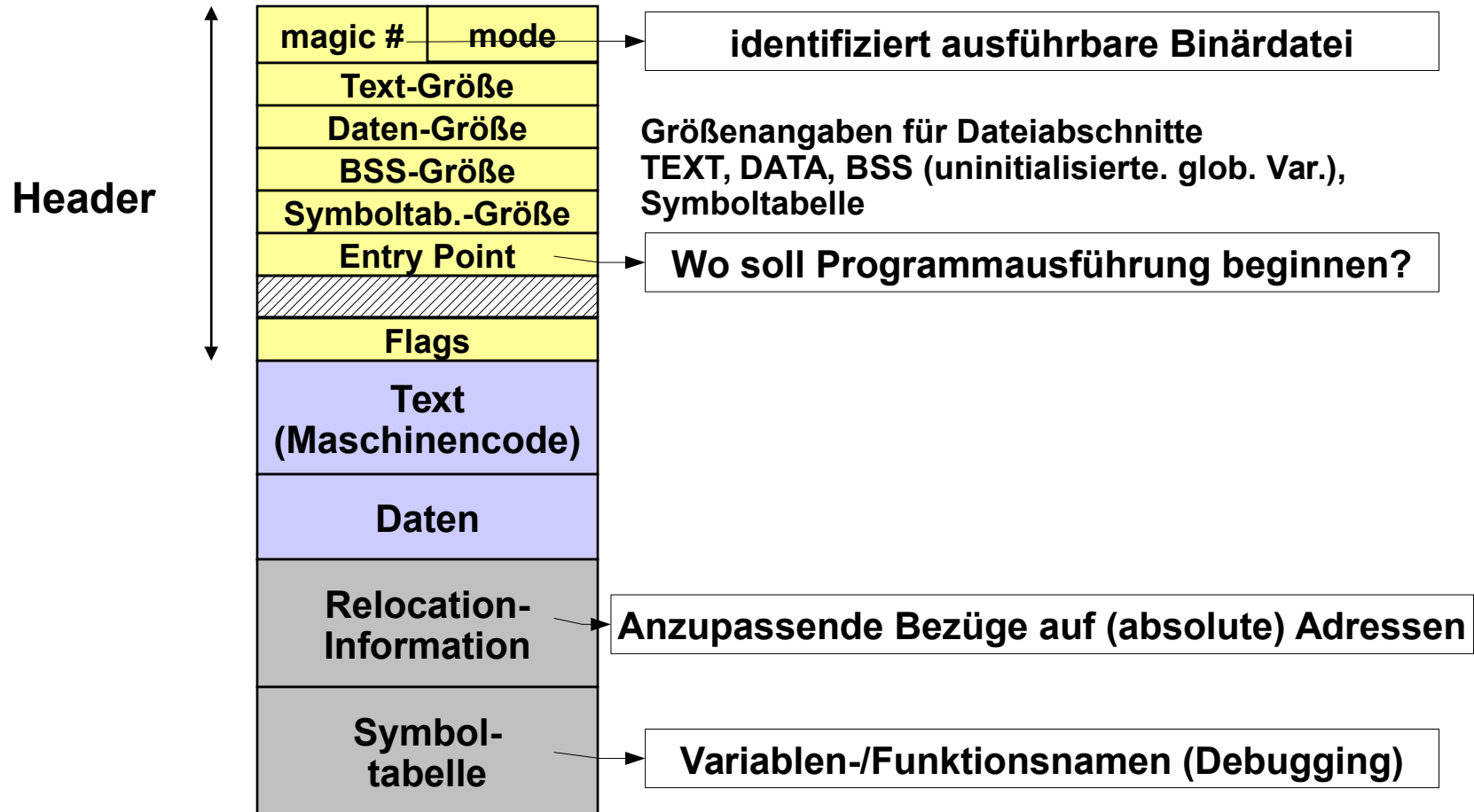


# 10.1.3 Dateitypen



- Gewöhnliche Dateien (Regular Files):
  - ASCII- („Text-“) Dateien:
    - ➔ Folge von Textzeilen variabler Länge, i.d.R. durch *betriebssystemabhängiges* Steuerzeichen getrennt (UNIX: „\n“, MacOS: „\r“, Windows: „\r\n“)
    - ➔ Vorteil: Mit Editor einfach manipulierbar.
  - Binär-Dateien:
    - ➔ Sonstige.
    - ➔ Verschiedene interne Formate, abhängig von Verwendung, häufig durch sog. Magic Numbers gekennzeichnet, z.B. als ausführbare Dateien.
- 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.

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



# open()



```
#include <fcntl.h>
int open(char *pathname, int mode, int accessmode);
int open(char *pathname, int mode);
```

- **pathname** - Name oder Pfad der zu öffnenden Datei
- **mode** - wie soll die Datei geöffnet werden? (→ **fcntl.h**)
  - **O\_RDONLY** nur lesen, **O\_WRONLY** nur schreiben
  - **O\_RDWR** lesen und schreiben
- Per Bit-ODER ("|") können verschiedene Flags hinzugefügt werden:
  - **O\_APPEND** Schreibzugriffe: am Dateiende anhängen
  - **O\_CREAT** Datei anlegen, falls noch nicht vorhanden  
(in diesem Fall nur Variante *mit mode* erlaubt)
  - **O\_EXCL** (mit **O\_CREAT**) Fehler, falls Datei schon da
  - **O\_TRUNC** löscht Dateiinhalt, falls schon vorhanden
  - **O\_NDELAY** erlaubt nicht-blockierende E/A-Operationen
  - **O\_SYNC** sofortiges Zurückschreiben bei Schreiboperationen
- **accessmode**: Bitmuster für Zugriffsrechte (bei **O\_CREAT**)
- Ergebnis: „Dateideskriptor“-Wert oder -1 bei Fehler



# Beispiel: open()

```
int fd1, fd2, fd3;  
  
fd1 = open("test-1", O_RDONLY);  
fd2 = open("test-2", O_WRONLY | O_APPEND);  
fd3 = open("test-3", O_RDWR | O_CREAT | O_TRUNC, 0640);
```

- Datei **test-1** kann über Dateideskriptor **fd1** gelesen werden
- Datei **test-2** kann über Dateideskriptor **fd2** beschrieben werden. Dabei wird am Ende angehängt.
- Datei **test-3** kann über Dateideskriptor **fd3** geschrieben und gelesen werden.
  - Falls die Datei noch nicht existiert, wird sie angelegt (creat)
  - Falls sie schon existiert, wird der Inhalt gelöscht (trunc)
  - Rechte: **rw- r-- ---** (drei 3-Bit-Gruppen → oktal 0640)  
110 100 000  
6 4 0

```
#include <unistd.h>
int read(int fd, char *daten, unsigned anzahl);
int write(int fd, char *daten, unsigned anzahl);
int close(int fd);
```

- **read()**
  - liest bis zu **anzahl** Bytes vom Dateideskriptor **fd** in den Hauptspeicher ab Adresse **daten** ein
  - Rückgabewert: Anzahl tatsächlich gelesener Bytes (oder -1 bei Fehler)
- **write()**
  - schreibt (bis zu) **anzahl** Bytes ab Adresse **daten** auf **fd**
  - Rückgabewert: Anzahl tatsächlich geschriebener Bytes (oder -1 bei Fehler)
- **close()**
  - schließt Datei mit Deskriptor **fd** (-1 bei Fehler)



# Exkurs: Fehlerbehandlung, perror()

- Viele Systemfunktionen liefern einen Wahrheitswert zurück (0 für „ok“, -1 für „Fehler“)
- Vorsicht, in „C“ ist 0 „falsch“ und nicht-Null „wahr“!
- Der genaue Fehlercode steht in der globalen `int`-Variablen `errno` (dazu: `#include <errno.h>`)
- Die Funktion `perror(char *)` gibt dann eine passende Fehlermeldung mit einem frei wählbaren **Begleittext** aus.
- Beispiel:

```
#include <errno.h>
int main(int argc, char *argv[]) {
    int ergebnis;
    ergebnis = rename(argv[1], argv[2]);
    if (ergebnis != 0) {
        perror("Fehler beim Umbenennen");
        return -1;
    }
    return 0;
}
```



```
$ a.out gibts_nicht irgendwas
Fehler beim Umbenennen: No such file or directory
```

# Beispiel: Kopierprogramm (1)



```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    enum { BUFSIZE=1000 };
    char buffer[BUFSIZE];
    int lese_fd, schreib_fd, gelesen, geschrieben;
    if (argc != 3) { printf("Falscher Aufruf\n"); exit(EXIT_FAILURE); }
    lese_fd = open(argv[1], O_RDONLY);
    if (lese_fd < 0) {
        perror("Bei Oeffnen der Eingabedatei");
        exit(EXIT_FAILURE);
    }
    schreib_fd = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0644);
    if (schreib_fd < 0) {
        perror("Bei Oeffnen der Ausgabedatei");
        exit(EXIT_FAILURE);
    }
}
```

**Programm sofort mit  
Rückgabewert 1 beenden**






# Beispiel: Kopierprogramm (2)

```
/* Fortsetzung ... */
while (1) {
    gelesen = read(lese_fd, buffer, BUFSIZE);
    if (gelesen == 0) {
        break;
    } else if (gelesen < 0) {
        perror("Lesefehler");
        break;
    }
    geschrieben = write(schreib_fd, buffer, gelesen);
    if (geschrieben <= 0) {
        perror("Schreibfehler");
        exit(EXIT_FAILURE);
    }
}
close(lese_fd);
close(schreib_fd);
return gelesen == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

N.B: gelesen, nicht BUFSIZE!



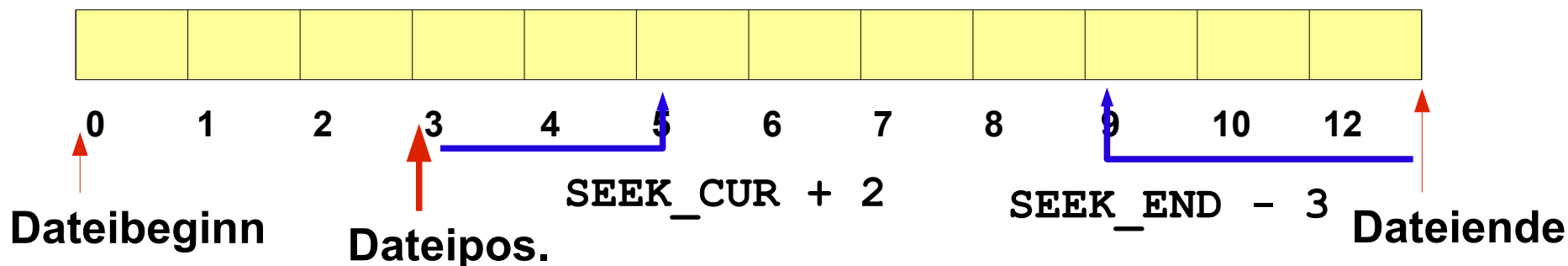




# Direktpositionierung: lseek()

```
#include <unistd.h>
#include <sys/types.h>
int lseek(int fd, off_t offset, int basis);
```

- `lseek()` positioniert die aktuelle Dateiposition von `fd` auf den Wert `offset` gemäß der Einstellung von `basis`
- Werte für "`basis`" (`offset` darf auch negativ sein):
  - `SEEK_SET`: neue Position wird auf `offset` gesetzt
  - `SEEK_CUR`: neue Position ist aktuelle Position + `offset`
  - `SEEK_END`: neue Position ist Dateiende + `offset`
- Rückgabewert: neue Position (ab Anfang gezählt) oder -1 bei Fehler





# Beispiel: Direktzugriff

- struct-Typ beschreibt Messwerte-Datensatz

```
typedef struct mw {  
    char ableser[20];  
    float temperatur;  
} Messwert;
```

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



Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
<b>Meier</b>	<b>Hinz</b>	<b>Kunz</b>	<b>Knüsel</b>	<b>Berger</b>	<b>Huber</b>	<b>Mayr</b>
<b>25.1</b>	<b>110.5</b>	<b>11.0</b>	<b>16.2</b>	<b>17.3</b>	<b>18.0</b>	<b>110.2</b>



# speichern() mit Direktzugriff

```
int speichern(int fd, Messwert *pm, int tag) {  
    if (lseek(fd, tag*sizeof(Messwert), SEEK_SET) < 0) {  
        perror("speichern (lseek)");  
        return -1;  
    }  
  
    if (write(fd, pm, sizeof(Messwert)) < 0) {  
        perror("speichern (write)");  
        return -1;  
    }  
    return 0;  
}
```

```
Messwert m;  
enum { SONNTAG, MONTAG, DIENSTAG, MITTWOCH, ...};  
...  
fd = open("messwerte.dat", O_RDWR | O_CREAT, 0644);  
err1 = speichern(fd, &m, SONNTAG);  
err2 = lesen(fd, &m, MONTAG); /* wie speichern() */
```



# stat(): Dateiattribute abfragen

```
int stat(char *file_name, struct stat *buf);  
int fstat(int filedeskriptor, struct stat *buf);
```

- liefert Informationen zu der durch `file_name` angegebenen Datei in der angegebenen stat-Struktur (-1 bei Fehler).
- Identisch `fstat` bei bereits geöffneter Datei.

```
struct stat  
{  
    dev_t      st_dev;      /* Device */  
    ino_t      st_ino;      /* INode */  
    mode_t     st_mode;     /* Zugriffsrechte */  
    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-Gerät)*/  
    off_t      st_size;     /* Größe in Bytes*/  
    unsigned long st_blksize; /* Blockgröße */  
    unsigned long st_blocks; /* Allozierte Blocks */  
    time_t     st_atime;    /* Letzter Zugriff */  
    time_t     st_mtime;    /* Letzte (Inh.)Änderung*/  
    time_t     st_ctime;    /* Letzte Statusänderung */  
};
```



# Dateiattribute setzen (Auswahl)

- **Zugriffsrechte ändern**

- `int chmod(char *Pfad, mode_t Rechte);`
- Ergebnis: 0 für ok, -1 für Fehler

```
if ( chmod("meineDatei.txt", 0600) == 0 ) {  
    /* ok! */  
}
```

- **Dateibesitzer / -gruppe ändern**

- `int chown(char *path, uid_t owner, gid_t group);`
- Ergebnis: 0 für ok, -1 für Fehler

```
if ( chown("meineDatei.txt", 7, 27) == 0 ) {  
    /* ok! */  
}
```

- **Hinweis:** ID-Nummern für Eigentümer (uid) und Gruppe (gid) stehen z.B. in der Datei `/etc/passwd`; Angabe von -1: keine Änderung

- **Zugriffs- und Modifikationszeitpunkte setzen (vgl. `<utime.h>`)**

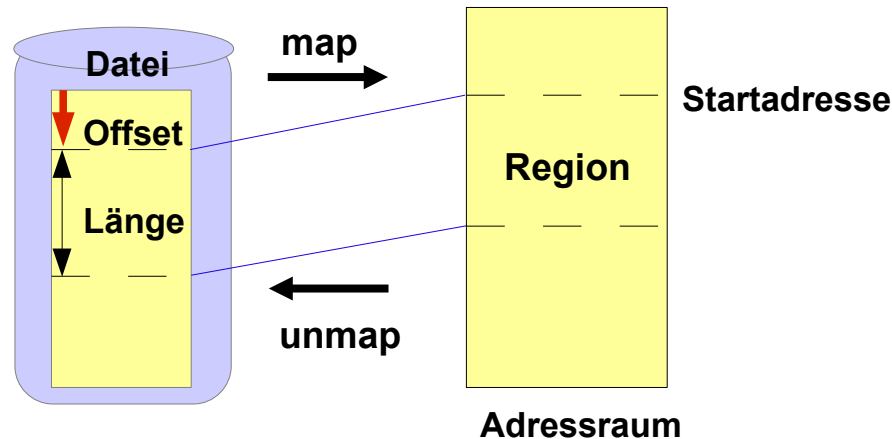
- `int utime(char *path, utimbuf *time);`

# 10.1.7 Memory-Mapped Files



## Def

- 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.
- Beispiele: UNIX, Windows NT.

# mmap()



```
#include <unistd.h>
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- **length** Bytes von Dateideskriptor **fd** ab Position **offset**
- sollte ab Adresse **start** eingeblendet werden (**start == 0**: System wählt Adresse selbst)
- **prot** gibt Zugriffsart an (lesen, schreiben, ausführ.)
- **flags**: z.B. **MAP\_SHARED**: Änderungen für andere sichtbar
- Ergebnis: Anfangsadresse oder -1 bei Fehler
- **int munmap(void \*start, size\_t length);**
  - Aufheben des Mappings
- Die ganze Wahrheit: **man mmap**

# 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 = mmap(0, laenge, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);

