Kapitel 6: Dateisysteme

6: Dateisysteme

Agenda

- Wie findet man Art, Eigentümer und Gruppe einer Datei heraus?
- Wie funktioniert der Boot-Vorgang unter Linux (BIOS)?
- Wie werden unter Linux Dateien gespeichert?
- Wie werden unter Linux Verzeichnisse und Verknüpfungen realisiert?
- Wie können unter Linux Verzeichnisse angelegt, gelöscht und gelesen werden?

6.1: Dateiattribute

```
struct stat {
  dev_t          st_dev;      /* Geraete-ID des Datentraegers */
  ino_t          st_ino;      /* INode */
  mode_t         st_mode;     /* Dateityp und -modus (rwxrwxrwx) */
  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;     /* Geraete-ID (falls Geraet) */
  off_t          st_size;     /* Groesse in Bytes */
  blksize_t      st_blksize;  /* Bevorzugte Blockgroesse (Performance) */
  blkcnt_t       st_blocks;   /* Anzahl der zugewiesenen 512B-Bloecke */

  struct timespec st_atim;  /* Zeit des letzten Zugriffs */
  struct timespec st_mtim;  /* Zeit der letzten Veraenderung*/
  struct timespec st_ctim;  /* Zeit der letzten Statusaenderung*/
};
```

Quelle: Manpage zu `stat` (Abschnitt 2)

Achtung: Die Reihenfolge der Einträge kann varieren.

Dateiattribute lesen

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

int stat (const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

- Bei Erfolg wird `0`, ansonsten `-1`, zurückgegeben
- **`stat()`**: Liefert Attribute zu der in `pathname` angegebenen Datei
- **`stat()`**: Liefert die Attribute zur Datei mit dem Filedeskriptor `fd`
- **`lstat()`**: Falls es sich bei `pathname` um einen symbolischen Link (Verknüpfung) handelt wird diesem, im Gegensatz zu `stat()`, **nicht** gefolgt

Dateiattribute: Beispiel I

```
1  #include <sys/stat.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5  #include <pwd.h>
6
7  int main(int args, char *argv[]) {
8    struct stat sb;
9
10   for(int i=1; i <args; i++) {
11     if (lstat(argv[i], &sb)) perror(argv[i]);
12     else {
13       printf("Typ␣and␣Mode:␣␣␣␣␣␣␣␣␣␣␣%lo␣(octal)\n",
14              (unsigned long) sb.st_mode);
15       printf("Link␣count:␣␣␣␣␣␣␣␣␣␣␣␣␣%ld\n", (long) sb.
     st_nlink);
16       printf("Owner:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n", getpwuid(sb.
     st_uid)->pw_name);
17
18       printf("Preferred␣block␣size:␣␣␣%ld␣bytes\n",
19              (long) sb.st_blksize);
20       printf("File␣size:␣␣␣␣␣␣␣␣␣␣␣␣␣%lld␣bytes\n",
```

Dateiattribute: Beispiel II

```
21                (long long) sb.st_size);
22        printf("Blocks␣allocated:␣␣␣␣␣␣␣%lld\n",
23                (long long) sb.st_blocks);
24
25        printf("Last␣status␣change:␣␣␣␣␣%s", ctime(&sb.
      st_ctime));
26        printf("Last␣file␣access:␣␣␣␣␣␣␣%s", ctime(&sb.
      st_atime));
27        printf("Last␣file␣modification:␣%s", ctime(&sb.
      st_mtime));
28        }
29    }
30    exit(EXIT_SUCCESS);
31    }
```

Dateiarten

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

int S_ISREG  (mode_t m) /* Regulaere Datei */
int S_ISDIR  (mode_t m) /* Verzeichnis */
int S_ISLNK  (mode_t m) /* Symbolischer Link /*

int S_ISCHR  (mode_t m) /* Zeichenorientiertes Geraet */
int S_ISBLK  (mode_t m) /* Blockorientiertes Geraet */

int S_ISFIFO (mode_t m) /* Pipe oder FIFO */
int S_ISSOCK (mode_t m) /* Socket */
```

- Bei Erfolg wird **0** zurückgegeben, ansonsten **−1**

Dateiarten: Beispiel

```
1  #include <sys/stat.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int args, char *argv[]) {
6    struct stat sb;
7
8    for(int i=1; i <args; i++)
9      if (lstat(argv[i], &sb)) perror(argv[i]);
10     else {
11       printf("%s:␣",argv[i]);
12       switch (sb.st_mode & S_IFMT) {
13       case S_IFBLK:  puts("blockorientiertes␣Geraet");   break;
14       case S_IFCHR:  puts("zeichenorientiertes␣Geraet"); break;
15       case S_IFDIR:  puts("Verzeichnis");                break;
16       case S_IFIFO:  puts("FIFO/Pipe");                  break;
17       case S_IFLNK:  puts("symbolischer␣Link");          break;
18       case S_IFREG:  puts("regulaere␣Datei");            break;
19       case S_IFSOCK: puts("Socket");                     break;
20       default:       puts("unbekannt?");                 break;
21       }
22     }
23   exit(EXIT_SUCCESS);
24 }
```

Gerätedateien

Die Geräte-ID einer eine Gerätedatei besteht aus zwei Teilen

1. Major Device Number: Legt den Gerätetreiber fest
2. Minor Device Number: Interpretation hängt vom Gerätetreiber ab

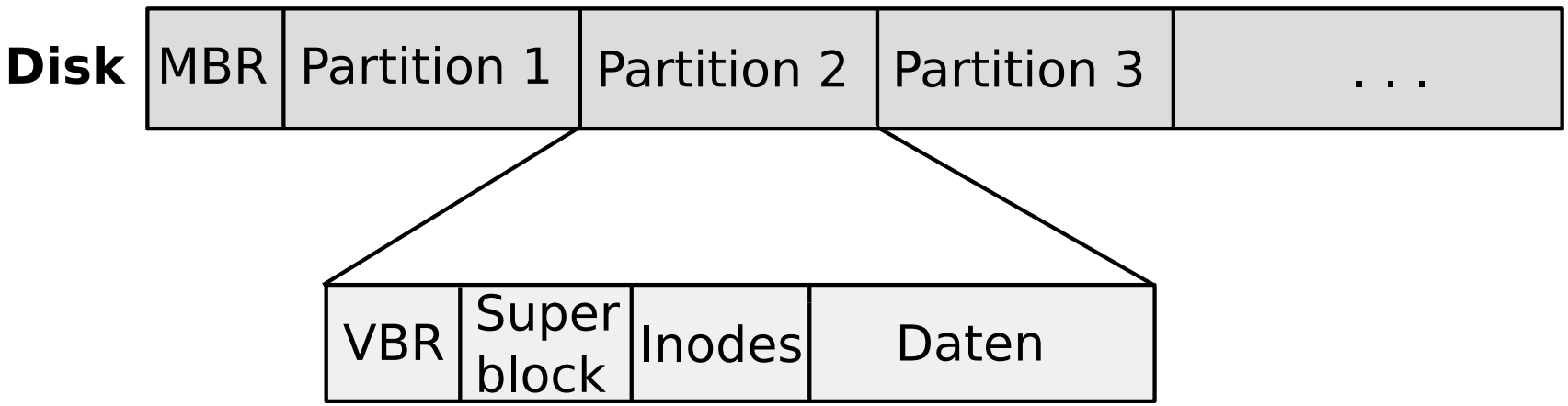
Beispiel: Erstellung und Nutzung einer Gerätedatei