    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;
}
```





## Gliederung

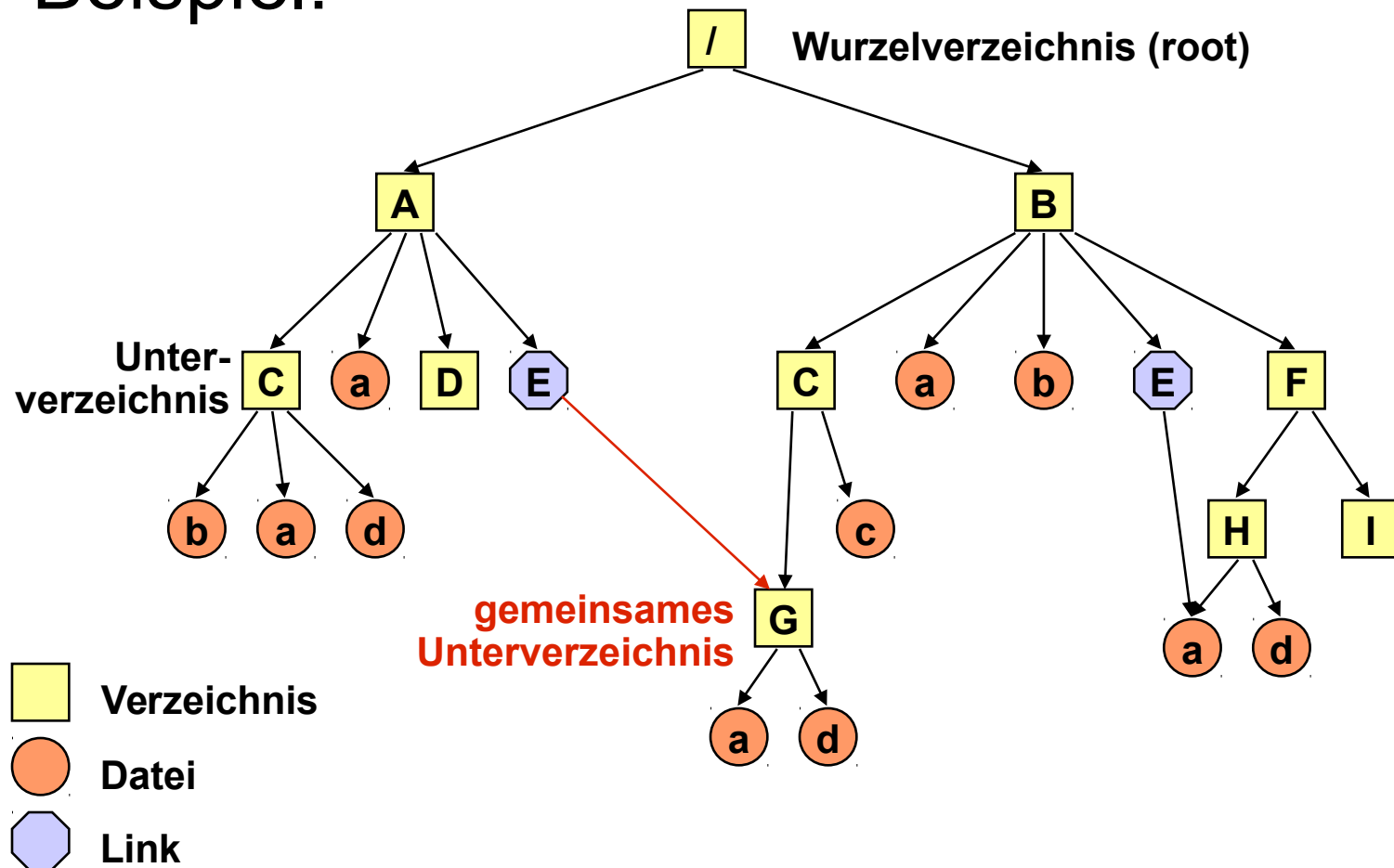
1. Hierarchische Verzeichnissysteme
2. Pfadnamen
3. Verzeichnisoperationen
4. Montierbare Verzeichnisbäume



**Def**

- 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).
- Alternativen:
  - Ein Verzeichnis für alle Dateien sämtlicher Benutzer.  
Beispiel: CP/M.
  - Ein Verzeichnis für jeden Benutzer.
  - Hierarchischer Verzeichnisbaum für jeden Benutzer zur Strukturierung seiner Dateimenge (heute Standard).  
Beispiel: MS-DOS.
  - Darüber hinaus Einführung zusätzlicher symbolischer Links zur flexiblen gemeinsamen Benutzung von Dateimengen.  
Dateisystemstruktur wird damit zu einem azyklischen Graphen.  
Beispiel: UNIX.

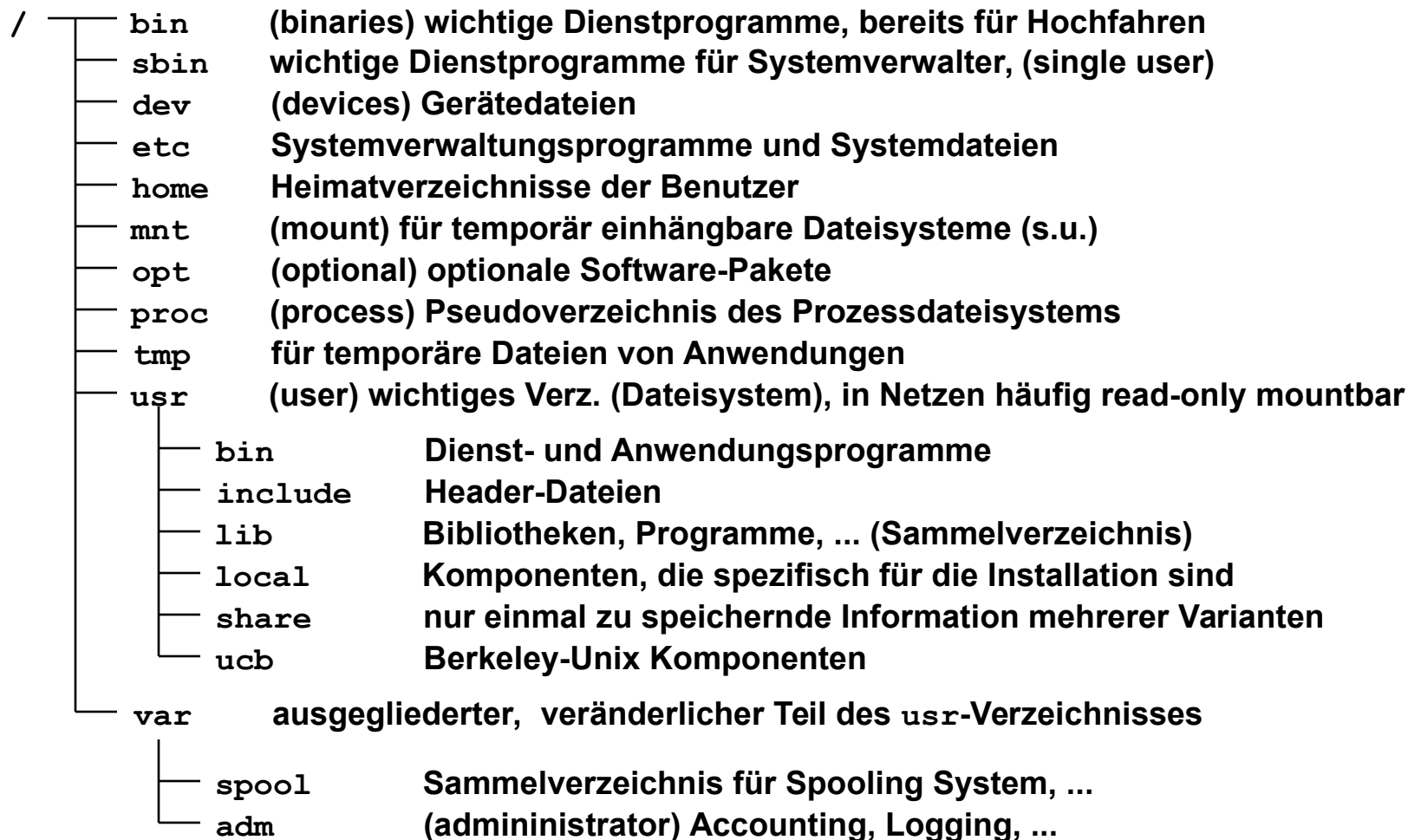
Beispiel:



**Def**

- Für hierarchische Verzeichnissysteme dienen Pfadnamen (*Path Names*) der Benennung von Dateien und Verzeichnissen.
- Alternativen:
  - Absolute Pfadnamen:
    - ➔ Beispiel UNIX: "/" Name des Wurzelverzeichnisses
    - ➔ Benennung vom Wurzelverzeichnis aus.
    - ➔ Beispiel 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 Vater-Verzeichnis.

## Beispiel: UNIX System V.R4



## 10.2.3 Verzeichnisoperationen

---



- Typische Verzeichnisoperationen sind
  - Create / Delete / Change Directory,
  - Opendir / Close / Read Directory,
  - Link, Unlink.



# UNIX: mkdir(), rmdir(), chdir()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int mkdir(char *pathname, mode_t mode);
int rmdir(char *pathname);
int chdir(char *pathname);
```

- **mkdir()** legt Verzeichnis **pathname** mit Zugriffsrechten **mode** und Einträgen für „.“ und „..“ an (auch mit **mknod** möglich)
- **rmdir()** löscht das (bis auf die Einträge „.“ und „..“ leere!) Verzeichnis **pathname**
- **chdir()** setzt das aktuelle Verzeichnis für den ausführenden Prozess auf **pathname**
- 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(), readdir(), closedir()



```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(char *pathname);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
```

- **opendir()** öffnet die Verzeichnisdatei **pathname** und gibt einen Zeiger auf **DIR** zurück (**NULL** bei Fehler)
- **closedir()** schließt eine Verzeichnisdatei; Ergebnis ist 0 (ok) oder -1 (Fehler)
- **readdir()** liefert jeweils nächsten Verzeichniseintrag. Bei Ende oder Fehler wird der **NULL**-Zeiger geliefert
- Die **struct dirent** enthält ein Feld **char d\_name[]** mit dem Namen des betreffenden Verzeichniseintrags



# Beispiel: mini-"ls" (Ausgabe)



**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)
...
```

# Beispiel: mini-„ls“ (1)



```
#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);
    }
}
```

# Beispiel: mini-„ls“ (2)



```
strcpy(pfadpuffer, argv[1]);
strcat(pfadpuffer, "/");
pfadp = pfadpuffer + strlen(pfadpuffer);
while (1) {
    eintrag = readdir(dir);
    if (eintrag == NULL) break;
    if (strcmp(eintrag->d_name, ".")==0 ||
        strcmp(eintrag->d_name, "..")==0) continue;
    strcpy(pfadp, eintrag->d_name);

    if (stat(pfadpuffer, &statbuf) == -1) {
        perror(pfadpuffer);
    } else if (S_ISDIR(statbuf.st_mode)) {
        printf("[%s]\n", pfadpuffer);
    } else {
        printf("%s (%ld Bytes)\n", pfadpuffer, statbuf.st_size);
    }
}
closedir(dir);
return EXIT_SUCCESS;
}
```

**pfadp**

/home/frieda/bsp.c\0

**S\_ISDIR(m)** ist „wahr“, wenn **m** der **st\_mode**-Wert eines Verzeichnisses ist (siehe: man stat)

# UNIX: link(), unlink()



```
#include <unistd.h>
int link(char *oldpath, char *newpath);
int unlink(char *pathname);
```

- **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 alter\_eintrag neuer\_eintrag**
- **unlink()** erniedrigt den Referenzzähler im zugehörigen inode und löscht den Verzeichniseintrag (sowie die Datei, falls der Referenzzähler auf 0 gefallen ist)
- Verwendet z.B. im Shell-Kommando „**rm**“

```
$ ls -l /usr/bin
-rwxr-xr-x  2 root    root      2003 Jun 23  2002 zdiff
-rwxr-xr-x  3 root    root      3029 Jun 23  2002 zegrep
-rwxr-xr-x  3 root    root      3029 Jun 23  2002 zfgrep
-rwxr-xr-x  1 root    root      1016 Jun 23  2002 zforce
-rwxr-xr-x  3 root    root      3029 Jun 23  2002 zgrep
-rwxr-xr-x  1 root    root      8408 Aug  4  2002 zic2xpm
-rwxr-xr-x  1 root    root     64344 Jun 24  2002 zip
```

- ls -l zeigt Anzahl der Verweise (Links) auf den inode einer Datei
- Option „-li“ zeigt zusätzlich inode-Nummer → so werden verschiedene Verweise auf denselben inode erkennbar:

```
$ cd /usr/bin; ls -li z*grep
376494 -rwxr-xr-x 3 root    root      3029 Jun 23  2002 zegrep
376494 -rwxr-xr-x 3 root    root      3029 Jun 23  2002 zfgrep
376494 -rwxr-xr-x 3 root    root      3029 Jun 23  2002 zgrep
377039 -rwxr-xr-x 1 root    root      1180 Jun 24  2002 zipgrep
```



# UNIX: **symlink()**, **readlink()**

```
#include <unistd.h>
int symlink(char *oldpath, char *newpath);
int readlink(char *path, char *buf, size_t bufsiz);
```

- **symlink()** legt symbolischen Verweis **newpath** an, der auf **oldpath** verweist (symbolischer Link, symlink)  
(Shell-Kommando: **ln -s oldpath newpath**)
- **readlink()** liest Verweis aus Symlink **path** in Zeichenvektor **buf** (max. **bufsiz** Zeichen)
- Für beide Funktionen: Ergebnis 0 für „ok“, -1 für Fehler
- Unterschiede zu Hard-Links:
  - Verweis **per Namen**, nicht inode (ändert Referenzzähler nicht)
  - Auflösung zur Laufzeit nötig, evtl. mehrstufig (s.u.)
  - Verweise über Filesystem- und Partitions Grenzen hinweg möglich (warum geht das mit harten Links nicht?)



# Symbolische Links: ls -l

```
$ cd /usr/lib
$ ls -l sendmail
lrwxrwxrwx    1 root    root           16 Apr  6 10:21
sendmail -> ../sbin/sendmail

$ ls -l ../sbin/sendmail
lrwxrwxrwx    1 root    root           21 Feb 14 00:47
../sbin/sendmail -> /etc/alternatives/mta

$ ls -l /etc/alternatives/mta
lrwxrwxrwx    1 root    root           27 Apr  6 10:21
/etc/alternatives/mta -> /usr/sbin/sendmail.sendmail