```
$ ls -lah /dev/zero
crw-rw-rw- 1 root root 1, 5 Sep  7 10:13 /dev/zero
$ dd bs=100 count=1 if=/dev/zero of=foo.foo > /dev/null
$ sudo mknod beuth c 1 5
$file beuth
beuth: character special (1/5)
$ dd bs=100 count=1 if=/dev/beuth of=foo.bar > /dev/null
$ diff foo.foo foo.bar; echo $?
0
```

Spezielle Gerätedateien

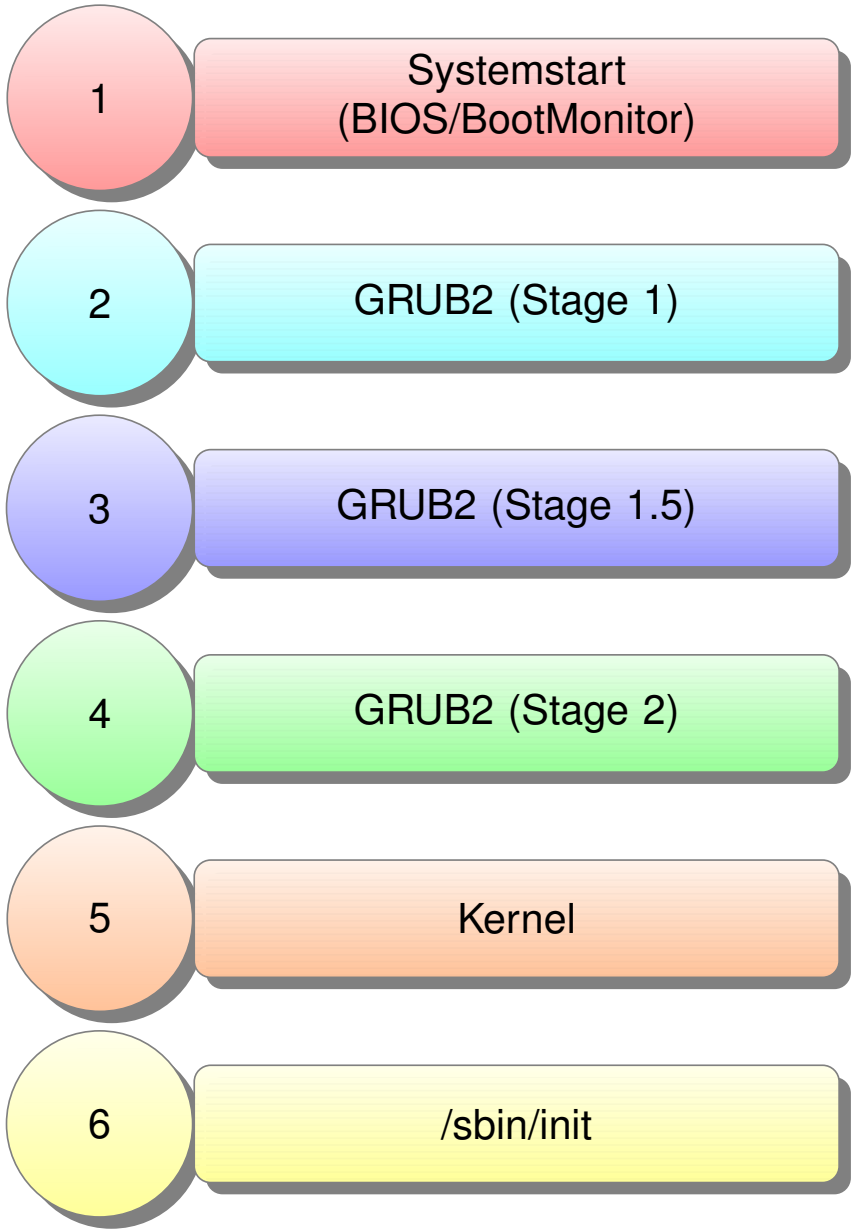
- `/dev/null`: Datengrab beim Aufruf von `read()` wird ein `EOF` gelesen.
- `/dev/zero`: Datengrab welches beim Lesen eine Sequenz von `0x00` Bytes liefert.
- `/dev/full`: Bei einem Aufruf von write `write()` kommt es zu dem Fehler `ENOSPC`. Beim Aufruf von `read()` werden `0x00` Bytes gelesen.
- `/dev/random`: Gibt eine Sequenz von Zufallszahlen zurück. Lesezugriff blockiert bei zu geringer Entropie.
- `/dev/urandom`: Gibt eine Sequenz von Zufallszahlen zurück. Lesezugriff blockiert nicht.

6.2: Partitionen, Filesysteme und Inodes



- MBR: Master Boot Record, VBR: Volume Boot Record
- Partition: Teil einer Disk (Beispiele: `C:`, `/dev/sda1`)
- Inodes: Metainformationen über gespeicherte Daten
- Inodes werden unter Windows FCBs (file control blocks) genannt
- Disks sind in logische 512-Byte-Blöcke eingeteilt
- Die tatsächliche Blockgröße ist ein Vielfaches von 512 Byte:
  - Normale HDD Block Größe: 4 KB
  - Normale SSD Block Größe: 512 KB

Linux-Bootvorgang (BIOS)

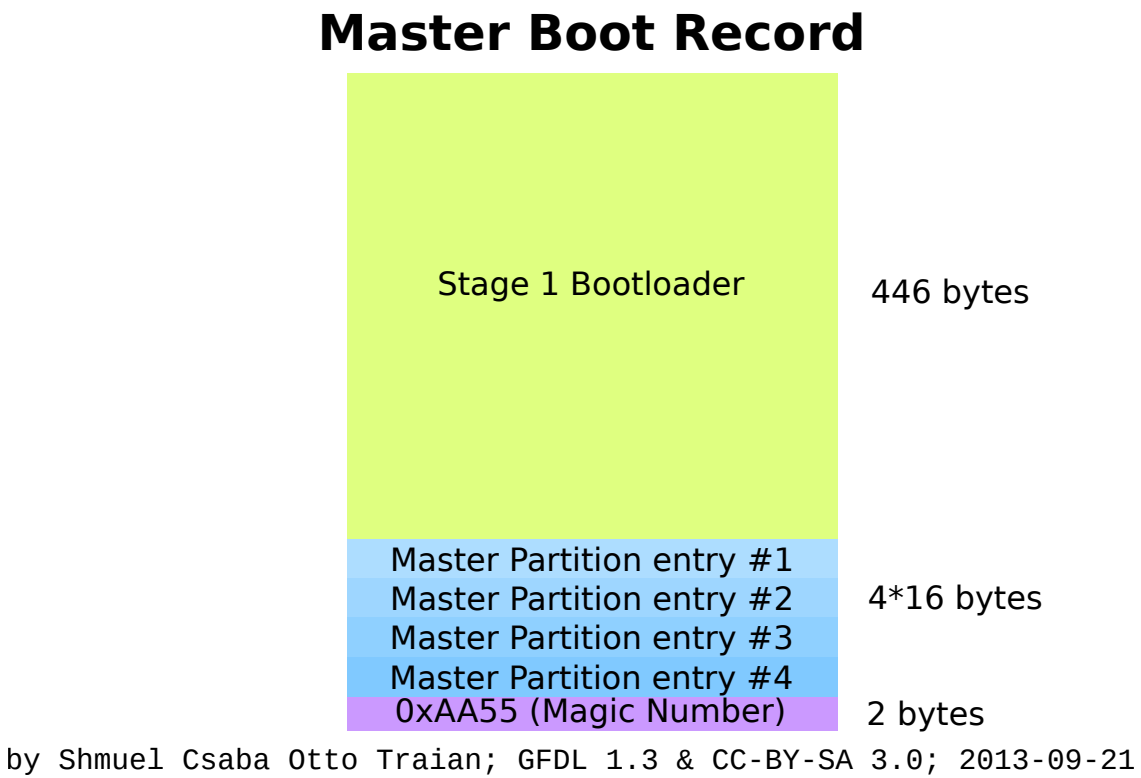


Linux-Bootvorgang: Systemstart

Systemstart einer i386 (IA-32) CPU mit BIOS.

- Die 64 Kilobyte des BIOS-EPROMs werden auf den physikalischen Speicherbereich [`0xFFFF_0000`, `0xFFFF_FFFF`] abgebildet
- Die Startroutine des BIOS befindet sich an der Speicheradresse `0xFFFF_FFF0`
- Der Befehlszeiger der CPU zeigt beim Einschalten der CPU auf `0xFFF_FFFF0`:
  - Segment Register `CS.BASE = 0xFFFF_0000`
  - Instruction Pointer `EIP = 0x0000_FFF0`
  - CS:EPI = CS.BASE + EIP: `CS.BASE = 0xFFFF_FFF0`
- Das BIOS versucht den MBR (erster 512-Byte-Block) eines Bootdevices zu laden

Linux-Bootvorgang: GRUB2 (Stage 1)

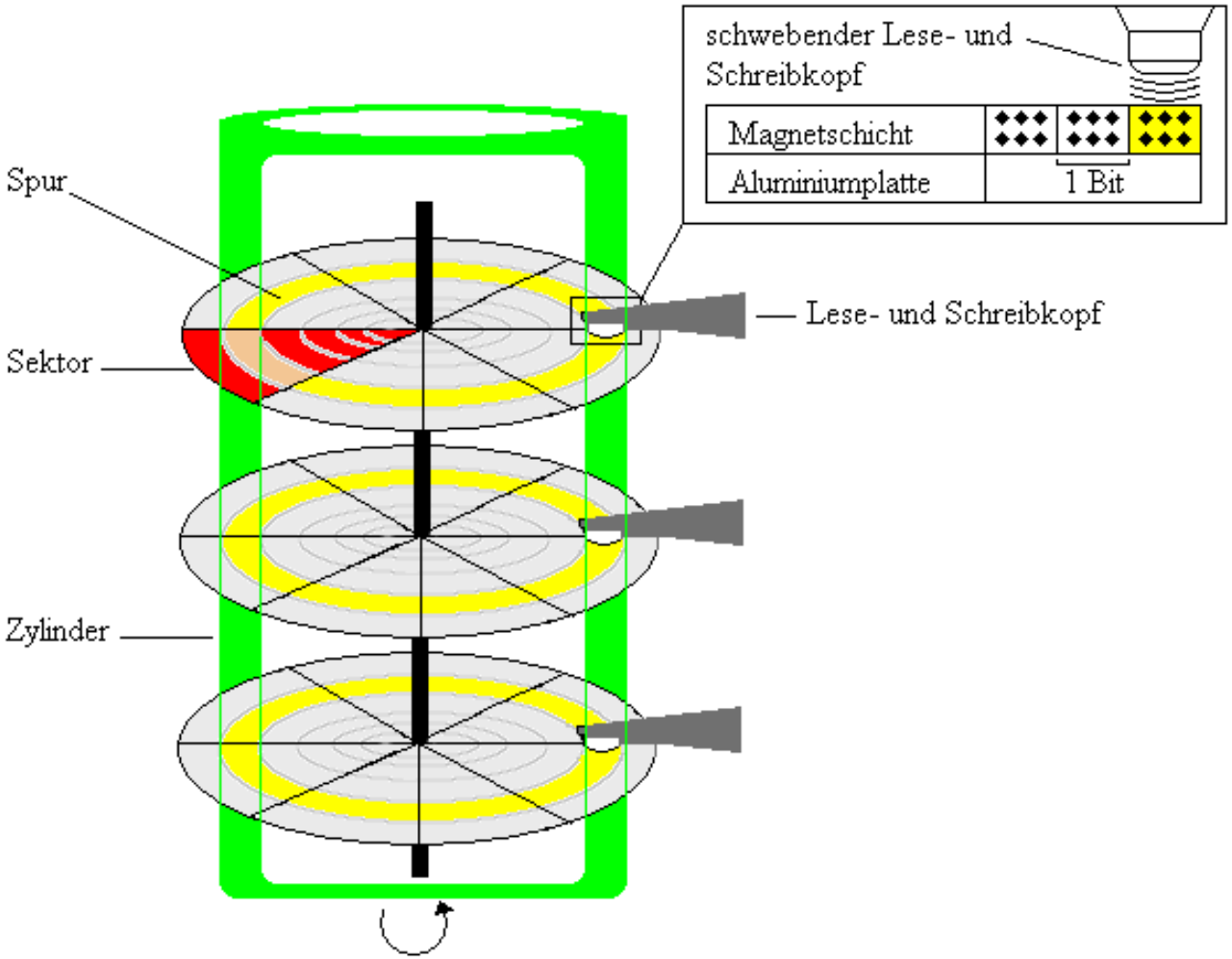


Aufgabe des *Stage 1 Bootloader* von GRUB2 (`boot.img`) ist es, die Dateisystemtreiber welche sich in *Stage 1.5* befinden, zu laden. Dafür enthält *Stage 1* einen Link (LBA-Wert, Logical Block Adressing) auf *Stage 1.5*

Bootvorgang: GRUB2 (Stage 1.5)

- Der 32 Kilobyte große Bootloader (`core.img`) befindet sich zwischen dem MBR und dem ersten Block der ersten Partition
- Aus historischen Gründen (Zylindergrenze) beginnt die erste Partition bei Sektor 63 (512-Byte-Block)
- Stage 1.5 enthält die benötigten Treiber um auf die Dateisysteme der einzelnen Partitionen zuzugreifen