$ ls -l /usr/sbin/sendmail.sendmail
-rwxr-sr-x    1 root    smmsp       818943 Mar 26 11:19
/usr/sbin/sendmail.sendmail
```

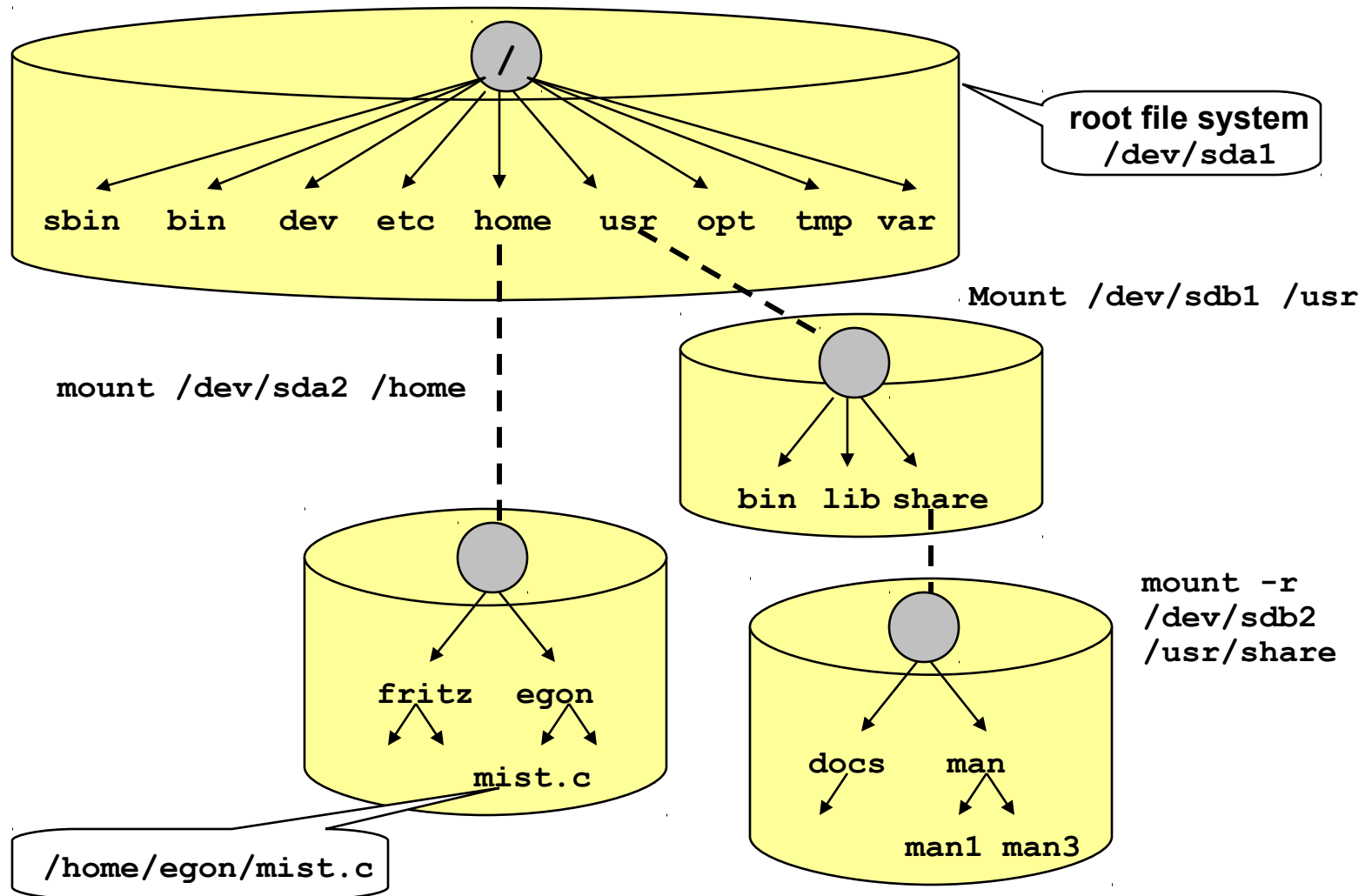
- Verweisziel wird bei symbolischen Links von `ls` angezeigt („->“)
- Typkennzeichen in ls-Ausgabe: „l“ (symLink)
- Hier: Zugriff auf `/usr/lib/sendmail` führt *letztlich* auf `/usr/sbin/sendmail.sendmail`



## 10.2.4 Montierbare Verzeichnisbäume

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

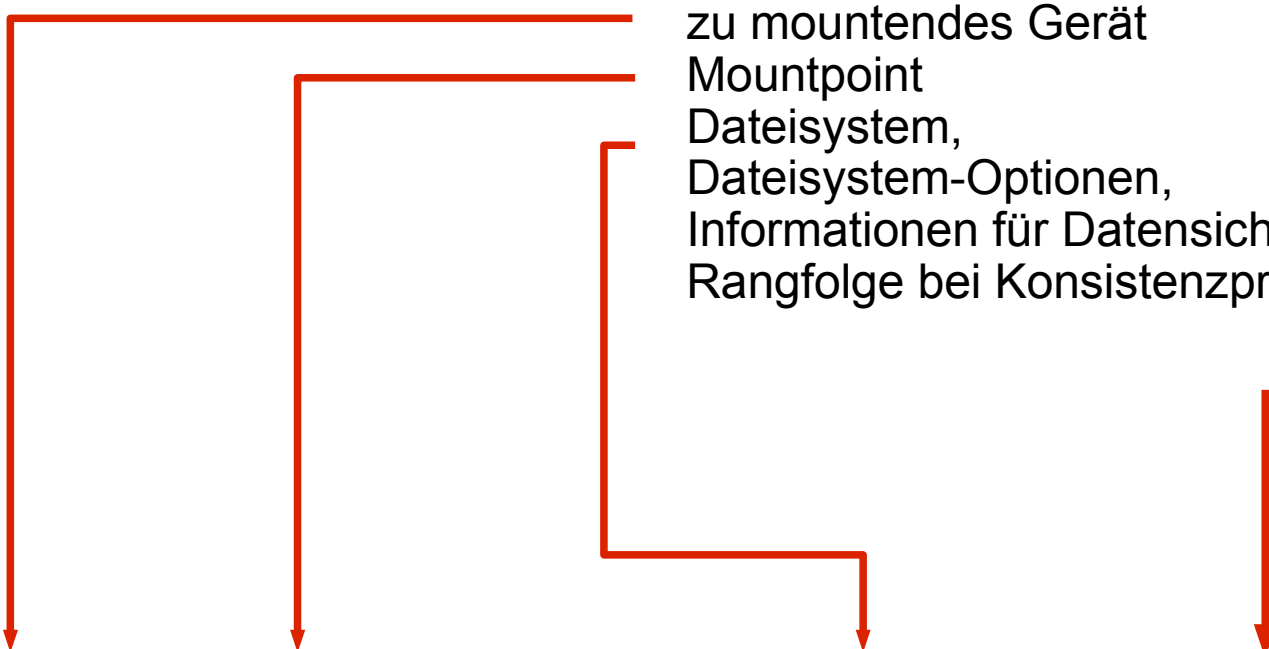




# Beispiel: /etc/fstab



zu mountendes Gerät  
Mountpoint  
Dateisystem,  
Dateisystem-Optionen,  
Informationen für Datensicherung (dump),  
Rangfolge bei Konsistenzprüfung



<code>/dev/sda3</code>	<code>/</code>	<code>ext4</code>	<code>errors=remount-ro</code>	<code>0</code>	<code>1</code>
<code>/dev/sda2</code>	<code>none</code>	<code>swap</code>	<code>sw</code>	<code>0</code>	<code>0</code>
<code>/dev/sr0</code>	<code>/media/cdrom0</code>	<code>udf,iso9660</code>	<code>user,noauto</code>	<code>0</code>	<code>0</code>
<code>/dev/sda1</code>	<code>/win</code>	<code>vfat</code>	<code>defaults</code>	<code>0</code>	<code>0</code>
<code>/dev/sdb1</code>	<code>/local</code>	<code>ext4</code>	<code>defaults</code>	<code>0</code>	<code>0</code>



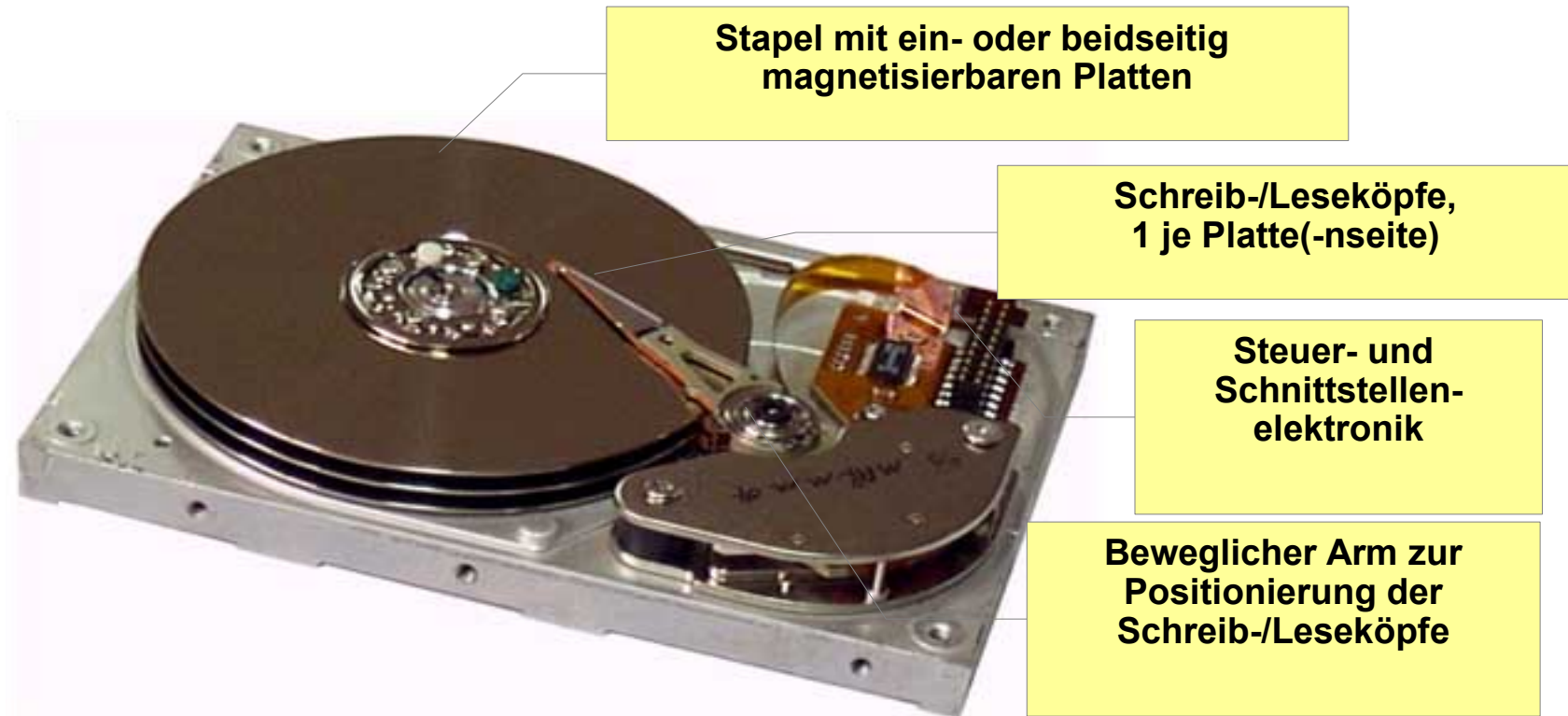
# Verbreitete Dateisysteme für Festplatten:

- BSD Fast File System (=UFS, historisch), Linux ext2 (verwandt)
- FAT16 (z.T. historisch), FAT32, NTFS (Windows)
- Linux Journaling File Systems (Logging-basiert):  
JFS, XFS, ext3/4 (i.w. ext2+Journaling), ReiserFS
- Linux ab 2.6.28: ext4 (i.w. ext3-Kompat+Extents+größereFS+...)
- Vergleich: [http://en.wikipedia.org/wiki/Comparison\\_of\\_file\\_systems](http://en.wikipedia.org/wiki/Comparison_of_file_systems)
- Im folgenden Sichtweise eines Implementierers
- Gliederung
  1. Technische Gegebenheiten
  2. Implementierung von Dateien
  3. Implementierung von Verzeichnissen
  4. Plattenplatz-Verwaltung
  5. Repräsentierung im Hauptspeicher
  6. Zuverlässigkeit des Dateisystems
  7. Performance des Dateisystems

# Speichermedium: Magnetplatte



Hochschule RheinMain  
University of Applied Sciences  
Wiesbaden Rüsselsheim

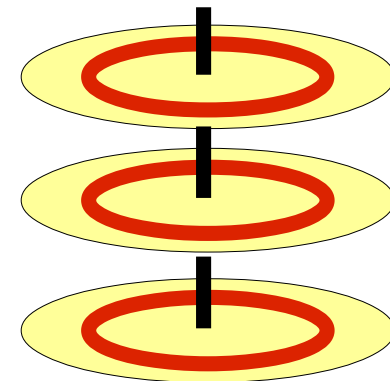
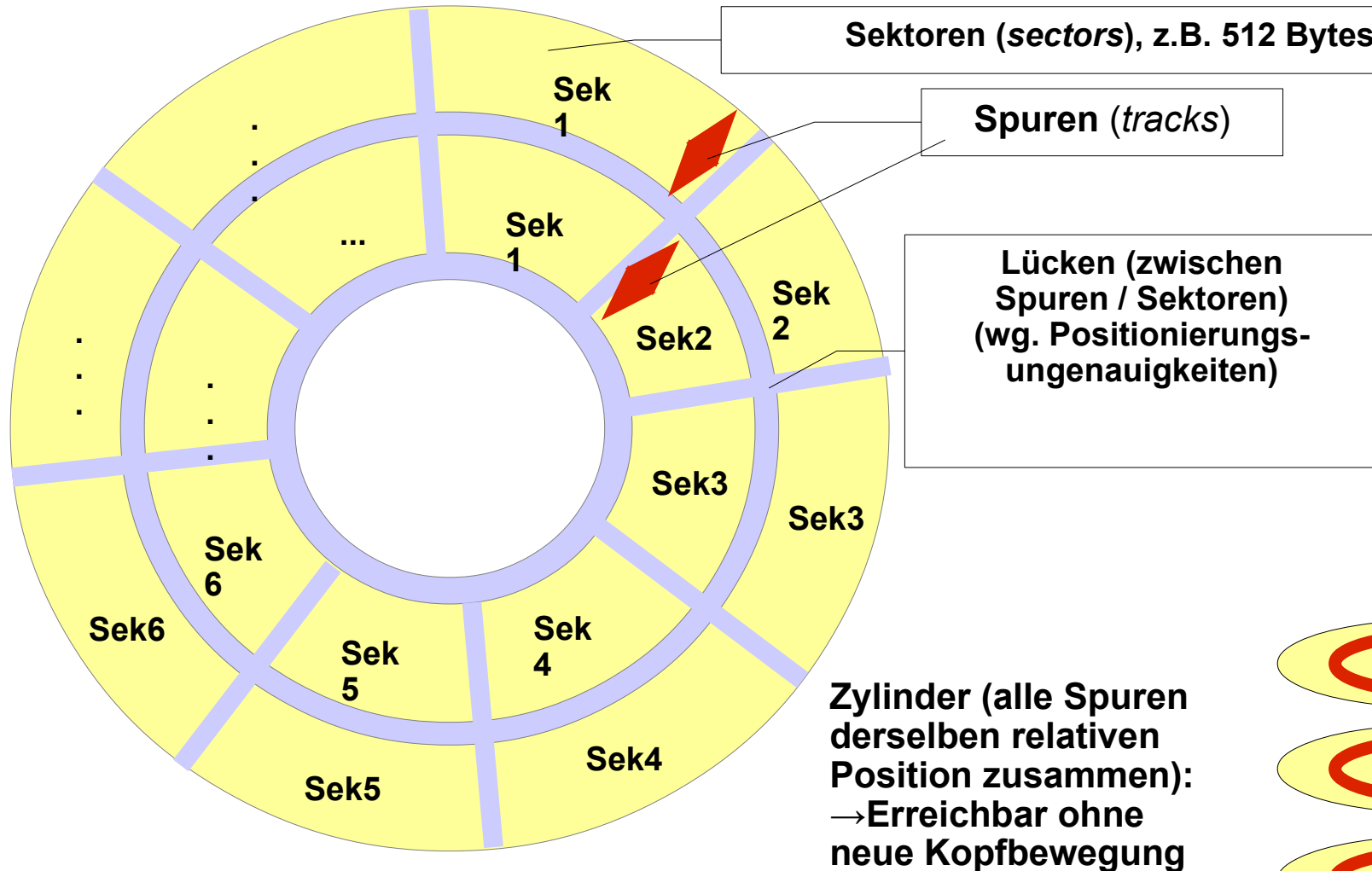


- 5400 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

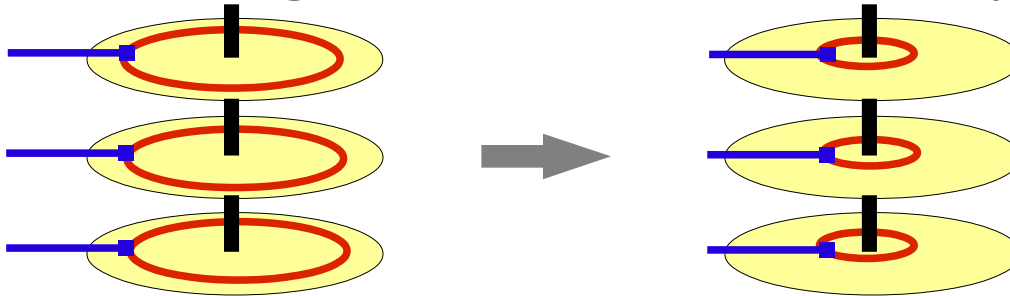
# Magnetplatte: Datenorganisation



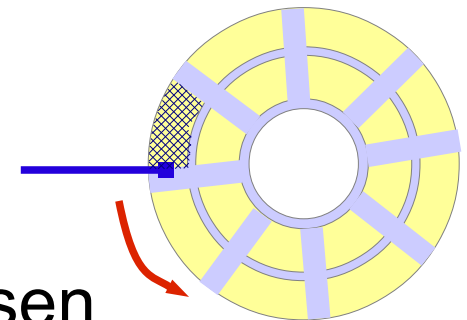
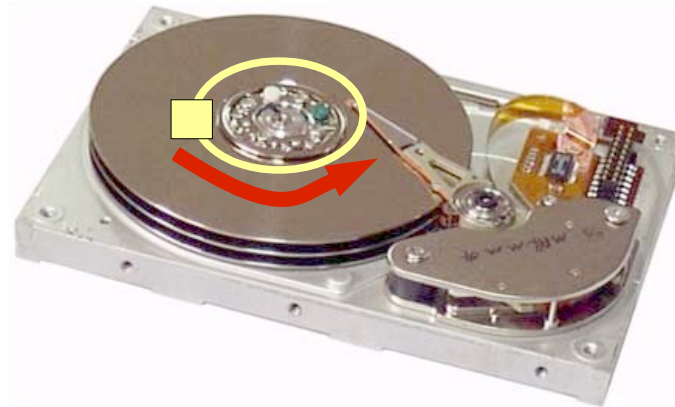
Hochschule RheinMain  
University of Applied Sciences  
Wiesbaden Rüsselsheim



- **Positionierungszeit:** Arm über Ziel-Zylinder bewegen



- **Rotationsverzögerung,** bis gesuchter Sektor unter Kopf ist



- Zeit zur **Datenübertragung** beim Auslesen

- 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 (gilt auch für Halbleiter-„Festplatten“ (SSD))
- Adressierung der Blöcke erfolgt prinzipiell über Zylinder-, Kopf- und Sektornummern (C/H/S)
- Heutige Festplatten verwenden jedoch logische Blockadressen (LBA), die ursprünglich folgendermaßen gebildet wurden:  
$$\text{LBA} = (\text{C} * \text{NumHeads} + \text{H}) * \text{BlocksPerTrack} + \text{S}$$
- Hierdurch können auch Geometriedaten der Festplatte, Eigenschaften wie z.B. eine variable Anzahl `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
- Moderne SSD-„Platten“ haben keine mechanischen Komponenten mehr  
→ Verzögerungen durch Kopfbewegungen treten nicht auf  
→ Datentransfergeschwindigkeit ist i.W. Durch die Schnittstelle und die Eigenheiten des Halbleiter-Speichermediums bestimmt.

# Beispiel: Festplattendaten



Hochschule RheinMain  
University of Applied Sciences  
Wiesbaden Rüsselsheim

## Hard Disk Drive Device Technical Details

- Hard Disk Model: HITACHI HDS5C1010CLA382
- Disk Family: Deskstar 5K1000
- Form Factor: 3.5"
- Capacity: 1000 GB (1000 x 1 000 000 000 bytes)
- **Number Of Disks: 2**
- **Number Of Heads: 4**
- Rotational Speed: 5700 RPM
- Rotation Time: 10.53 ms
- Average Rotational Latency 5.26 ms
- Disk Interface: Serial-ATA/300
- Buffer-Host Max. Rate: 300 MB/seconds
- Buffer Size: 8192 KB
- Drive Ready Time (typical): 10 seconds
- **Average Seek Time: 14 ms**