- Stage 1.5 lädt den Stage 2 Bootloader welcher auf der Systempartition unter `/boot/grub/` zu finden ist

Festplattengeometrie

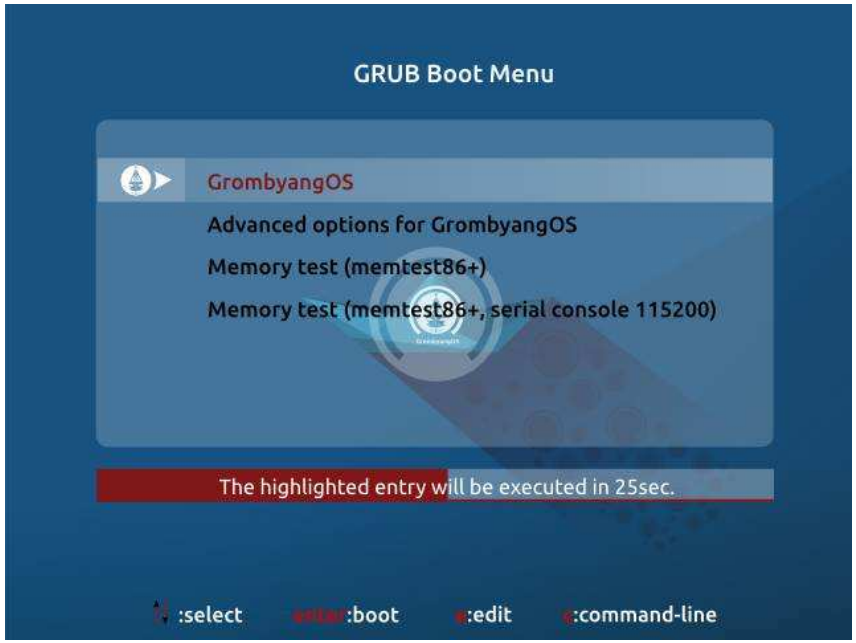


Quelle: Bagok (CC BY-SA 3.0)

Bootvorgang: GRUB2 (Stage 2)

- Aufgabe von Stage 2 ist das Laden des Kernels
- Stage 2 unterstützt auch das Laden einer optionalen RAM-Disk
- Stage 2 verfügt über ein Command-Line-Interface (CLI)
- Stage 2 kann auch BSD- oder MacOS-X-Kernels laden
- Stage 2 kann auch andere Bootloader laden
- Stage 2 unterstützt das dynamische Laden von Treibern
- Stage 2 hat ein graphisches Boot-Menü (Auswahl von Kernels)
- Der Zugriff auf Stage 2 kann durch ein Passwort geschützt werden

GRUB2 Bootmenü-Impressionen



Quelle: `https://wiki.mageia.org/en/File:Grub2_mga3.png`, `https://github.com/grOS-TEAM/grOS-Grub2-Themes`

Bootvorgang: `/sbin/init`

- Der Linux-Kernel ist komprimiert ((b)zImage)
- Der Header des Kernelimages enthält eine Startroutine:
  - Grundlegendes Hardwaresetup
  - Anlegen des Kernelstacks
  - Entpacken des Kernelimages
- Der Kernel wird nach seinem Entpacken durch Prozess 0 geladen
- (Die initale-RAM disk (`initrd`) wird als temporäres `root`-Dateisystem gemountet und Treiber (Module) werden nachgeladen)
- Der Kernel startet den ersten Userspace-Prozess (PID 1) `/sbin/init` (ggf. Symlink auf `systemd`)
- Der Init-Prozess initialisiert den Userspace

Der Inode

- Schreibweisen: I-Node, Inode, INode, i-node, inode
- Ein Inode (Index-Node) ist eine Datenstruktur (Struct) welche die Metadaten über eine bestimmte Datei enthält
- Der Superblock einer Partition enthält die Startposition der Inodes und die Länge eines einzelnen Inodes
- Die Inode-Nummer ist der Index des Inodes
- Aufbau;
  - Art der Datei (Datei, Verzeichnis, Gerätedatei, Socket, . . .)
  - Eigentümer (UID) und Gruppe (GID)
  - Zugriffsrechte
  - Zeitstempel (access time und modification time)
  - Zeit der letzten Statusänderung des Inodes
  - Dateigröße
  - Linkzähler (Anzahl der Hardlinks)
  - Verweise auf Blöcke die den Inhalt der Datei enthalten

File Allocation Table (FAT)

- Die FAT ist eine Tabelle, über die einerseits eine Zuordnung von Blöcken (Clustern) zu Dateien verwaltet wird.
- Im Zuge der technologischen Weiterentwicklung wurden über die Jahre verschiedene FAT-Versionen entwickelt. Unter anderem waren dies:
  - FAT12 (1980, 32 MB Partitionen)
  - FAT16 (1984, 4 GB Partitionen)
  - FAT32 (1996, 16 TB Partitionen, 4 GB Dateien)
- Eine Datei besteht aus einer verketteten Liste aus Blöcken.