- **Track To Track Seek Time: 0.8 ms**
- Width: 101.6 mm (4.00 inch)
- Depth: 147 mm (5.79 inch)
- Height: 26.1 mm (1.03 inch)
- Weight: 680 grams (1.50 pounds)
- Required Power For Spinup: 3300 mA
- Power Required (Seek): 7 W
- Power Required (Idle): 5 W
- Power Required (Standby): 2 W
- Manufacturer: Hitachi Global Storage Technologies

**Grenze für Roh-Datenrate**

**Spitzen-Datenrate**  
(Schnittstellen-Eigenschaft  
→ gilt auch für SSD-„Platten“)

**Bedingt durch Kopfbewegung**  
→ Spitzenrate wird nicht dauerhaft erreicht → Vorteil von SSD



**Will der Benutzer Sektoren, Spuren etc. selbst ansteuern,  
Seine Daten in 512-Byte-Blöcke aufteilen müssen usw?  
Wohl kaum. Er will ...**

- Mithilfe der in 10.1 beschriebenen Funktionen Dateien und Verzeichnisse bearbeiten und verwalten
- optimale **Hardwarenutzung**  
(... natürlich ohne eigene Hardware-Kenntnisse)
- **einheitlichen Zugang** zu *vielen* (verschiedenen) Speichergerätearten (standardisierte Schnittstelle)



- Hauptproblem:

Verwaltung der Plattenblöcke und ihrer Zugehörigkeit zu einer Datei.

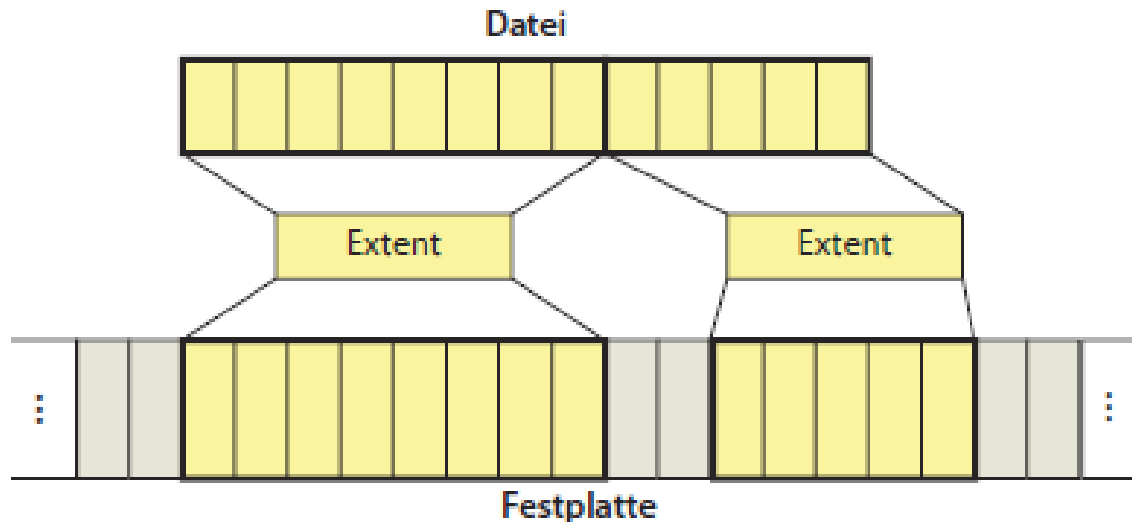
- Alternativen:

- Kontinuierliche Allokation.
- Allokation mittels einer verketteten Liste.
- Allokation mittels einer verketteten Liste und einem Index.
- Allokation mittels Index Nodes.



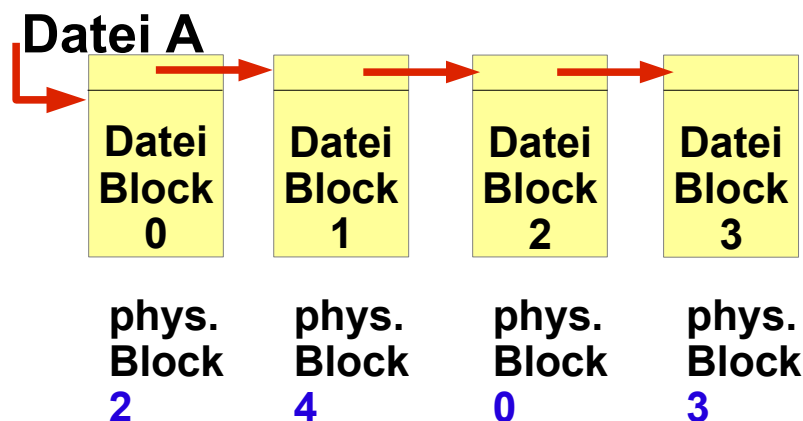
- Jeder Datei wird eine Menge zusammenhängender Plattenblöcke zugeordnet (beim Anlegen *reserviert*).
- Vorteile:
  - Einfach zu implementieren (nur die Adresse des ersten Blocks ist zu speichern).
  - Sehr gute Performance beim Lesen und Schreiben der Datei (minimale Kopfbewegungen).
- Nachteile:
  - Max. Größe der Datei muss zum Erzeugungszeitpunkt bekannt sein.
  - Externe Fragmentierung möglich.
  - Verdichtung extrem aufwendig.
- Anwendbarkeit:
  - Echtzeit-Anwendungen (Contiguous Files)
  - Write-Once Dateisysteme (CD/DVD, Logs, Backups, Versionierung).

- Extent: Stück einer Datei wird als Folge zusammenhängender Plattenblöcke gespeichert
- Datei besteht aus Folge von Extents
- Vorteile kontinuierlicher Allokation bleiben i.w. erhalten
- Nachteile werden z.T. durch persistente Präallokation umgangen



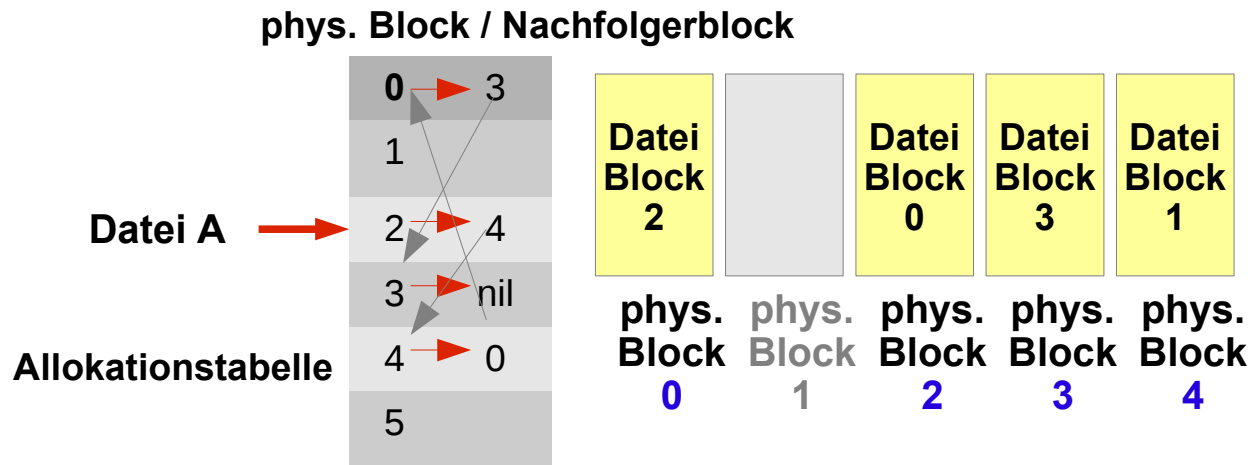
# Allokation mittels verketteter Liste

- Datei: Speicherblöcke 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



- Vor-/Nachteile:
  - + keine externe Fragmentierung
  - Wahlfreier Zugriff seeehr langsam
  - Es steht nicht der gesamte Datenblock für Daten zur Verfügung

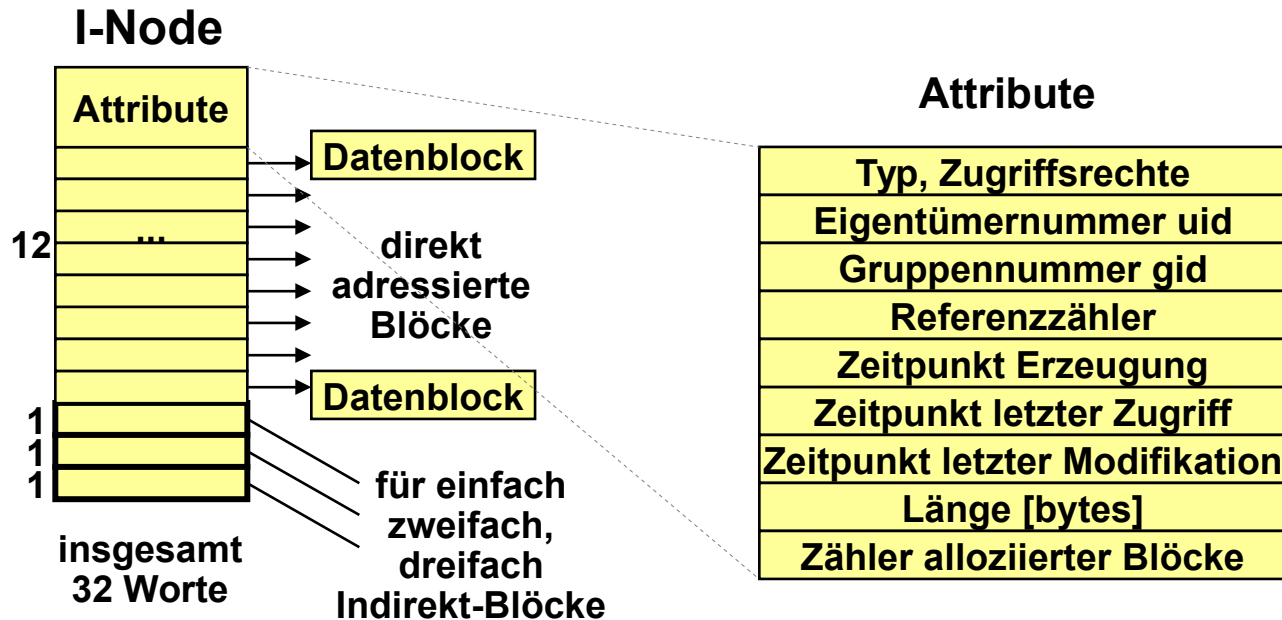
- Speicherung der Verkettungsinformation in einer separaten Tabelle (Index oder File Allocation Table = FAT).



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

**Def**

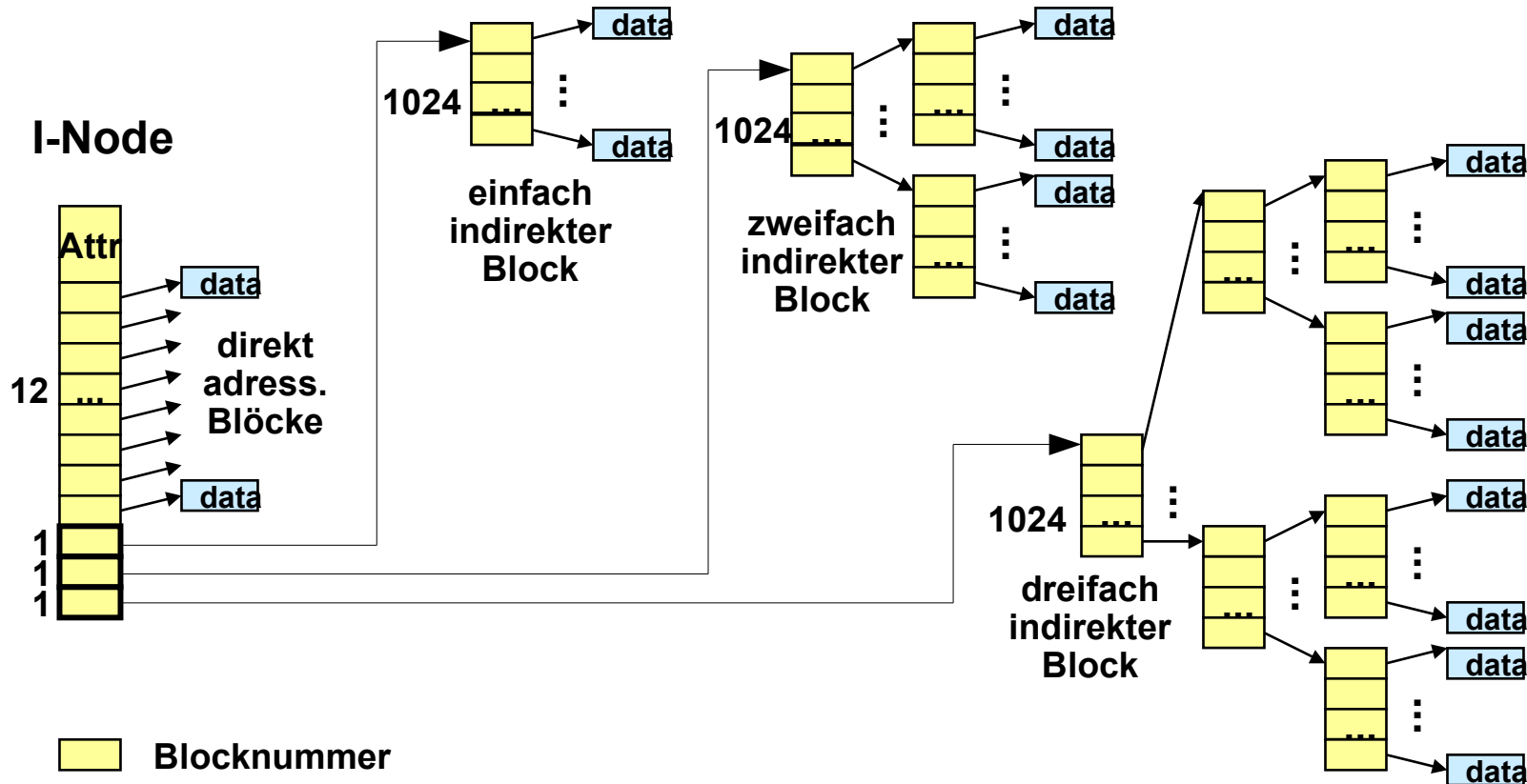
- Ein I-Node (UNIX: Inode) oder Index Node ist ein Dateikontrollblock
  - enthält neben Attributen der Datei kleine Tabelle mit Adressen von zugeordneten Plattenblöcken.
  - Ursprung Beispiel: BSD UNIX Fast File System (ufs)



# Allokation mittels Index Nodes (2)



- Nutzung von Indirekt-Blöcken





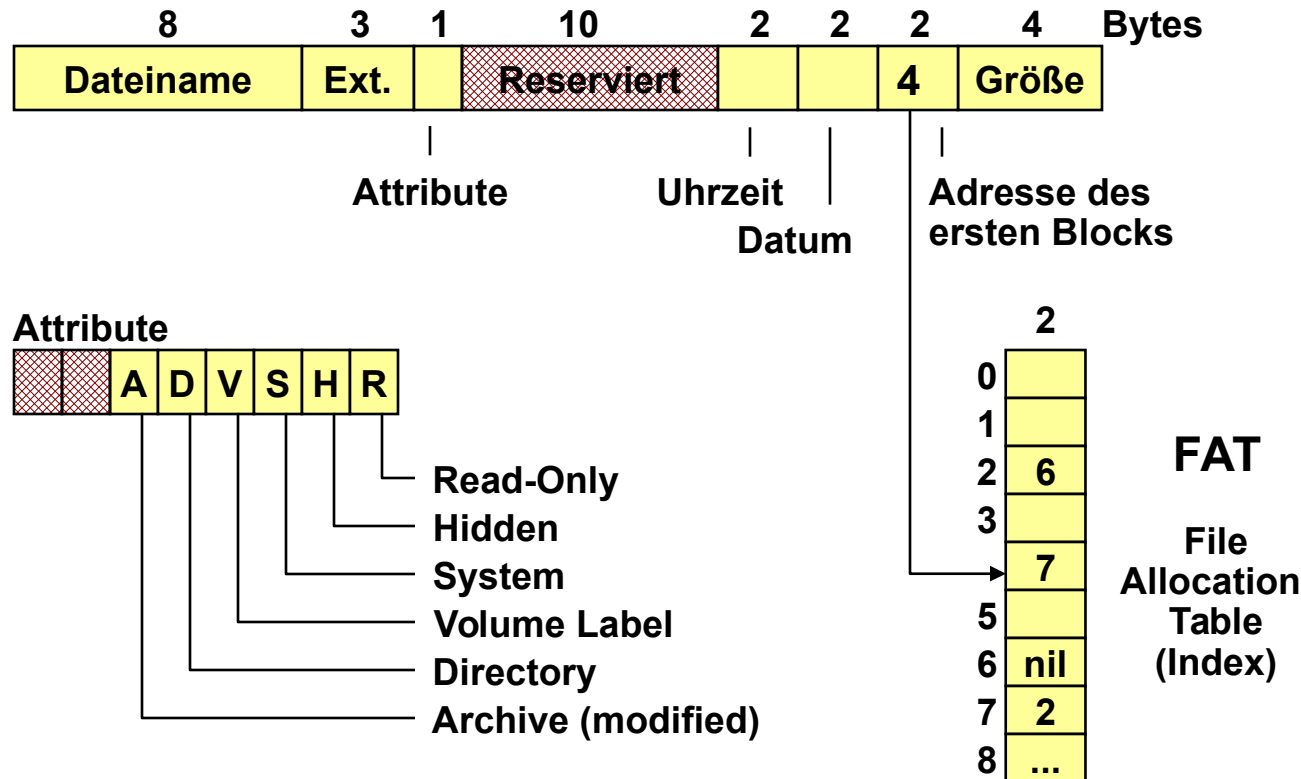


- Logische Blocknummern einer Datei
  - fortlaufend vergeben
  - beginnend bei den direkt adressierten Blöcken.
- Wenige Blockadressen im Inode selbst gespeichert.
  - ⇒ Nach Öffnen einer Datei und damit verbundenem Einlagern des Inodes in den Hauptspeicher stehen diese Blockadressen unmittelbar zur Verfügung.
  - ⇒ Schneller Zugriff zu kleinen Dateien.
- Für größere bis sehr große Dateien werden nach und nach einfach, zweifach und dreifach indirekte Blöcke zur Speicherung verwendet.
  - ⇒ Sinkende Zugriffsgeschwindigkeit bei wachsender Dateigröße.
- Beispiel: BSD UNIX Fast File System (ufs), ext2, ext3
  - Typische Blockgröße: 4 KB oder 8 KB (4 KB bei ext2 und ext3)
  - Direkt adressierte Blöcke: 12 (= 48 KB oder 96 KB).
  - 1-fach, 2-fach und 3-fach indirekte Blöcke vorgesehen.
  - Indirekte Blöcke (4KB) speichern bis zu 1024 ( $=2^{10}$ ) weitere Verweise  
⇒ Maximale Dateigröße:  
 $(12 + 2^{10} + 2^{20} + 2^{30}) * 4 \text{ KB} = 48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB} \approx 4\text{TB}$

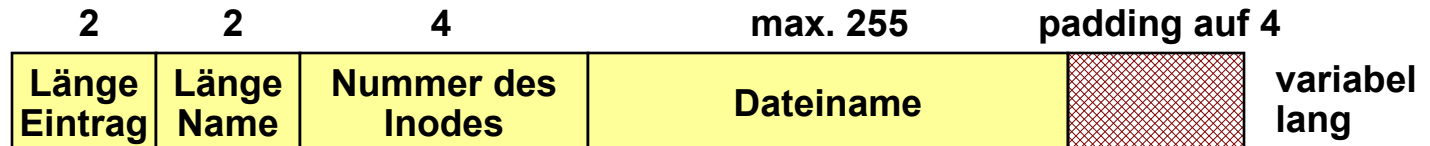


- Hauptaufgabe des Verzeichnissystems:  