- Der Inode addressiert den Anfang einer Liste.

FAT5 Datei-Allokation

Inode	Block
0	1
1	10
2	7
3	9
4	16

HDD

0	3	8	20	16	18	24	nil
1	0	9	23	17	frei	25	26
2	8	10	11	18	4	26	16
3	31	11	2	19	frei	27	28
4	nil	12	frei	20	21	28	nil
5	6	13	frei	21	5	29	frei
6	nil	14	frei	22	25	30	frei
7	22	15	frei	23	24	31	27

Linux-Dateisystem-Geschichte (1/2)

- 1992: Extended File System (ext)
  - Max. Länge des Dateinamens: 256 Byte
  - Indirektes Block-Mapping (Verweise auf Daten)
  - Max. Größe einer Datei/eines Dateisystems: 2 GB
  - Freie Inodes und Blöcke wurden in verketteten Liste gespeichert
  - Inode hatte nur einen Timestamp für `atime`, `mtime` und `ctime`
- 1993: Extended File System 2 (ext2):
  - Max. Größe einer Datei: 2TB
  - Max. Größe des Dateisystems: 32 TB
  - Geringere Fragmentierung durch Block Gruppen und Allocation Bitmaps
  - Individuelle Timestamps für `atime`, `mtime` und `ctime`

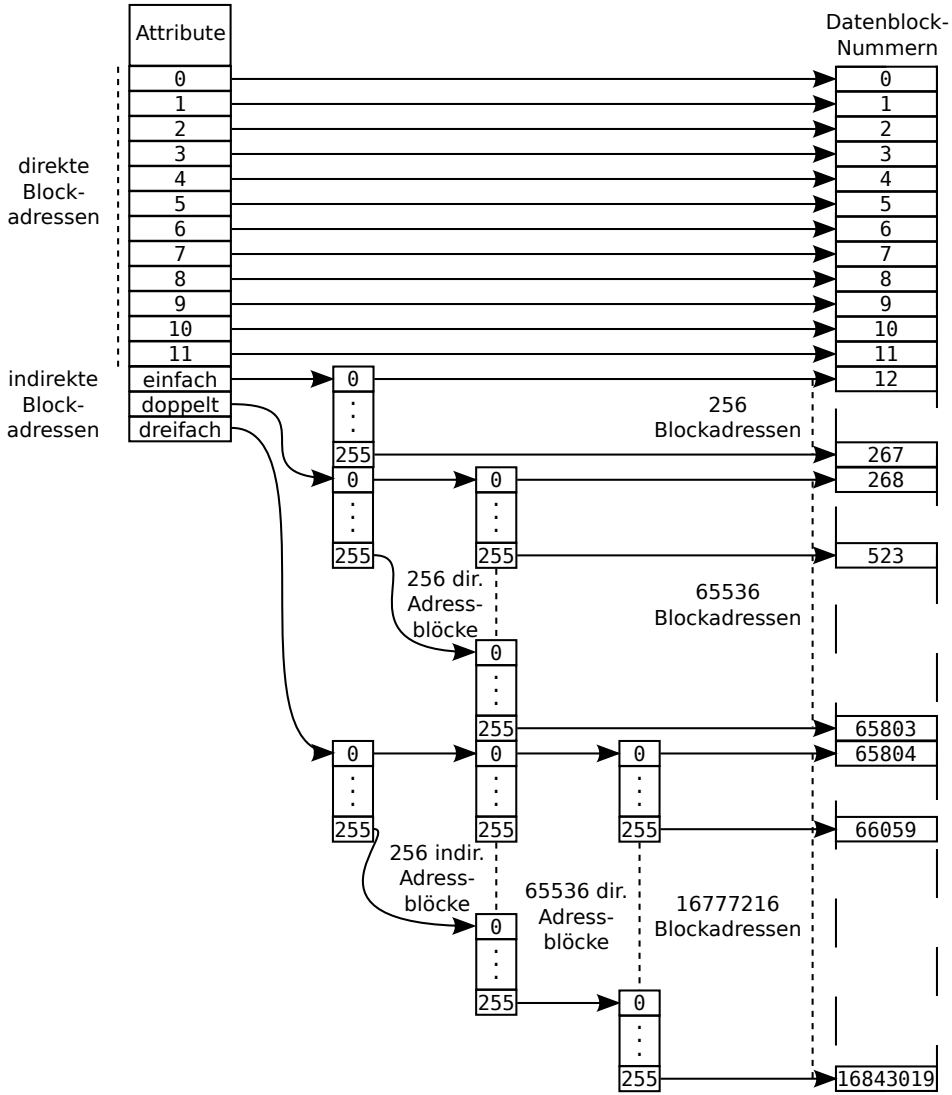
Historie der Linux Dateisystem Geschichte (2/2)

- 2001: Extended File System 3 (ext3)
  - Journaling (Transaktionslog) garantiert einen konsistenten Datenbestand nach Systemabstürzen/Stromausfällen
  - Hashed Binary Trees (H-Bäume) als Verzeichnisindices
  - Access Control Lists (ACL)
  - Abwärtskompatibel zu ext2 sowie vorwärtskompatibel
- 2006: Extended File System 4 (ext 4)
  - Extents (Verweise auf Daten)
  - Max. Größe einer Datei: 16 TB
  - Max. Größe des Dateisystems: 1 EB
  - Filesystemcheck ist ca. 10-mal schneller als bei ext3
  - Metadaten-Checksumme (CRC32)
  - Nanosekunden-Zeitstempel

Verweise auf Daten: ext3

- Die Inodegröße beträgt bei ext3 128 Byte
- Adressierung des Dateiinhaltes
  - Verweise auf die ersten 12 Datenblöcke der Datei.
  - Verweis auf 1. Indirektionsblock (256 Verweise auf Datenblöcke)
  - Verweis auf 2. Indirektionsblock ($256 \cdot 256$ Verweise auf Datenblöcke)
  - Verweis auf 3. Indirektionsblock ($256 \cdot 256 \cdot 256$ Verweise auf Datenblöcke)
- Durch die Adressierung können $12 + 256 + 256^2 + 256^3 = 16.843.020$ Datenblöcke adressiert werden
- Die max. Dateigröße ist von der Blockgröße abhängig; normalwerweise zwischen 16 GB und 4 Terrabyte

Verweis auf Daten in ext3



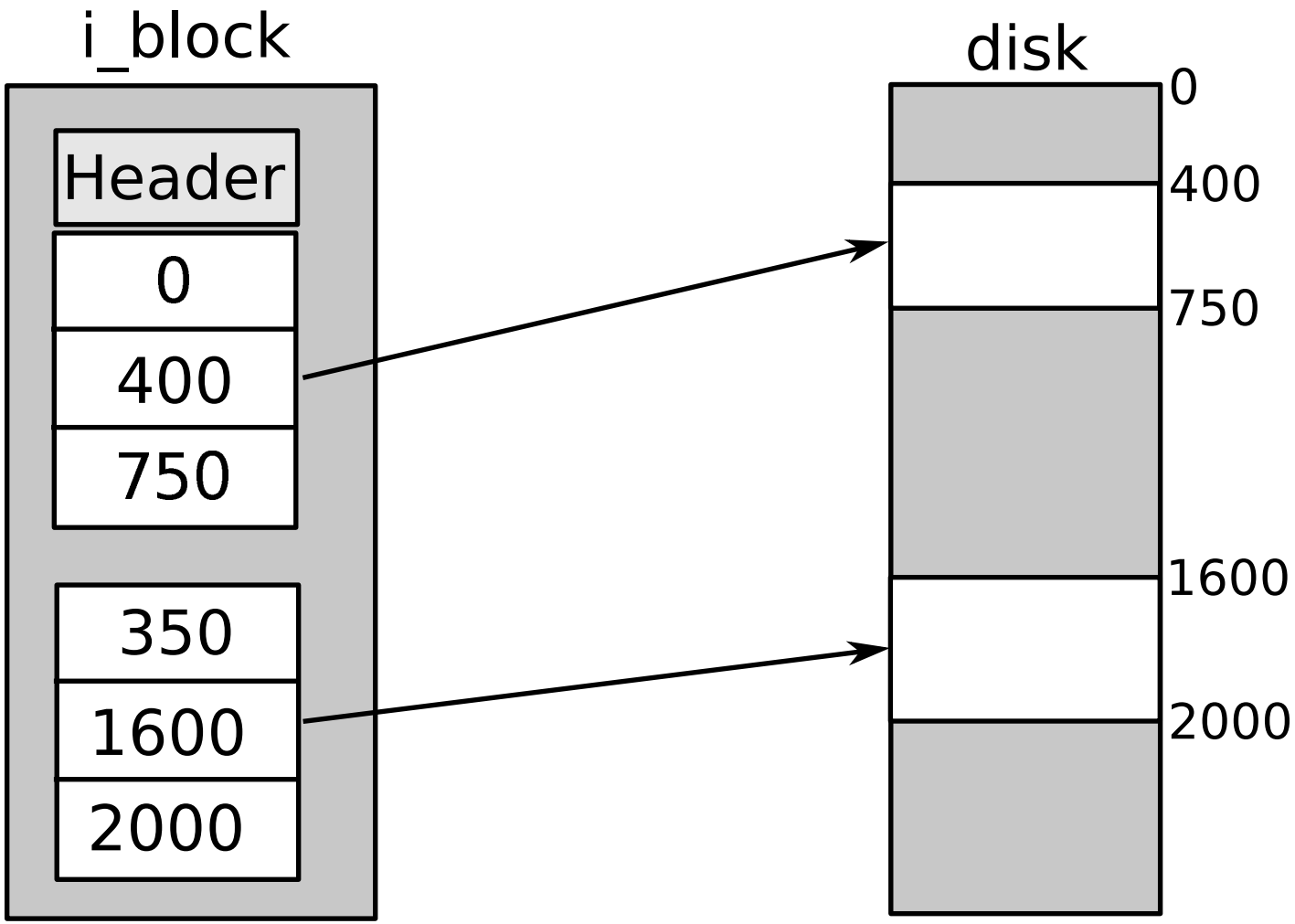
Quelle: Benutzer:Dfelsing (Wikipedia)

Extents in ext4

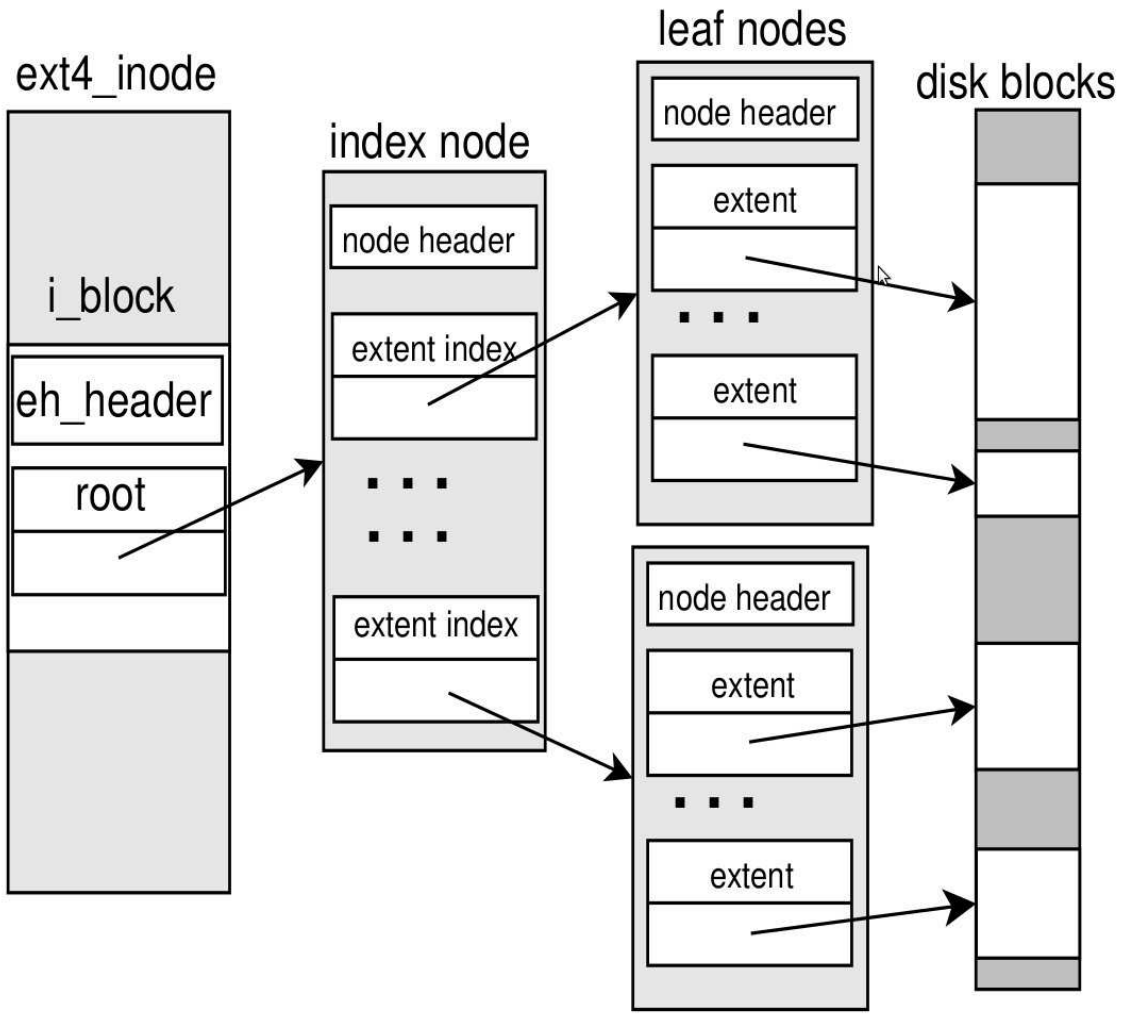
- Einen Zeiger pro Datenblock ist sehr ineffizient
- Extent: Datenstruktur die auf ein Bereich von bis zu $2^{16} - 1$ zusammenhängenden Blöcken zeigt
- Ein Inode kann bis zu drei direkte Extents verwalten
- Bei größeren Dateien wird ein B-Baum als Index verwendet