Abbildung der Zeichenketten-Namen von Dateien in Informationen zur Lokalisierung der zugeordneten Plattenblöcke.
- Bei Pfadnamen werden die Teilnamen zwischen Separatoren schrittweise über eine Folge von Verzeichnissen umgewandelt.
- Verzeichniseintrag liefert 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.

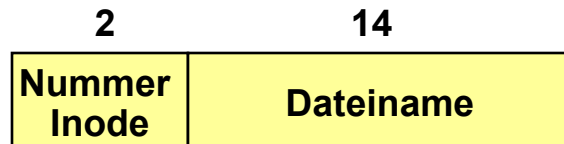
- Hierarchisches Verzeichnissystem.
- Allokation von Plattenblöcken mittels verketteter Liste und Index.
- Verzeichniseintrag:



- Hierarchisches Verzeichnissystem.
- Allokation von Plattenblöcken mittels Index Nodes.
- Verzeichniseintrag BSD UNIX Fast File System (ufs):



- Verzeichniseintrag klassisches UNIX System V (s5):



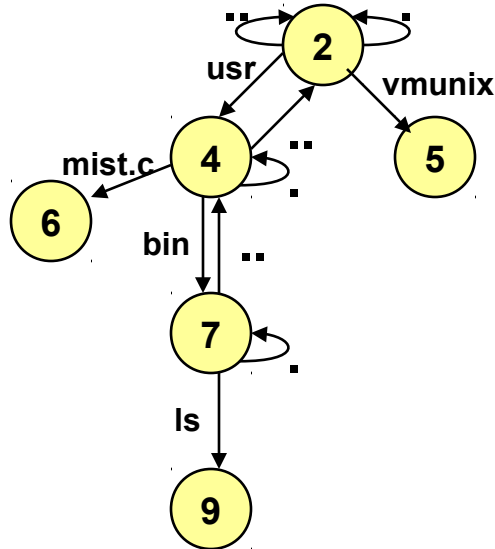
# Beispiel: UNIX (2)



## 10.3.2

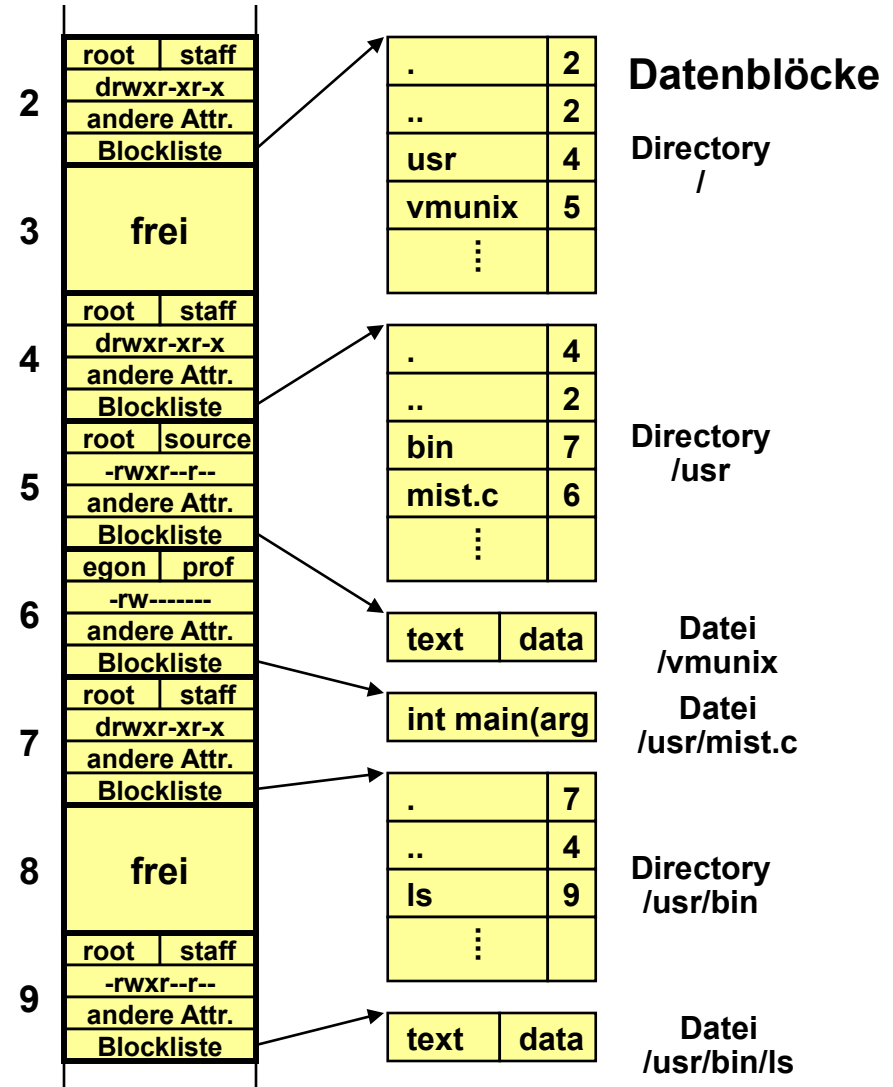
- Prinzip der Umsetzung eines Pfadnamens

### Logische Dateisystemstruktur



### Inode-Liste

aus Leffler et al: Design and Impl.  
of the 4.3BSD Operating System





### Hauptgesichtspunkte:

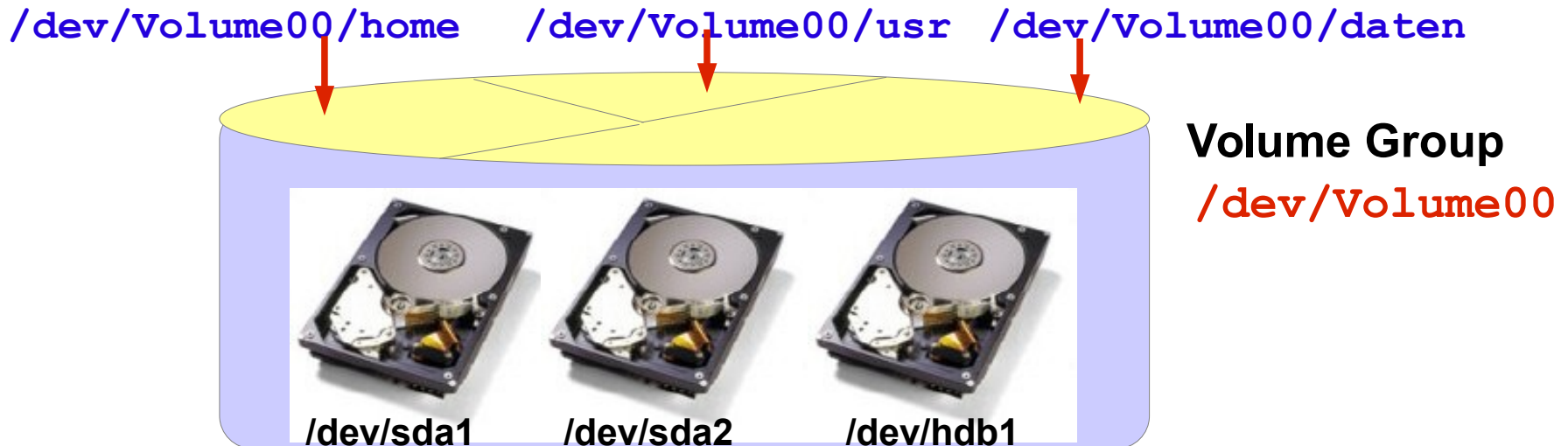
1. Logische Laufwerke.
2. Wahl der Blockgröße.
3. Dateisystemstruktur.
4. Quotas (Plattenplatz-Kontingentierung).

# 10.3.3.1 Logische Laufwerke



## Def

- Heutige Betriebssysteme erlauben, Logical Volumes zu erzeugen, die sich über mehrere physische Laufwerke erstrecken.
- Vorteil: sehr große Dateisysteme möglich.
- Beispiel; Linux **Logical Volume Manager** (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.



- Im laufenden Betrieb (!) ...
  - kann die Kapazität der Laufwerksgruppe durch **Hinzufügen weiterer physische Volumes** vergrößert werden
  - können **Daten** von alten Platten auf neue **verlagert** und die alten Platten 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



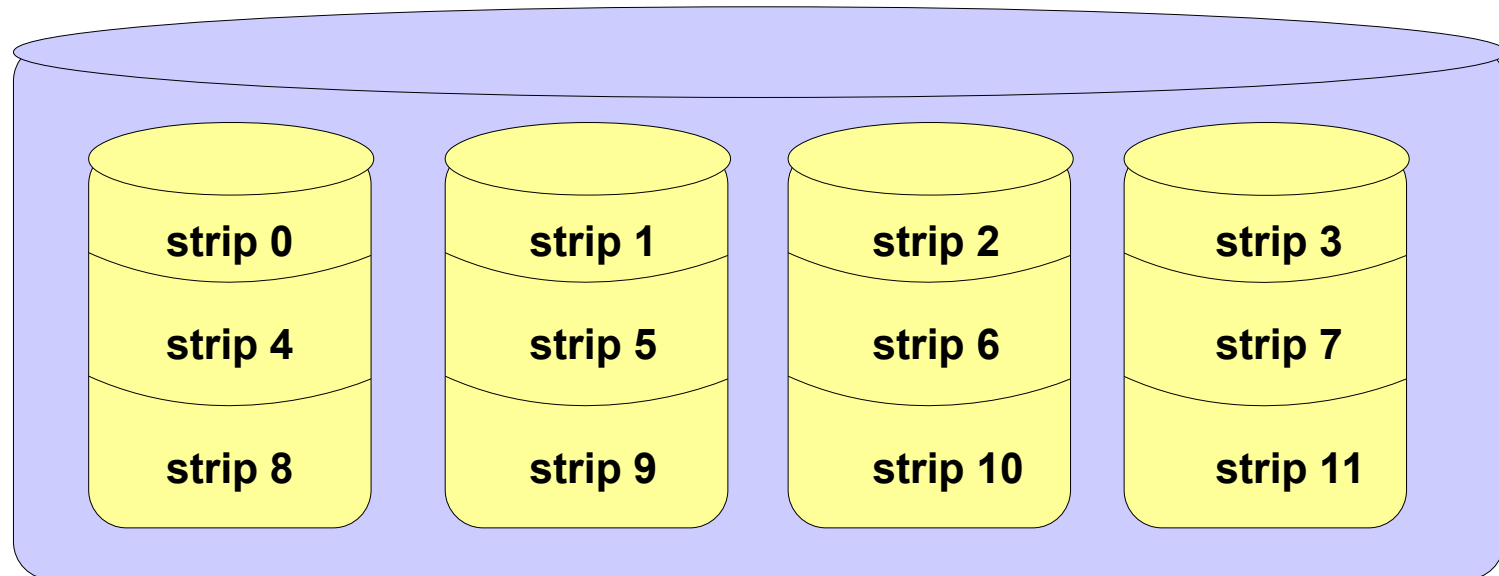


- Weitere Möglichkeit: RAID:  
*Redundant Array of Independent (Inexpensive) Disks*
- **Viele (preisgünstige) Platten** zusammengeschaltet, sehen für den Rechner **wie eine (sehr große) Platte** aus.
- **Realisierungen:**
  - Hardware-RAID (spezieller Festplatten-Controller)
  - Software-RAID (Betriebssystem verwaltet mehrere angeschlossenen Platten als RAID)
- **Erhöhung der Datensicherheit** durch geschickte redundante Speicherung möglich
- In der Regel 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

# RAID 0 - „striping“



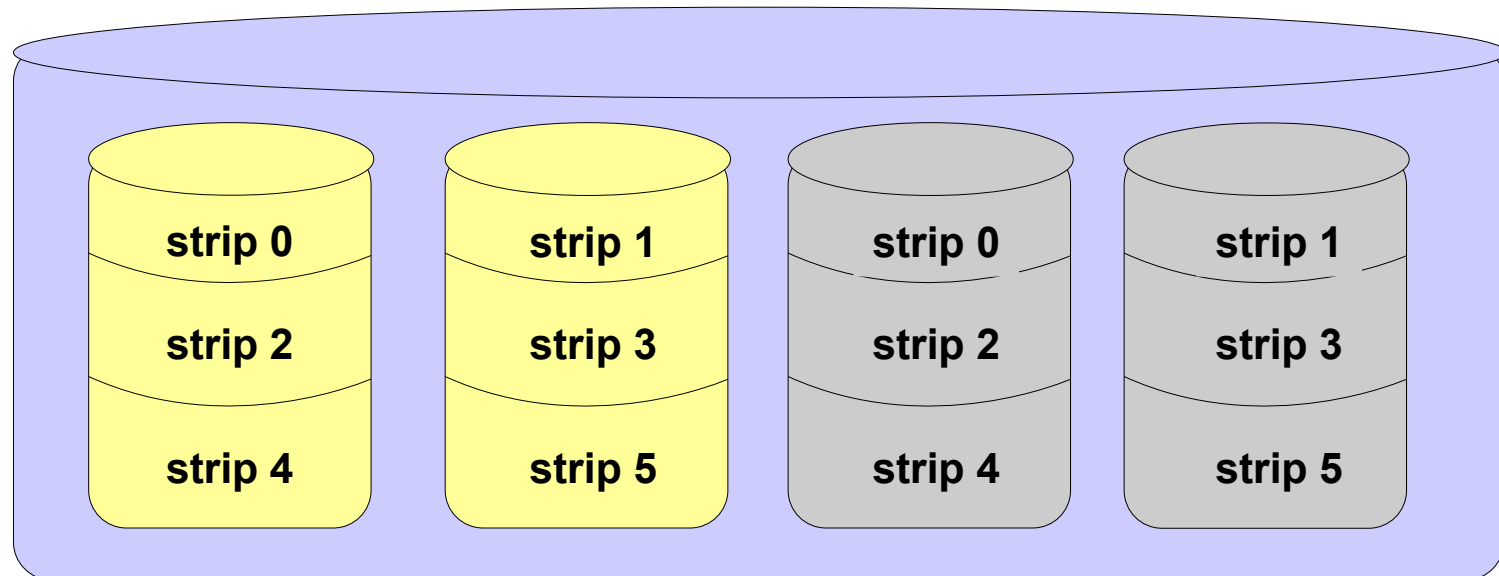
- RAID-Platte wird in „Streifen“ mit  $k$  Sektoren eingeteilt
- Streifen werden reihum auf den angeschlossenen Platten abgelegt.
- **keine Redundanz**, damit keine höhere Fehlertoleranz
- **Schneller Zugriff** besonders bei großen Dateien, da Platten parallel arbeiten können
- RAID-Kapazität: Summe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



# RAID 1 - „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
- Kapazität: Hälfte der addierten Plattenkapazitäten

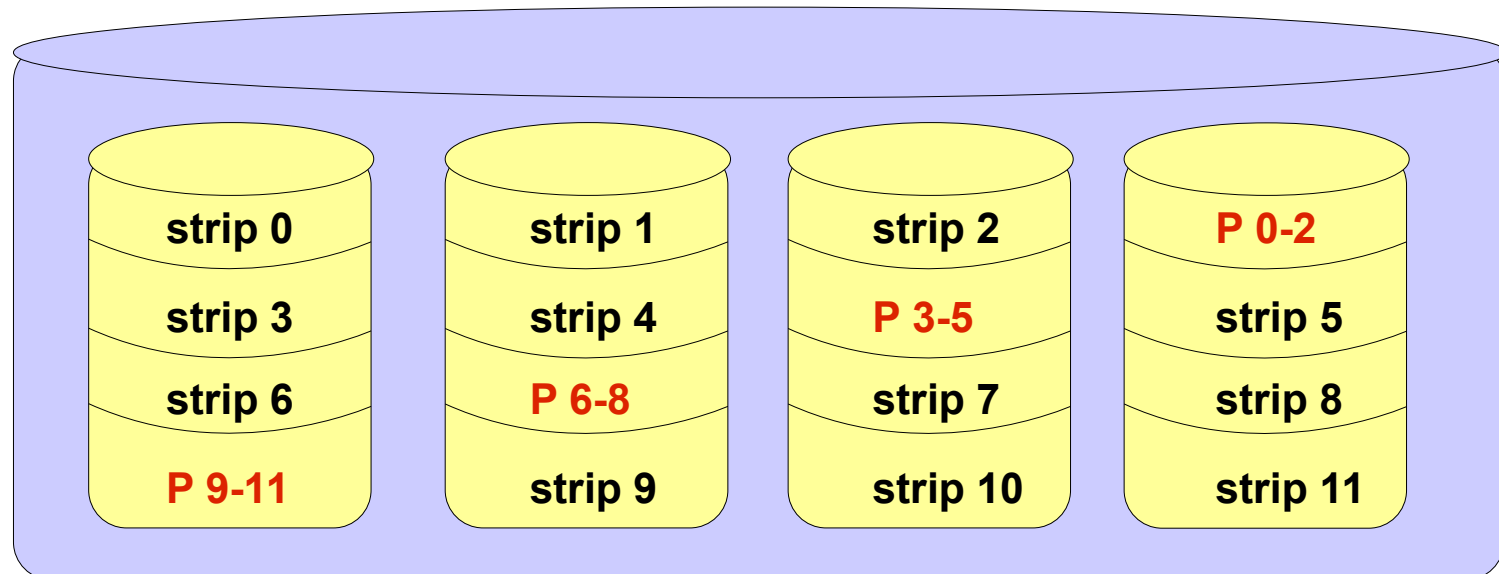


# RAID 5



- Paritätsinformation 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:  $A = B \oplus C \oplus P$
  - oder:  $B = A \oplus C \oplus P$
  - oder:  $C = A \oplus B \oplus P$
- Solange maximal eine (beliebige) Platte ausfällt, kann ihr Inhalt aus den Übrigen (im lfd. Betrieb) rekonstruiert werden (wieder per XOR)

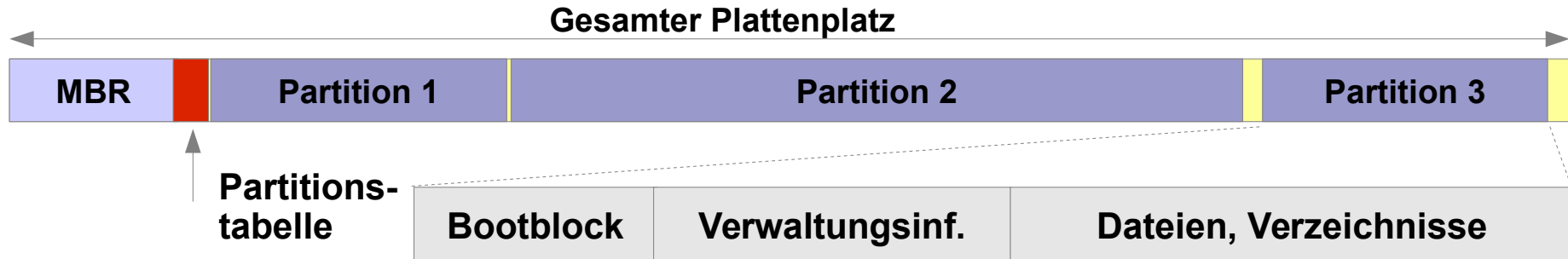
**+ Fehlertolerant bei  
guter Kapazitätsnutzung  
+ Leseoperationen schnell  
- Schreiben aufwändiger**



## Klassischer Fall:

**Def**

- Aufteilung eines physischen Laufwerks in mehrere logische Laufwerke (= Partitionen).
- Partitionierungsinformation (Partition Table) wird auf dem physischen Laufwerk gespeichert.
- Jede Partition kann genau ein Dateisystem aufnehmen.
- Dienstprogramme:
  - MS-DOS: `fdisk`
  - UNIX: `fdisk` (Linux), `chpt`, u.a.



- **MBR** (master boot record) enthält ausführbaren Code, der beim Systemstart vom BIOS (basic input/output system) geladen und gestartet wird.
- Dieser 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 (z.B. Boot-Menü)
- Die Partitionstabelle beschreibt die Aufteilung der Platte in Partitionen (Anfang, Länge, Typ, ggf. „bootbar“-Flag)

# Beispiel: Linux fdisk



## Partitionierungstabelle der Festplatte /dev/sda (=erste SATA-Festplatte)