```
struct ext4_extent {
  __le32  ee_block;   /* Erster logischer Block */
  __le16  ee_len;     /* Anzahl zusammenhängender
    Bloecke */
  __le16  ee_start_hi; /* Die ersten 16 Bit des
    Startblockes */
  __le32  ee_start;    /* Die letzten 32 Bit des
    Startblockes */
};
```

Extents in ext4



B-Baum für Extents in ext4



Quelle: `http://cfile25.uf.tistory.com/image/197738494FD6AE7005DE79`

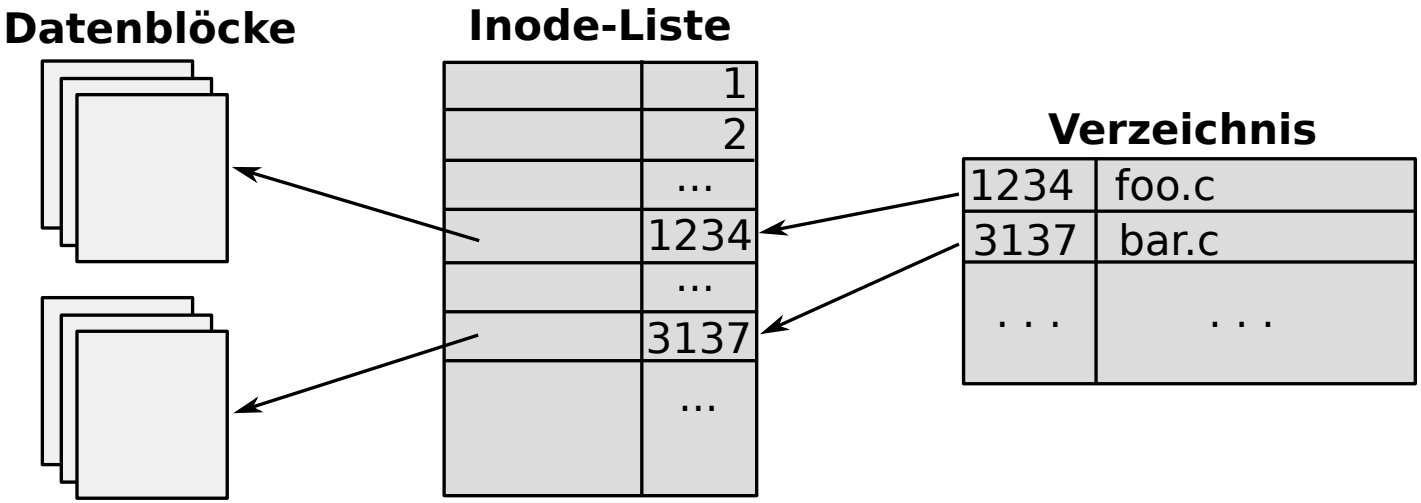
Ausgewählte ext-Dateiattribute

- Append only (a): Datei kann weder gelöscht noch verschoben werden. Es können weder Hardlinks angelegt werden noch vorhanden Bytes verändert werden
- no atime updates (A): Beim Dateizugriff wird der Zeitstempel atime nicht aktualisiert
- extent format (e): Dateiblöcke werden mittels Extents referenziert
- immutable (i): Datei kann nicht modifiziert werden
- synchronous updates (S): Schreiboperationen werden direkt – ohne eine Zwischenspeicherung im Puffercache – durchgeführt
- Die ext-Dateiattribute können mit dem Kommando `chattr` modifiziert und mit Kommando `lsattr` gelesen werden

Puffercache

- I/O-Operationen werden nicht direkt auf der Festplatte sondern auf einem Puffercache ausgeführt.
- Das physikalische Schreiben der Daten auf die Festplatte findet daher verzögert statt (delayed write).
- **Frage:** Warum werden die I/O-Operationen nicht direkt auf der Festplatte ausgeführt?
- Der Systemcall `void sync()` veranlasst das physikalische Schreiben aller ausstehenden Daten auf die Festplatte.
- Der Systemcall `int fsync(int fd)` veranlasst das physikalische Schreiben der ausstehenden Daten von `fd`.

6.3: Verzeichnisse



- Verzeichnisse werden unter Unix als Dateien realisiert
- Bei einem Verzeichnis handelt es sich um eine Liste mit Einträgen
- Ein Eintrag besteht aus der Inode-Nummer und dem Dateinamen
- Anlegen einer neuen Datei:
  - Initalisierung und Zuweisung eines freien Inodes
  - Anlegen eines neuen Verzeichnisseintrages

6.3: Verzeichnisse

Dieses Unterkapitel beschäftigt sich mit den folgenden Themen:

- Anlegen von Verzeichnissen
- Löschen von Verzeichnissen
- Verzeichniswechsel
- Erfragen des aktuellen ArbeitsVerzeichnisses
- Öffnen und Schließen von Verzeichnissen
- Lesen von Verzeichnisinhalten

Anlegen und Löschen von Verzeichnissen

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

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

- Bei Erfolg wird `0`, ansonsten `-1` zurückgegeben
- **`mkdir()`** versucht, ein Verzeichnis mit dem Namen `pathame` mit den Zugriffsrechten `mode` zu erzeugen
- Das neue Verzeichnis hat die effektiven Benutzer- und Gruppen-ID des aufrufenden Prozesses
- **`rmdir()`** entfernt ein leeres Verzeichnis `pathname`

Beispiel: `mkdir` I

```
1  #define _GNU_SOURCE
2  #include <errno.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <stdio.h>
7  #include <sys/stat.h>
8
9  #define P              1
10 #define V              2
11 #define MODE_BITS      3
12 #define MODE_GROUPS    4
13 #define DELIMITERCHAR '/'
14
15 static void error(char *msg) {
16   perror(msg);
17   exit(EXIT_FAILURE);
18 }
19
20 /*****************************************************************************/
21
22 static inline void usage(void) {
23   fputs("Usage: mkdir [-v] [-p] [-m mode] directory...\n"
24         "  -p    no error if existing, make parent directories as needed\n"
25         "  -m    set permission mode like -m 644\n"
26         "  -v print a message for each created directory",stderr);
27   exit(EXIT_FAILURE);
28 }
29
```

Beispiel: `mkdir` II

```
30
31  /* Function "cast" Decimal Number (e.g., 644) into mode. */
32  int itom(unsigned int x) {
33    unsigned int i, t, shift=0;
34    unsigned int erg=0;
35
36    /* Vier mal 3 bits: IS,USER,GROUP,OTHER */
37    for(i=1; i <= MODE_GROUPS; i++) {
38      t = x % (unsigned int) pow10(i);
39      t/= (unsigned int) pow10(i-1);
40      erg += t<<shift;
41      shift += MODE_BITS;
42      }
43    return erg;
44  }
45
46  /*****************************************************************************/
47
48  void domkdir(char *dirname, unsigned int mode, int iexist, int flags)  {
49    errno=0;
50    if(*dirname!=0 && mkdir(dirname,mode))
51      if(!iexist || errno!=EEXIST) error(dirname);
52
53    if((errno==0) && (flags&V))
54      printf("mkdir:␣created␣directory␣'%s'\n", dirname);
55  }
56
57
58
59
```

Beispiel: `mkdir` III