```
$ fdisk /dev/sda
```

```
...
```

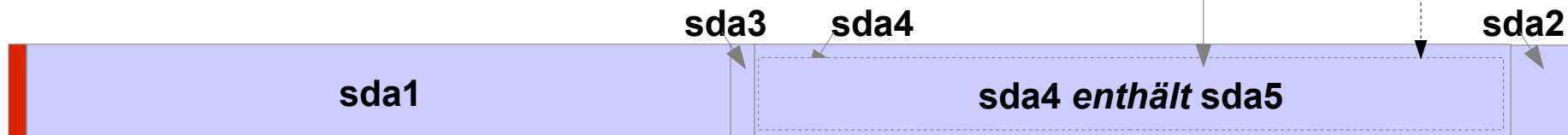
```
Befehl (m für Hilfe): print
```

```
Festplatte /dev/sda: 240 Köpfe, 63 Sektoren, 2584 Zylinder
```

```
Einheiten: Zylinder mit 15120 * 512 Bytes
```

Gerät	boot.	Anfang	Ende	Blöcke	Id	Dateisystemtyp
/dev/sda1		1	1163	8792248+	c	Win95 FAT32 (LBA)
/dev/sda2	*	2386	2584	1504440	c	Win95 FAT32 (LBA)
/dev/sda3		1164	1173	75600	83	Linux
/dev/sda4		1174	2385	9162720	f	Win95 Erw. (LBA)
/dev/sda5		1174	2385	9162688+	8e	Linux LVM

Partition table entries are not in disk order



## MBR

sda4 ist eine sogenannte „erweiterte Partition“, die Unterpartitionen (hier: sda5) enthalten kann

## 10.3.3.2 Wahl der Blockgröße

---



- Fast alle Dateisysteme bilden Dateien aus Plattenblöcken *fester* Länge.
- Plattenblock umfasst Folge von Sektoren mit aufeinanderfolgenden Adressen.
- Problem: Welches ist die optimale Blockgröße?
- Kandidaten aufgrund der Plattenorganisation sind:
  - Sektor        512 B (noch üblich).
  - Spur         z.B. 256 KB.
- Untersuchungen an Dateisystemen in UNIX-ähnlichen Systemen zeigen:
  - Die meisten Dateien sind klein ( $< 10$  KB) aber mit wachsender Tendenz.
  - In Hochschul-Umgebung im Mittel 1 KB (Tanenbaum).

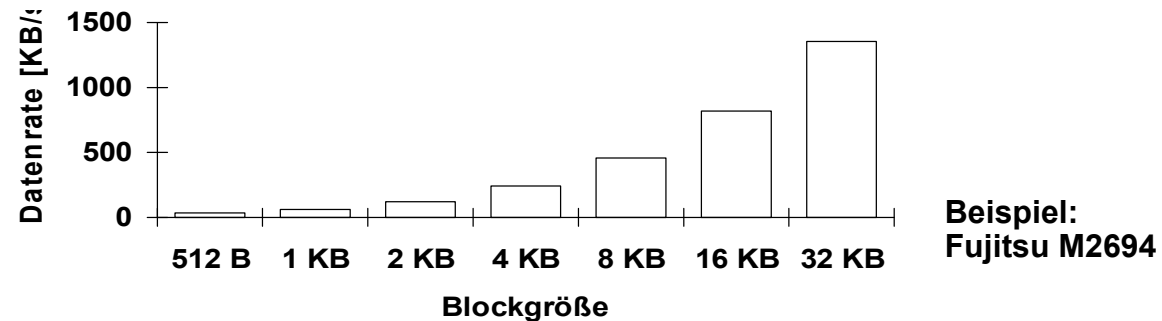




- Eine große Allokationseinheit (z.B. Spur) verschwendet daher zuviel Platz.
- Beispielrechnung: Verschwendeter Platz  
Daten basieren auf realen Dateien, Quelle [Leffler et al].

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

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



	Fujitsu M2694	Seagate Barracuda 7200.12
Jahr	1994	2009
Kapazität	1 GB	160-1000 GB
Drehzahl	5400 min <sup>-1</sup>	7200 min <sup>-1</sup>
Transferrate	4 MB/s	125 MB/s
Positionierzeit (Mittel)	10,0 ms	8,5 ms
Positionierzeit (track-to-track / Max)	2,5 ms / 22,0 ms	?
Latenzzeit (1/2 Umdrehung)	5,6 ms	4,2 ms

- 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 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
  - ➔ Kompatibilitätsprobleme (Lösung durch Emulation) können Performance-Nachteile haben



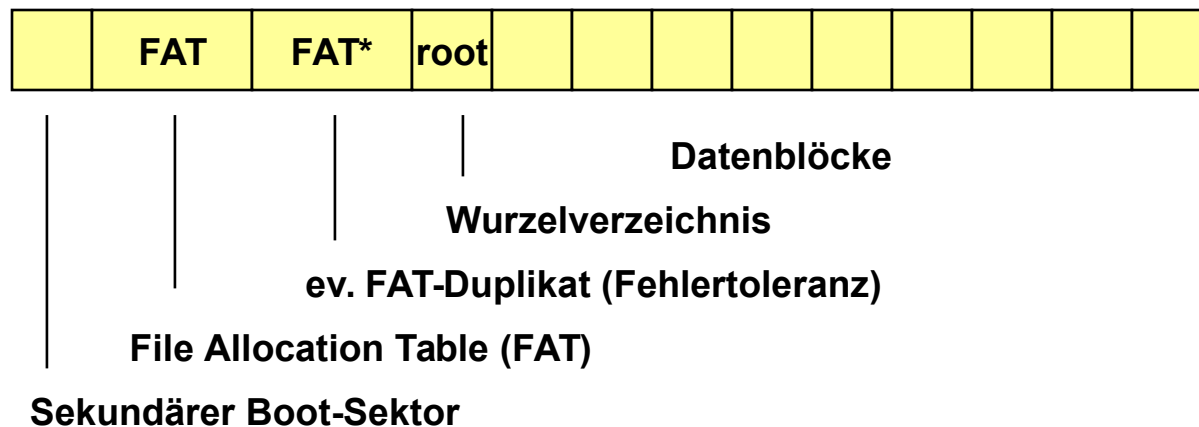


- Beispiel: BSD UNIX Fast File System (heute ähnlich ext2)
  - Einführung zweier Blockgrößen, genannt Block und Fragment als Teil eines Blocks (typ. heute 8 KB / 1 KB).
  - Eine Datei besteht aus ganzen Blöcken (falls nötig) sowie ein oder mehreren Fragmenten am Ende der Datei.
- ⇒ Transfer großer Dateien 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.

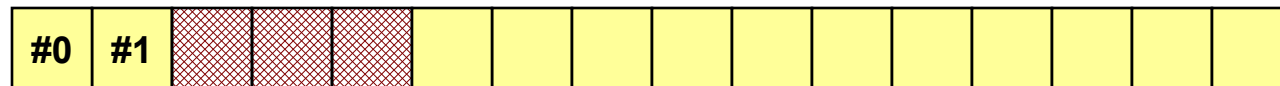
# 10.3.3.3 Dateisystemstruktur



- Die Struktur eines Dateisystems wird beim Erzeugen auf die Blockmenge eines logischen Laufwerks (Partition) aufgeprägt.
- Dienstprogramme zum Erzeugen:
  - MS-DOS: `format` (nicht zu verwechseln mit dem Formatieren eines Mediums, in MS-DOS low-level Formatierung genannt)
  - UNIX: `mkfs`, (`newfs`)
- Beispiel: MS-DOS



- Beispiel: klass. UNIX System V (s5)



I-Node-  
Blöcke

Datenblöcke

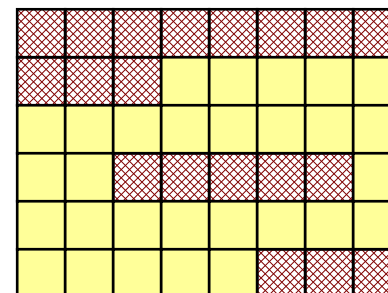
Superblock enthält Layoutbeschreibung:  
Blockgröße (512B, 1K, 2k), Anzahl Inodes, Anzahl Blöcke,  
Kopf der Freiliste, Kopf der Liste der freien Inodes

nicht Teil des Dateisystems, als Boot-Block reserviert

- Angewendete Methoden:
  - Verkettete Liste.  
Beispiele: MS-DOS FAT, UNIX System V
  - Bitmap.  
Vorteil: Größere freie Bereiche einfacher erkennbar.  
Beispiel: BSD UNIX Fast File System

11
39
4711
2814
9
134
999
21

**Verkettete  
Liste**



 **Belegt**     **Frei**

**Bitmap**

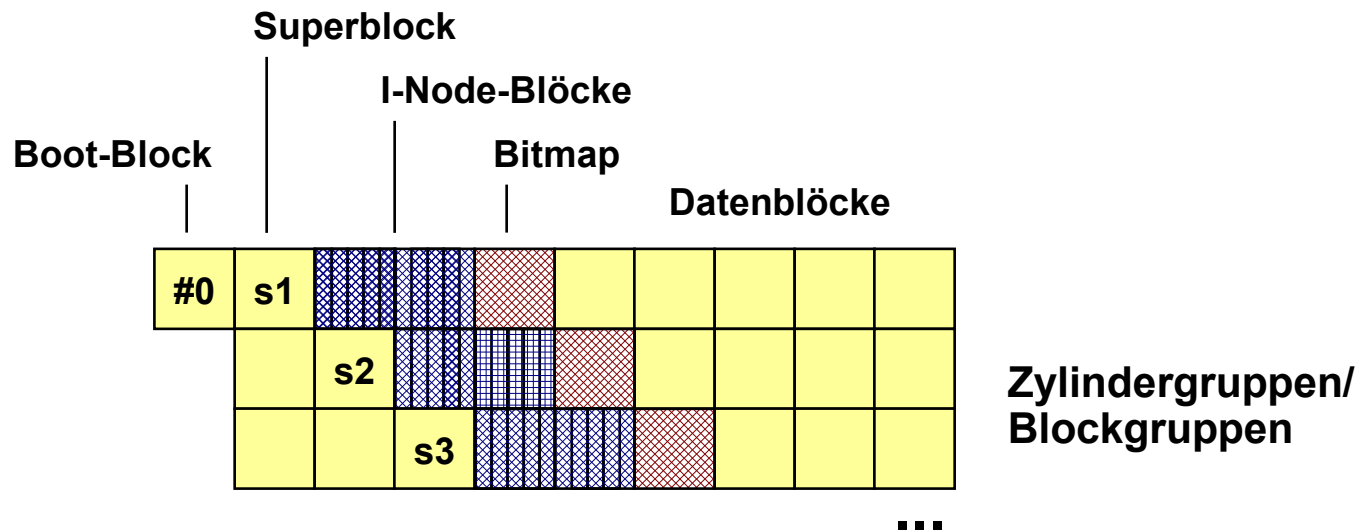
# Beispiel: BSD Fast File System (ufs)

10.3.3.3



Hochschule RheinMain  
University of Applied Sciences  
Wiesbaden Rüsselsheim

- Weitere Besonderheiten:
  - Einführung sogenannter Zylindergruppen/Blockgruppen, d.h. Mengen von aufeinanderfolgenden Zylindern/Blöcken innerhalb eines Dateisystems mit jeweils eigenen Verwaltungsstrukturen.
  - I-Node und zugehörige Datenblöcke sollen möglichst dicht beisammen bleiben (Performance).



Redundante Kopien s1, s2, ... des Superblocks in jeder Zylindergruppe an verschiedenen Stellen (Kopfpositionen) zur Verbesserung der Verfügbarkeit der Layoutinformation auch bei Plattenfehlern.





- Zuordnungstrategie:
  - Normalfall:  
Vergrößerung einer Datei erfolgt bei Bedarf  
Block für Block.
  - Neuere Zuordnungsstrategie  
in DEC OSF/1 Advanced File System:  
In Folge 1, 2, 3, 6, 12, 32, 32, ... Blöcke von jeweils 8 KB  
zugewiesen.  
⇒ Bei großen Dateien weniger Zuordnungsvorgänge  
notwendig.



- 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.



- 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).

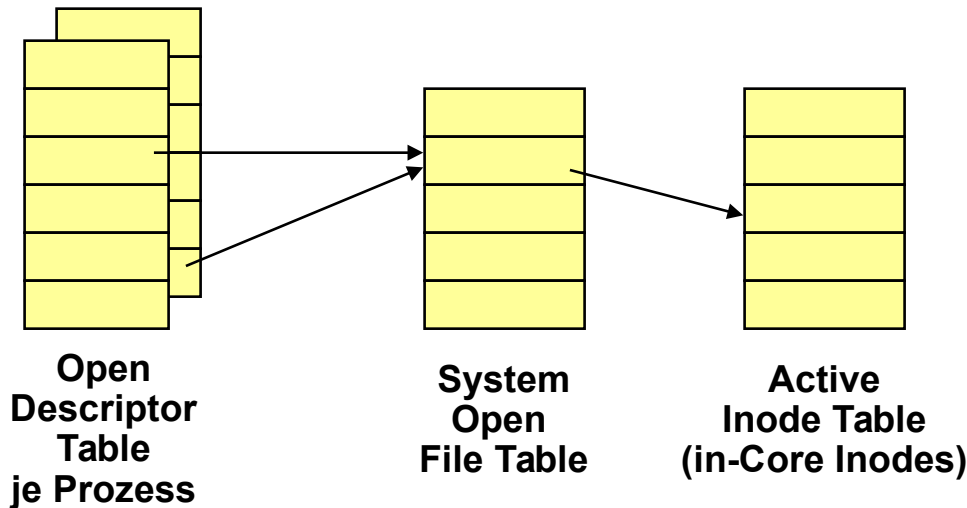
## 10.3.4 Repräsentierung im Hauptspeicher

---



- Bisher wurde die Repräsentierung von Dateien und Verzeichnissen auf dem Hintergrundspeicher betrachtet.
- Geöffnete Dateien besitzen eine Repräsentierung im Arbeitsspeicher
  - mit Informationen des Dateikontrollblocks des Hintergrundspeichers
  - einige zusätzliche Informationen
  - Felder für die Verkettung der Deskriptoren
- Repräsentierung von Dateien im Hauptspeicher wird im folgenden am Beispiel BSD UNIX konkretisiert.

- Überblick:



Teil der User Structure,  
auslagerbar

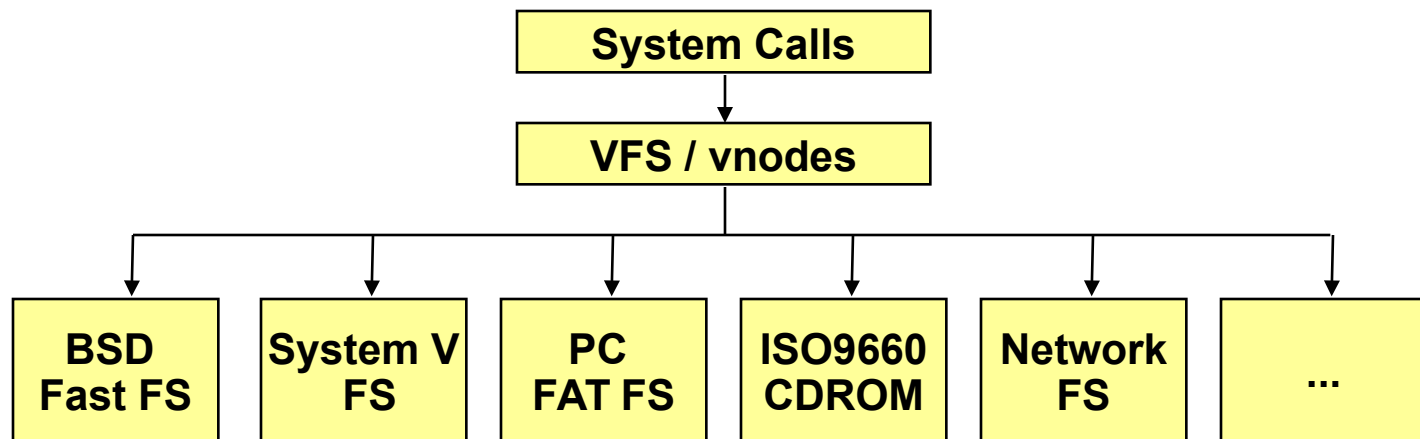
**Organisation:**

**Hash-Listen  
gemäß Schlüssel  
(inode#, device#)**

**Unbenutzte Inodes  
gemäß LRU verkettet.**

Die System Open File Table enthält insbesondere die Datei-Position jeder geöffneten Datei. Da geöffnete Dateien bei fork vererbt werden, ist die einmalige Verarbeitung des Dateiinhalts durch Vater und Sohn oder durch mehrere Söhne möglich. Erneutes Öffnen derselben Datei resultiert dagegen in einem neuen Eintrag mit unabhängigem Positionszeiger.

- Innerhalb des BS-Kerns wurde neue Schnittstelle des Dateisystems eingezoogen, 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 System V /proc-Prozessdateisystem (jedem aktiven Adressraum entspricht eine Datei) oder RAM-Disk.





### Hauptgesichtspunkte:

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

- Festplatten haben i.d.R. von Anfang an fehlerhafte Sektoren (z.B. aufgrund ungleichmäßiger Magnetisierung der Oberflächen).
- Fehlerhafte Sektoren werden beim Formatieren der Platte (Aufbau der Sektoren) festgestellt. (Im PC-Umfeld wird das Formatieren auch low-level-Formatierung genannt).
- Verzeichnis der fehlerhaften Sektoren wird Media Defect List genannt.
- Hardware-Lösung:
  - Media Defect List wird auf Platte selbst geführt.
  - Jedem defekten Sektor wird Ersatzsektor (i.d.R. auf einem zusätzlichen Zylinder) zugeordnet, der statt des defekten Sektors benutzt wird.
  - Problem: evtl. unvorhersehbare Kopfbewegungen
- Software-Lösung:
  - Es wird eine Datei konstruiert, die nie gelesen oder geschrieben wird und der alle defekten Blöcke zugeordnet werden.



- Regelmäßige Datensicherung ist wichtig: Schutz vor Datenverlust im Falle von Defekten ...
  - Plattencrash
  - Feuer, Überschwemmung, ...
- ... nicht-redundanter Laufwerke oder Fehlern
  - Sabotage / Viren
  - Benutzer- / Programmierfehler.
- Einheit der Datensicherung sind i.d.R. Dateisysteme.
- Dienstprogramme in UNIX:  
dump/restore, backup (System V), tar
- Voller Dump - inkrementeller Dump.
- Dump-Strategie: z.B.
  - Voller Dump wöchentlich.
  - Inkrementeller Dump täglich.
  - Sicherungskopien in geeigneter Entfernung sicher aufbewahren
- Sicherung dauert lange → Problem mit aktiven Dateisystemen
  - ➔ Sicherung nachts / an Wochenenden
  - ➔ Volume Management (z.B. LVM, vgl. 10.3.3.1)

`$ rm * .bak`  
**rm: Entfernen von „.bak“  
nicht möglich: Datei oder  
Verzeichnis nicht gefunden**

## Def

- 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 ev. Wiederherstellung der Konsistenz bei ev. auftretendem Datenverlust.

- UNIX 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, vgl. 10.3.6) 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).

- Übliche Dienstprogramme zur Überprüfung / Reparatur der Konsistenz eines nicht benutzten Dateissystems bei möglichem Datenverlust: `fsck` (File System Check), auch z.T. `ncheck` (inode check).
- In jüngerer Zeit neue Dateisystemtypen, als Journalled File Systems bezeichnet, die bzgl. der Meta-Informationen des Dateissystems ein Write-Ahead-Logging (analog Datenbanken) durchführen:
  - Konsistenz ist gewährleistet, z.B. bei Systemzusammenbruch aufgrund Stromausfall.
  - Kein `fsck` notwendig (Schneller Restart).
  - Beispiele: IBM AIX 3.2, DEC OSF/1 2.0, NT NTFS, Linux Reiser, Linux ext3.
- Zur Verbesserung der Verfügbarkeit von Daten im Falle von Plattenfehlern werden ebenfalls in jüngerer Zeit z.B. eingesetzt:
  - Spiegelplattenbetrieb (Disk Mirroring, vgl. 10.3.3.1).
  - RAID-Laufwerke (vgl. 10.3.3.1).

fsck führt verschiedene Konsistenzüberprüfungen durch:

- **Blocküberprüfung**

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

Blocknr																belegte Blöcke
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	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	1	1	1	0	0	1	0	1	freie Blöcke

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

# Blockprüfung: Fehler (1)



- Fehlender Block: Block 4 ist weder belegt noch frei?

- Maßnahme: Block zu freien Blöcken hinzunehmen

0					5					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

- Doppelter Block in Freiliste (Block 9)

- Maßnahme: Freiliste neu aufbauen

0					5					10					15	
1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0	
0	0	1	0	1	0	1	1	1	2	1	0	0	1	0	1	
																belegte Blöcke
																freie Blöcke

- Doppelter belegter Block (Block 12)

- Maßnahme: Block kopieren, Kopie-Block in eine der beiden betroffenen Dateien statt Block 12 einbauen

0					5					10					15	
1	1	0	1	0	1	0	0	0	0	0	1	2	0	1	0	
0	0	1	0	1	0	1	1	1	1	1	0	0	1	0	1	
																belegte Blöcke
																freie Blöcke



# Maßnahmen zur Performance-Steigerung:

- Zylindergruppen.
  - Ziel: Vermeiden von weiten Kopfbewegungen.
  - Beispiel: BSD Fast File System (vgl. 10.3.3.3).
- 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 % (4.3BSD UNIX).
  - 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.

- Dateinamens-Cache.
  - Ziel: Verringerung der Anzahl der Schritte bei der Abbildung von Dateipfadnamen 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.



## Gliederung

1. Einordnung und Begriffe
2. Angriffe auf die Sicherheit
3. Entwurfsprinzipien
4. Aufgabenbereiche
5. Beispiel: Benutzer-Authentifikation

# 10.4.1 Einordnung



- Das deutsche Wort Sicherheit beinhaltet:
  - Sicherheit gegen technische Fehler (technische Zuverlässigkeit) bei verschiedenen Betriebsbedingungen:
    - ➔ Normalbetrieb (Fehlertolerante Systeme).
    - ➔ Unfall, Feuer, Verbrechen (Back-up-Systeme).
    - ➔ Naturkatastrophen (Erweiterte Back-up-Lösungen).
    - ➔ Kriegsfall oder terroristische Angriffe (?).

Hauptziel: Ausschließen einer Gefährdung (Safety).
  - Sicherheit gegen menschliches Versagen wegen:
    - ➔ Überforderung.
    - ➔ Unaufmerksamkeit.
    - ➔ Fahrlässigkeit.
    - ➔ Mutwilligkeit.

Neben organisatorischen Aspekten häufig ein Problem der Mensch-Maschine-Schnittstelle.
  - Sicherheit gegen Ausspähung (im weiteren behandelt).
  - Sicherheit gegen Manipulation (im weiteren behandelt).

### Def

- Sicherheit (Security) bezeichnet das allgemeine Problem, Information nur nach festgelegten Regeln (Policies) berechtigten (autorisierten) Personen lesend oder verändernd zugänglich zu machen.  
Aspekte:
  - politische
  - juristische
  - organisatorische
  - technische (hier im Vordergrund).
- Privatsphäre (Privacy) als spezielles Sicherheitsproblem: Schutz des Individuums vor dem Missbrauch der über es gespeicherten Information.
- Schutzmechanismen (Protection Mechanisms) bezeichnen Verfahren in Rechensystemen, die dem Schutz von Informationen dienen, um Sicherheit zu erreichen.



- Verletzung von Sicherheit
  - durch Eindringlinge (Intruders).
  - Vorgang als Penetration bezeichnet.
- Passive Eindringlinge versuchen, Dateien zu lesen, ohne dass sie dazu autorisiert sind (Ausspähung).
- Aktive Eindringlinge versuchen nichtautorisierte Veränderungen an gespeicherter Information (Manipulation).

- Kategorien von Eindringlingen:
  - Gelegentliches Ausspionieren durch normale Benutzer.
  - Unterlaufen vorhandener Sicherheitsvorkehrungen durch Experten (persönliche Herausforderung, z.B. Hacker).
  - Betrug oder Diebstahl mit Rechnern als Hilfsmitteln (kriminelles Verhalten).
  - Kommerzielle oder militärische Spionage.
- Eindringlinge können außer Benutzern auch sein:
  - Betreiber.
  - Wartungspersonal.
  - Hersteller
  - Behörden (NSA)

## 10.4.3 Entwurfsprinzipien

---



- Erste systematische Behandlung durch Saltzer und Schroeder (1975) im Rahmen des MULTICS-Projekts am MIT.
- Regeln:
  - Der Entwurf des Systems sollte öffentlich zugänglich sein. Sicherheit, die auf Geheimhaltung vor möglichen Eindringlingen basiert, ist nie lange gewährleistet.
  - Jeglicher Zugriff muss explizit erlaubt werden.  
Default-Fall: kein Zugriff.
  - Ein einmal gewährter Zugriff muss wiederholt überprüft werden (Revalidierung).
  - Befristung oder Widerruf (Revocation) eines Rechts sollte möglich sein.
  - Jeder Prozess sollte so wenig Zugriffsrechte wie nötig haben (Least Privilege Principle oder Need-to-Know Principle).



- Der Schutzmechanismus sollte einfach ( $\Rightarrow$  verständlich), einheitlich und in den untersten Ebenen des Systems verankert sein. Sicherheit nachträglich in ein existierendes Betriebssystem einzubauen, gilt als aussichtslos.
- Trennung von Schutzstrategie (Policy) und Mechanismus.
  - Strategie legt fest, wessen Daten vor wem geschützt werden sollen
  - Mechanismus legt fest, wie (mit welchen Mitteln) diese Festlegungen durchgesetzt werden.
- Akzeptanz der Mechanismen durch die Benutzer muss gegeben sein.

# 10.4.4 Aufgabenbereiche

---



- Authentifikation:

Authentifikation (oder Authentifizierung) bedeutet, Gewissheit zu erlangen (Validierung) über die Identität eines Subjekts (vgl. 10.4.5).

- Autorisation:

Autorisation beinhaltet die Verwaltung von Berechtigungen für Subjekte zur Durchführung legaler Aktionen (vgl. 10.5).

- Kryptographie:

Kryptographie beinhaltet die Verschlüsselung gespeicherter und übertragener Informationen einschließlich entsprechender Schlüsselverwaltungssysteme (hier nicht behandelt, vgl. andere Lehrveranstaltungen).

Aufgrund der Vernetzung der Systeme und der wirtschaftlichen Nutzung der Netze haben Verschlüsselungsverfahren eine hohe Bedeutung erlangt (Beachte auch die politische Diskussion und Stellungnahmen dazu, vgl. Informatik & Gesellschaft).



## 10.4.5 Beispiel: Benutzer-Authentifikation

---



- Authentifikation bedeutet, Gewissheit zu erlangen über eine vorgegebene Identität eines Subjekts.
- Benutzer-Authentifikation beinhaltet, dass das Betriebssystem eine vorgegebene Identität eines Benutzers überprüfen kann, wenn er Zugang zum System verlangt.
- Prinzipiell sollte sich ein Benutzer auch über die Identität eines Rechensystems Gewissheit verschaffen können (gegenseitige Authentifikation).

- Passworte.
  - Regeln beachten für brauchbare Passworte.
  - Passworte nicht in unverschlüsselter Form speichern.
  - Regelmäßiges Ändern von Passworten, z.T. über Alterungsmechanismus erzwungen.
  - Extreme Form: Einmal-Passworte gemäß einer Liste von vereinbarten Passworten (Beispiel: TANs).
- Herausforderung / Antwort - Spiel (Challenge / Response).
- Physische/Biometrische Identifikation.
  - Chipkarte mit Password (PIN) oder SmartCards.
  - Bestimmung schwer fälschbarer physischer Merkmale:
    - ➔ Fingerabdrücke.
    - ➔ Sprachprobe.
    - ➔ Iris-Abbildung.
    - ➔ Unterschriftenanalyse.
    - ➔ Blutprobe (Akzeptanzproblem).



- Örtliche und zeitliche Einschränkungen
  - Benutzern werden Terminalleitungen und mögliche Zeiten zugeordnet, über die und während der ein Benutzer Zugang zum System erhalten kann.
- Rückruf
  - bei Einwahl über Modem mit vorab fest vereinbarter Telefonnummer.
- Fallen
  - Aufstellen von "Fallen" im System, mit denen Eindringlinge auf frischer Tat ertappt werden können.

- Mehrbenutzersystem
- Nur eingerichtete Benutzer können System nutzen
- Einrichten von Benutzern ist dem privilegierten Benutzer Systemadministrator (Super-User) vorbehalten.
  - In neueren Versionen (z.B. System V.R4) wurden spezielle Administratoren mit abgestuften Rechten eingeführt (vgl. auch Windows NT).
  - Einrichten geschieht durch spezielle Werkzeuge (z.B. `adduser`, `SAM`) oder durch Editieren der Datei `/etc/passwd`, in der alle (lokalen) Benutzer geführt werden.
- Benutzer
  - besitzt Namen (account)
  - systeminterne Kennung `uid` (Benutzernummer, User Identification), die Basis für Rechtsüberprüfungen ist. Der Super-User besitzt die `uid 0`.
  - ist ein und mehreren Benutzergruppen zugeordnet, die in `/etc/group` geführt werden.

- Anmeldungsvorgang eines Benutzers (`login`)
  - verwendet Passwort-basierte Authentisierung.  
Passwörter in verschlüsselter Form in der Datei `/etc/passwd` (bzw. für Gruppen in `/etc/group`) gespeichert.
    - ➔ Problem: Datei muss „world readable“ sein
    - ➔ Jeder kann sie kopieren und in aller Ruhe die enthaltenen Passwörter (per „brute force“) entschlüsseln
    - ➔ Heute in der Regel: Verschlüsselte Passwörter in „shadow“-Passwortdatei ausgelagert, nur für root zugreifbar
- Nach erfolgreicher Authentisierung
  - besitzt der Benutzer die systeminterne Identität (`uid`, `gid`) entsprechend den Festlegungen in `/etc/passwd`
  - und die dort ebenfalls festgelegte initiale Anwendung (i.d.R. eine Shell) wird gestartet.



Zur Erinnerung:

- Schutzmechanismen beinhalten technische Verfahren in Rechensystemen, die dem Schutz von Informationen dienen, um Sicherheit zu erreichen.

## Gliederung

1. Schutzzumgebungen (Protection Domains)
2. Zugriffskontrolllisten
3. Capabilities
4. Standardisierte Sicherheitskriterien

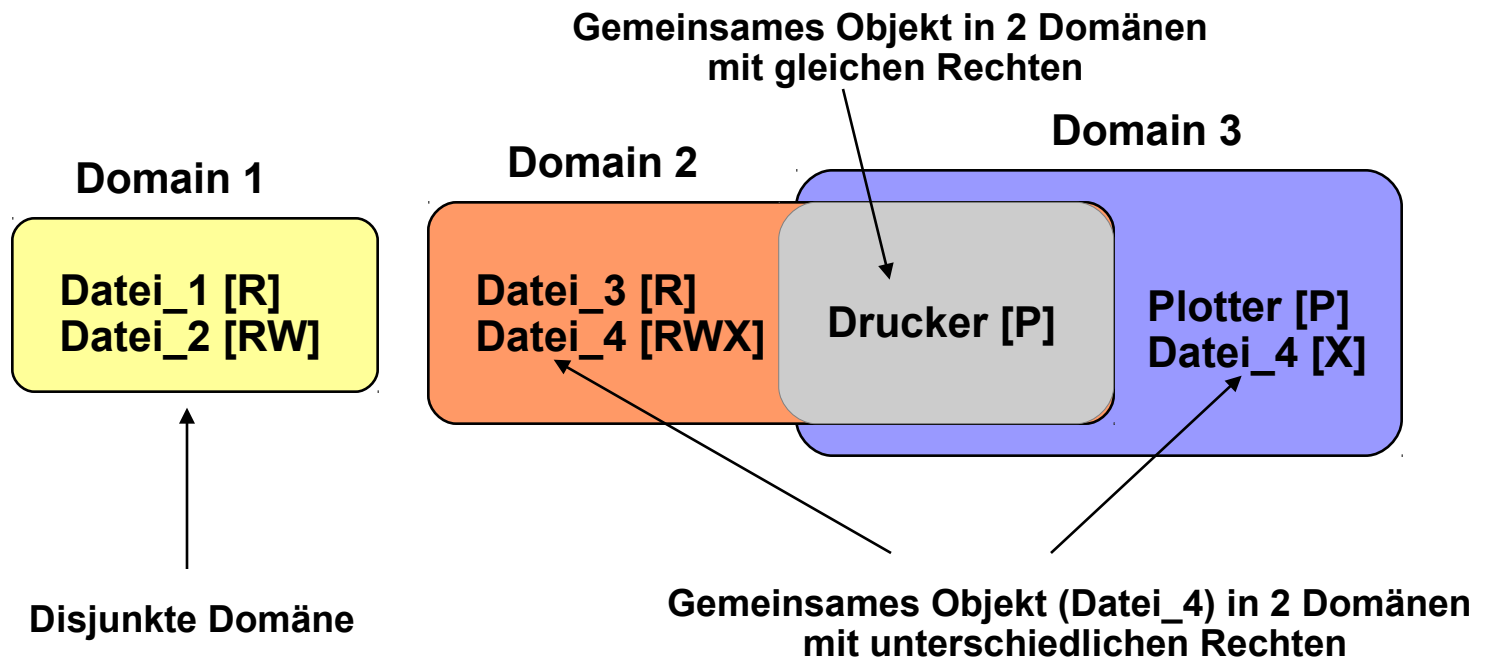
## 10.5.1 Schutzumgebungen (Protection Domains)



### Def

- Objekte
  - sind die Einheiten, die geschützt werden sollen, z.B. Dateien.
  - besitzt einen eindeutigen Namen, über den auf das Objekt zugegriffen wird
  - Menge von Operationen, die auf dem Objekt ausgeführt werden können (Sicht von abstrakten Datentypen).
- Subjekte