```
60  void withp(char *dirname, unsigned int mode, int flags) {
61    char *st, *temp;
62    temp = dirname;
63
64    if((int)*temp==DELIMITERCHAR) temp++;
65
66    if((st=alloca(strlen(dirname)))==NULL) error(dirname);
67
68      for(;*temp;temp++)  {
69        if((int)*temp==DELIMITERCHAR) {
70      memcpy(st,dirname,temp-dirname);
71      domkdir(st,mode,1,flags);
72    }
73      }
74    domkdir(dirname,mode,1,flags);
75  }
76
77  /*****************************************************************************/
78
79  int main(int argc, char *argv[]) {
80    unsigned int mode = 0777 - umask(0);
81    int flags=0;
82
83    if(argc<2) usage();
84    while(*++argv) {
85      if ((**argv=='-') && (strlen(*argv)==2))
86        switch (*++*argv)  {
87        case 'p': flags|=P; break;
88        case 'v': flags|=V; break;
89        case 'm':
```

Beispiel: `mkdir` IV

```
90          if(!++argv) usage();
91          if(!(mode = atoi(*argv))) usage();
92          mode = itom(mode);
93          break;
94        case 'h': usage(); break;
95        default : usage();
96        }
97      else
98        if (flags&P) withp(*argv,mode,flags);
99        else domkdir(*argv,mode,0,flags);
100   }
101   exit(errno);
102 }
```

Beispiel: `rmdir` I

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  #define I 1
8  #define P 2
9  #define V 4
10
11 /*****************************************************************************/
12
13 void usage() {
14   fputs("Usage: rmdir [-v] [-p] [-i] directory...\n"
15         "  -p    remove DIRECTORY and its ancestors\n"
16         "  -i    ignore fail on non empty directories\n"
17         "  -v    print a message for each removed directory\n", stderr);
18   exit(EXIT_FAILURE);
19 }
20
21 /*****************************************************************************/
22
23 static void error(char *msg) {
24   perror(msg);
25   exit(EXIT_FAILURE);
26 }
27
28 /*****************************************************************************/
29
```