  - sind die aktiven Einheiten, die auf Objekte zugreifen wollen.
  - Beispiele: Benutzer, Prozesse.
- Recht
  - beinhaltet die Erlaubnis zur Ausführung einer Operation auf einem Objekt.
- Schutzumgebung (Protection Domain, Domäne)
  - Menge von Paaren (Objekt, Recht).
- Einem Subjekt ist in jedem Augenblick eine Schutzumgebung zugeordnet, die die Menge der zugreifbaren Objekte und der auf ihnen ausführbaren Operationen beschreibt.
- Ein Subjekt kann prinzipiell seine Schutzumgebung wechseln.

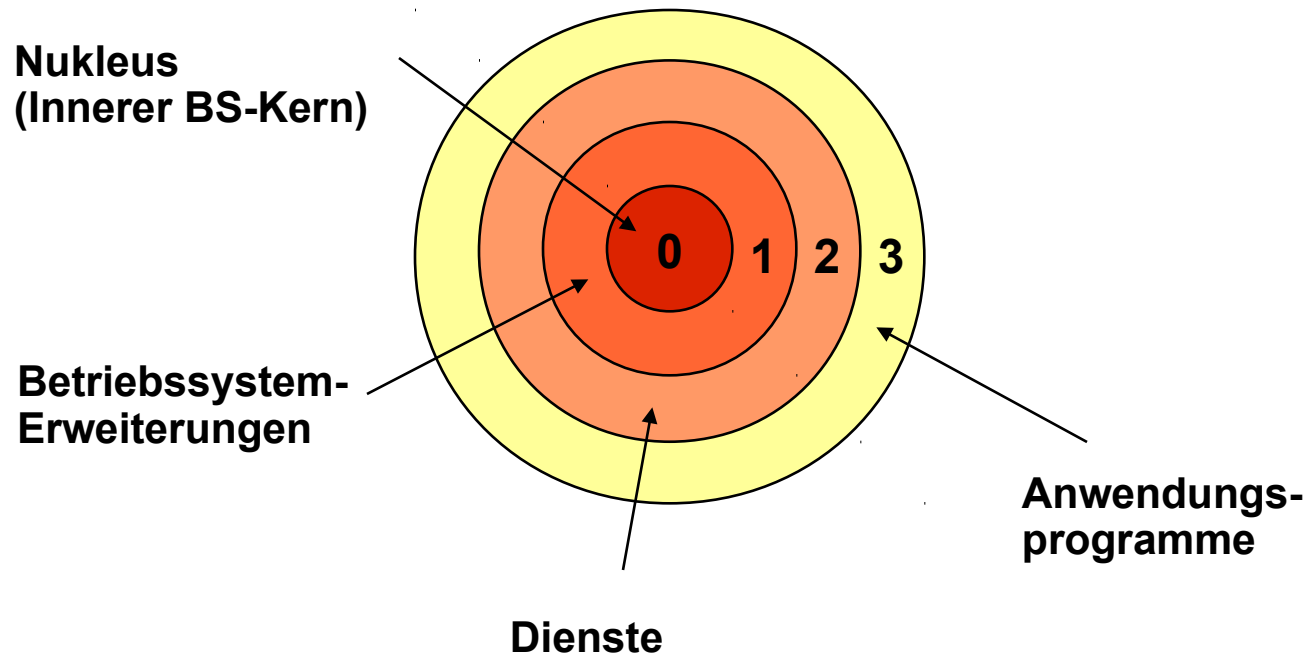
- Objekte:
  - Dateien mit den Operationen Read, Write, eXecute.
  - Drucker, Plotter mit der Operation Print.
- Domänen:





- Objekte: Dateien einschl. Spezialdateien (Geräte).
- Operationen: Read, Write, Execute.
- Subjekte: UNIX-Prozesse.
- Jedem Prozess zugeordnet:
  - reale Benutzer- und Gruppennummer (uid, gid):
    - ➔ Prozess ist "Stellvertreter" des realen Benutzers uid und der Gruppe gid im System.
  - effektive Benutzer- und Gruppennummer (euid, egid):
    - ➔ zur Überprüfung der Dateizugriffsrechte verwendet (s.u.).
  - Zugriffsumgebung:
    - ➔ Menge aller Dateien, deren Eigentümer oder Gruppe mit der effektiven Benutzer-/Gruppennummer des Prozesses übereinstimmt.
    - ➔ Rechte bzgl. Dateien sind durch Festlegungen im Inode gegeben.

- Verallgemeinerung der Zweiteilung Benutzermodus / Kernmodus.
- Ursprung: MULTICS protection rings, z.B. 16 Ringe.
- Beispiel: Intel '386 erlaubt 4 Ringe (Privilege Levels).



- Eine Schutzmatrix (oder Zugriffsmatrix) ist eine abstrakte Beschreibung einer Rechtssituation des Gesamtsystems.
- dient der Überprüfung der Rechtmäßigkeit eines beabsichtigten Zugriffs.

		Objekte					
		Datei_1	Datei_2	Datei_3	Datei_4	Drucker	Plotter
D o m ä n e n	Domäne 1	Read	Read Write				
	Domäne 2			Read Write	Read Write Execute	Print	
	Domäne 3				Execute	Print	Print

**Menge der Rechte in einer Domäne  
an einem Objekt**

- Modellierung der Rechtmäßigkeit von Domänenwechseln durch Auffassen von Domänen als Objekte mit der Operation "enter".

	Objekte						Domäne		
	Datei_1	Datei_2	Datei_3	Datei_4	Drucker	Plotter	1	2	3
Domäne 1	Read	Read Write						Enter	Enter
Domäne 2			Read Write	Read Write Execute	Print				
Domäne 3				Execute	Print	Print			

## 10.5.2 Zugriffskontrolllisten (ACL)

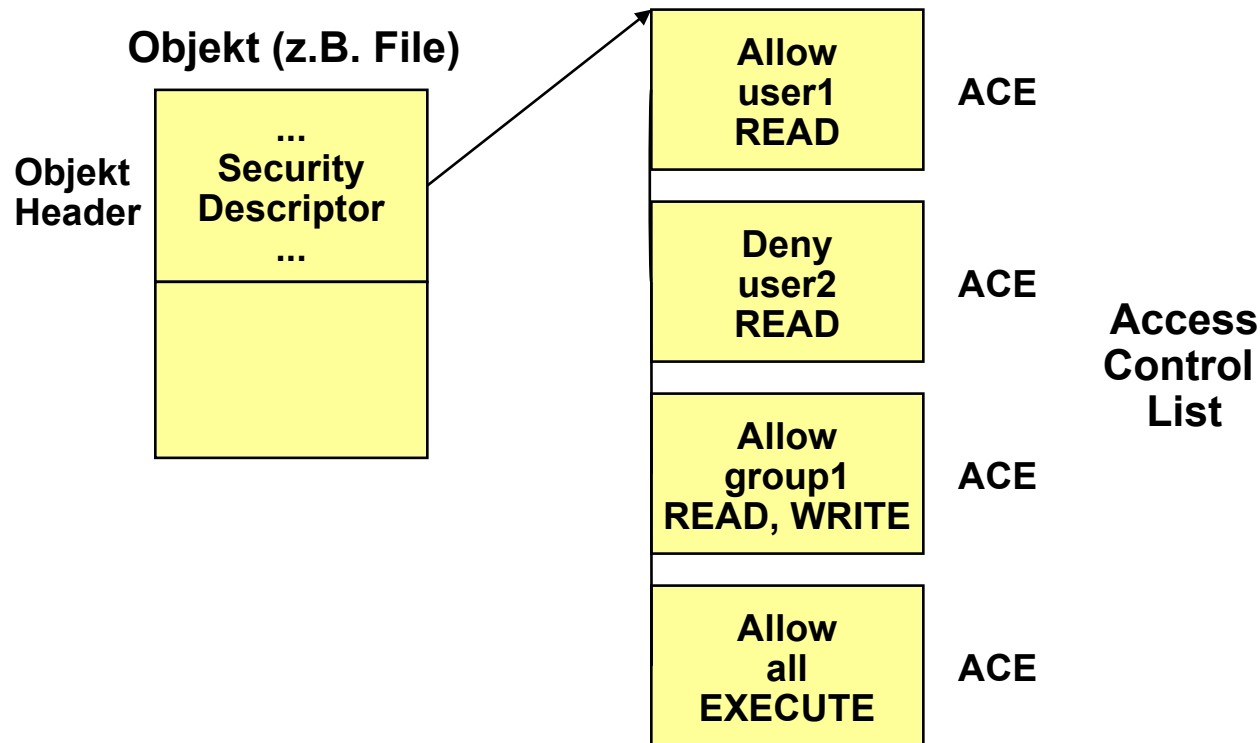
---



- Information der abstrakten Schutzmatrix wird nicht als Matrix gespeichert (groß, dünn besetzt).  
Hier betrachtet: *spaltenweise* Speicherung.
- Zugriffskontrollliste oder Access Control List (ACL)
  - Jedem Objekt zugeordnet
  - enthält alle Domänen, die auf dieses Objekt zugreifen dürfen, zusammen mit den jeweiligen Rechten.
- Bei Zugriff auf ein Objekt
  - Auswertung der Schutzumgebung
  - Entscheidung über die Zulässigkeit des beabsichtigten Zugriffs.
- Eigentümer eines Objekts kann die Zugriffskontrollliste jederzeit ändern
  - Rechteentzug (*Revocation*) einfach
  - Für bereits "geöffnete" Objekte kann der Rückruf eines Rechts schwierig oder unmöglich sein.

- Sehr einfache Form von Zugriffskontrolllisten für Dateien (und Geräte usw.)
- Im Inode der Datei geführt
- Jeweils 3 Rechte-Bits (rwx für Lesen, Schreiben, Ausführen)
  - für den Eigentümer der Datei (Owner),
  - die Gruppe des Eigentümers (Group)
  - für alle andere Benutzer (Others)
- Als Zusatzprodukte sind allgemeine ACL-basierte Dateisysteme verfügbar.
  - Diese werden zur Erreichung einer C2-Sicherheitsstufe benötigt, vgl. 10.5.4.

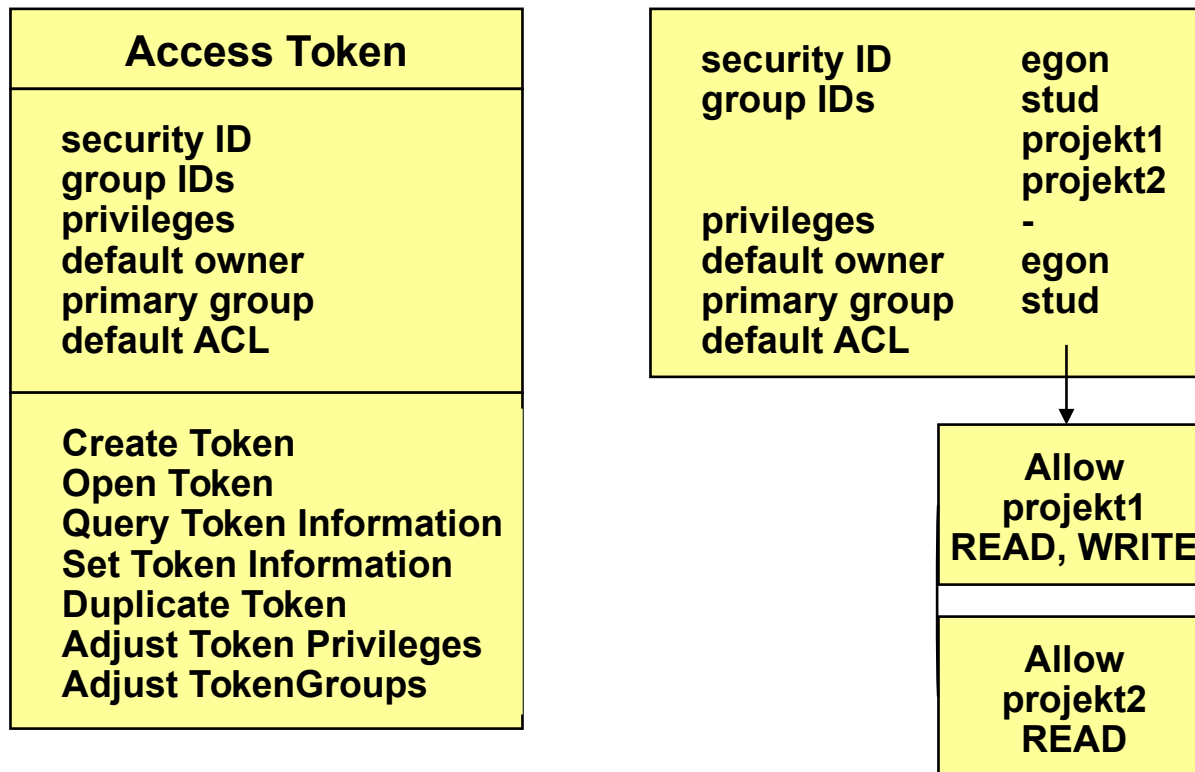
- Windows NT verwaltet für jedes Systemobjekt (z.B. File, Thread, Event) eine Zugriffskontrollliste (ACL), die Bestandteil eines Security-Deskriptors des Objekts ist. Die Einträge der ACL werden Access Control Entries (ACE) genannt.



# Beispiel: Windows NT (2)



- Windows NT erzeugt nach erfolgreicher Authentifizierung eines Benutzers ein Objekt vom Typ Access Token, das jedem Prozess des Benutzers fest zugeordnet wird.

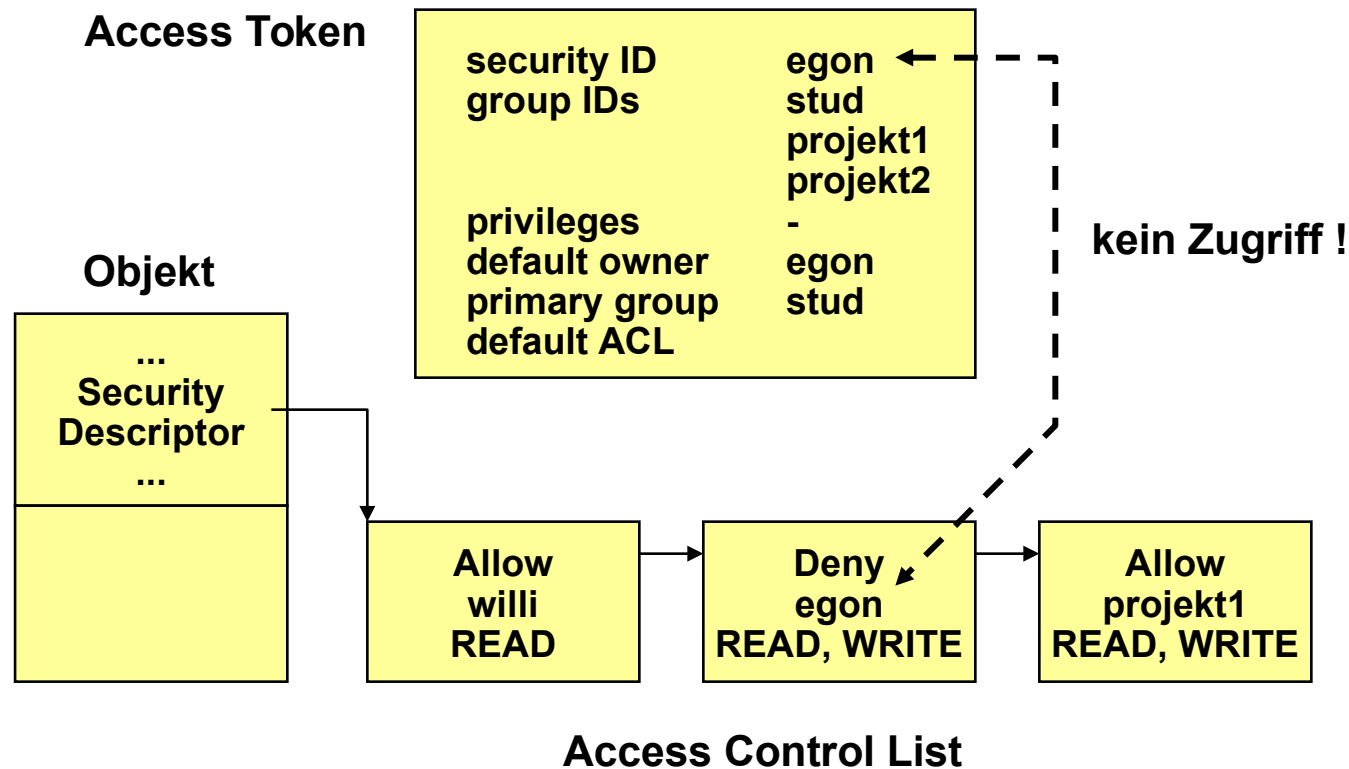




# Beispiel: Windows NT (3)



- Windows NT überprüft beim Öffnen eines Objects die Access Control Liste des Objekts mit der Information des Access Token des zugreifenden Prozesses.



## 10.5.3 Capabilities



- Hier: *zeilenweise* Speicherung der Schutzmatrix.
- Jedem Prozess wird Capability-Liste (C-Liste) zugeordnet:
  - jedes Element heißt Capability  
(Dennis, van Horn 1966; Fabry 1974)
  - Capability enthält Typfeld, Rechtefeld und Objektfeld mit Referenz auf das entsprechende Objekt
  - Liste benennt damit alle Objekte mit jeweiligen Zugriffsrechten.
- Capabilities werden i.d.R. durch ihre relative Position in der Capability-Liste identifiziert (analog UNIX File Deskriptoren).
- Beispiel einer typischen Capability-Liste:

	Typ	Rechte	Objekt
0	Datei	R W -	Zeiger auf Datei_3
1	Datei	R W X	Zeiger auf Datei_4
2	Drucker	P	Zeiger auf Drucker1

- Alternativen zur Sicherstellung der Unfälschbarkeit von Capabilities im Arbeitsspeicher:
  - Kennzeichnende Lösung (Tagged Memory): Jedes Speicherwort enthält Zusatzbit, das das Wort als Capability kennzeichnet. Tag-Bit kann nur im Kernmodus vom Betriebssystem geändert werden.
  - Aufteilende Lösung: Capabilities und Daten werden getrennt aufgehoben, C-Listen werden nur durch das Betriebssystem manipuliert. Ein Prozess referiert eine Capability durch Angabe des Index in der C-Liste.

Beispiele: Betriebssystem Hydra (Wulf, CMU);  
Prozessor Intel iAPX 432.
  - Verschlüsselung: Capabilities werden im Benutzeradressraum aufbewahrt, aber mit einem für Benutzer geheimen Schlüssel verschlüsselt. Unversehrtheit kann vom Betriebssystem mit grosser Sicherheit überprüft werden.

Beispiel: Betriebssystem Amoeba (Tanenbaum).

## 10.5.4 Standardisierte Sicherheitskriterien

---



- Klassifizierung für sogenannte Sichere Systeme der Informationstechnik basierend auf Kriterienkatalogen.
- Kriterienkatologe finden häufig bei der Beschaffung "Sicherer IT-Systeme" Anwendung.
- Ursprung US DoD "Orange Book"
  - TCSEC: Trusted Computer Systems Evaluation Criteria, 1983
- Deutschland:
  - Zuständigkeit heute:  
Bundesamt für Sicherheit in der Informationstechnik (BSI)
  - ehemals Zentralstelle für Sicherheit in der Informationstechnik (ZSI)
  - Deutsche IT-Sicherheitskriterien (Grünbuch), 1989
  - Konzept der Trennung von Funktionalität und Prüftiefe (Qualitätsstufe)



- Europa:
  - Information Technology Security Evaluation Criteria - ITSEC, 1998
  - Common Criteria zur Bewertung (Gegenseitige Anerkennung von Prüfsertifikaten), 1998
  - Common Criteria Version 2.1 als weltweiter Standard ISO/IEC 15408
    - ➔ Teil 1: Introduction and General Model
    - ➔ Teil 2: Security Functional Requirements
    - ➔ Teil 3: Security Assurance Requirements
  - Unterscheidung Funktionalität und Vertrauenswürdigkeit
  - Vordefinierte Beispielklassen (Funktionalitätsklassen)
  - Vertrauenswürdigkeit unterscheidet zwischen Korrektheit und Wirksamkeit (Stärke *niedrig*, *mittel* und *hoch*)
  - Bewertung des Vertrauens in die Korrektheit durch sechs hierarchische Evaluationsstufen E1 (niedrig) bis E6 definiert.



- Grobe Zuordnung:

<i>ITSEC</i>	<i>TCSEC</i>
E0	D
F-C1, E1	C1
F-C2, E2	C2
F-B1, E3	B1
F-B2, E4	B2
F-B3, E5	B3
F-B3, E6	A1



- Für Klasse C2 (Orange Book) sind insbesondere mit technischen Mitteln innerhalb des Betriebssystems zu erfüllen:
  - Sichere Login-Prozedur zur Authentifizierung
  - Discretionary Access Control
    - ➔ Eigentümer eines Objekts hat freie Entscheidung, welche Rechte er welchem Subjekt oder Gruppe von Subjekten an dem Objekt zubilligt
    - ➔ wie bisher besprochen, z.B. auf der Basis eines ACL-Mechanismus durchgesetzt.
  - Auditing
    - ➔ Möglichkeit zur Erkennung und Protokollierung wichtiger sicherheitsrelevanter Ereignisse im System zusammen mit der Identität des verursachenden Benutzers.
  - Initialisierung aller Speicherbereiche
    - ➔ niemand kann Informationen eines früheren Benutzers einer Datenstruktur in Erfahrung bringen.



- Die Klasse B2 (Orange Book) ist i.w. für Anwendungen im militärischen Bereich sinnvoll:
  - Mandatory Access Control
  - Jeder Benutzer und jedes Objekt hat Sicherheitseinstufung (Clearance Level):  
z.B. Vertraulich, Geheim, Streng Geheim.
  - Information darf nicht zu Benutzern mit geringerer Sicherheitstufe fließen.



Was haben wir in Kap. 9 gemacht?

- Konzepte von Dateisystemen.
- Verwalten von Mengen von Dateien und Verzeichnissen.
- Sicherheit und Schutz.
- In 10.1 und 10.2 zunächst eine äußere Benutzersicht
  - Benennung von Dateien
  - Dateistrukturen und -Typen
  - Zugriffsarten
  - Dateiattribute
  - Moderne Dateisysteme unterstützen hierarchische Strukturierung beliebiger Tiefe.

- In 10.3 Dateisysteme aus der Sicht eines Implementierers
  - Zuordnung von Plattenblöcken
  - Freispeicherverwaltung
  - Realisierung von Verzeichnissen
  - Repräsentierung von Dateien im Hauptspeicher
  - Zuverlässigkeitsaspekte
  - Performance-steigernde Maßnahmen
- In 10.4 und 10.5: allgemeine Sicherheitsaspekte sowie Schutzmechanismen in Betriebssystemen
  - Authentifikation von Benutzern
  - Zugriffskontrollisten und Capabilities als Realisierungen einer abstrakten Schutzmatrix.