Beispiel: `rmdir` II

```
30  void dormdir(char *dirname, int flags, int iexist) {
31    if (rmdir(dirname))
32      if (!iexist || errno!=EEXIST)
33        if(!(errno==ENOTEMPTY && flags&I))  error(dirname);
34    if ((!errno) && (flags&V))
35      printf("rmdir:␣removing␣directory␣'%s'\n", dirname);
36  }
37
38  /*****************************************************************/
39
40  void minusp(char *dirname, int flags) {
41    char *t=strrchr(dirname,'/');
42    do {
43      dormdir(dirname, flags,1);
44      if (!t) return;
45      *t=0;
46      t=strrchr(dirname,'/');
47    } while (1);
48  }
49
50  /*****************************************************************/
51
52  int main(int argc,char *argv[]) {
53    char *dirname;
54    int flags=0;
55
56    if(argc<2) usage();
57
58     while(*++argv)
59       if ((**argv=='-') && (strlen(*argv)==2))
```

Beispiel: `rmdir` III

```
60          switch (*++*argv) {
61          case 'p': flags|=P; break;
62          case 'v': flags|=V; break;
63          case 'i': flags|=I; break;
64          default: usage(); break;
65          }
66        else {
67          dirname = *argv;
68          if (flags&P) minusp(dirname, flags);
69          else dormdir(dirname, flags,0);
70        }
71      exit(EXIT_SUCCESS);
72  }
```

Verzeichniswechsel

```
#include <unistd.h>

int chdir(const char *pathname);
int fchdir(int fd);
char *getcwd(char *buf, size_t len);
```

- **(f)chdir()** Bei Erfolg wird **0**, ansonsten **-1**, zurückgegeben
- **chdir()**: Wechselt in des Verzeichnis `pathname`
- **fchdir()**: Wechselt in das Verzeichnis auf das der Dateideskriptor `fd` verweist
- **getcwd()** kopiert bis zu `len` Bytes des aktuellen Arbeitsverzeichnisses (absoluter Pfadname) nach `buf`. Bei Erfolg wird ein Zeiger auf `buf`, ansonsten `NULL`, zurückgegeben

Beispiel: chdir

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <limits.h>
5
6  int main(int args, char *argv[]) {
7    if (args >= 2)
8      if( chdir(argv[1]) ) {
9        perror(argv[1]);
10       return EXIT_FAILURE;
11     }
12   puts(getcwd(NULL,0));
13
14   return EXIT_SUCCESS;
15 }
```

Ein Verzeichnis öffnen

```
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

- Bei Erfolg wird ein Zeiger auf einen Verzeichnis-Datenstrom (Folge von Verzeichnis-Einträgen) `DIR` zurückgegeben, ansonsten `NULL`
- **`opendir()`** öffnet einen Verzeichnis-Datenstrom des Verzeichnisses `name`
- **`fdopendir()`** öffnet einen Verzeichnis-Datenstrom des Verzeichnisses auf das der Dateideskriptor `fd` verweist
- Beide Funktionen liefert einen Zeiger auf den ersten Eintrag des geöffneten Verzeichnisses zurück

Ein Verzeichnis schließen

```
#include <dirent.h>

int closedir(DIR *dirp);
```

- Bei Erfolg wird `0`, ansonsten `-1`, zurückgegeben
- **`closedir()`** schließt den Directory-Stream auf den `dirp` zeigt
- Ein erfolgreicher Aufruf schließt auch den unterliegenden, zu `dirp` gehörenden Dateideskriptor

Verzeichnisseinträge lesen

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);

struct dirent {
  ino_t d_ino:              /* Inode Nummer */
  char  d_name[255];        /* Dateiname (mit '\0') */
  unsigned char  d_type;    /* Dateityp (Linux, BSD)*/
}
```

- **`readdir()`** liefert einen Zeiger auf eine **`dirent`**-Struktur zurück, welche den nächsten Eintrag in dem Verzeichnis darstellt, auf den `dirp` weist
- Falls das Dateiende erreicht wurde oder ein Fehler auftrat, wird `NULL` zurückgegeben

Dateityp-Konstanten

```
#include<dirent.h>

DT_BLK     /* Blockorientiertes Geraet */
DT_CHR     /* Zeichenorientiertes Geraet */
DT_DIR     /* Verzeichnis */
DT_FIFO    /* FIFO (named Pipe) */
DT_LNK     /* Symbolischer Link */
DT_REG      /* Regulaere Datei */
DT_SOCK     /* UNIX Domain Socket */
DT_UNKNOWN  /* Unbekannter Dateityp */
```

Beispiel: Verzeichnisseinträge

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <dirent.h>
4  void error(char *msg, int usage) {
5    if(usage) fputs(msg, stderr);
6    else      perror(msg);
7    exit(EXIT_FAILURE);
8  }
9
10 char *get_type( unsigned char type) {
11   switch(type) {
12   case DT_BLK:  return "block␣device"; break;
13   case DT_CHR:  return "character␣device"; break;
14   case DT_DIR:  return "directory"; break;
15   case DT_FIFO: return "named␣pipe"; break;
16   case DT_LNK:  return "symbolic␣link"; break;
17   case DT_REG:  return "regular␣file"; break;
18   case DT_SOCK: return "UNIX␣domain␣socket"; break;
19   default: return  "unknown";
20   }
21 }
22
23 int main(int args, char *argv[]) {
24   DIR *dirp; struct dirent *entry;
25
26   if (args != 2) error("readdir␣<directory>\n",1);
27   if(!(dirp = opendir(argv[1]))) error(argv[1],0);
28
29   while( (entry = readdir(dirp)) )
30     printf("%s:␣%s\n\n", entry->d_name, get_type(entry->d_type));
31 }
```

Verzeichnispositionen verwalten

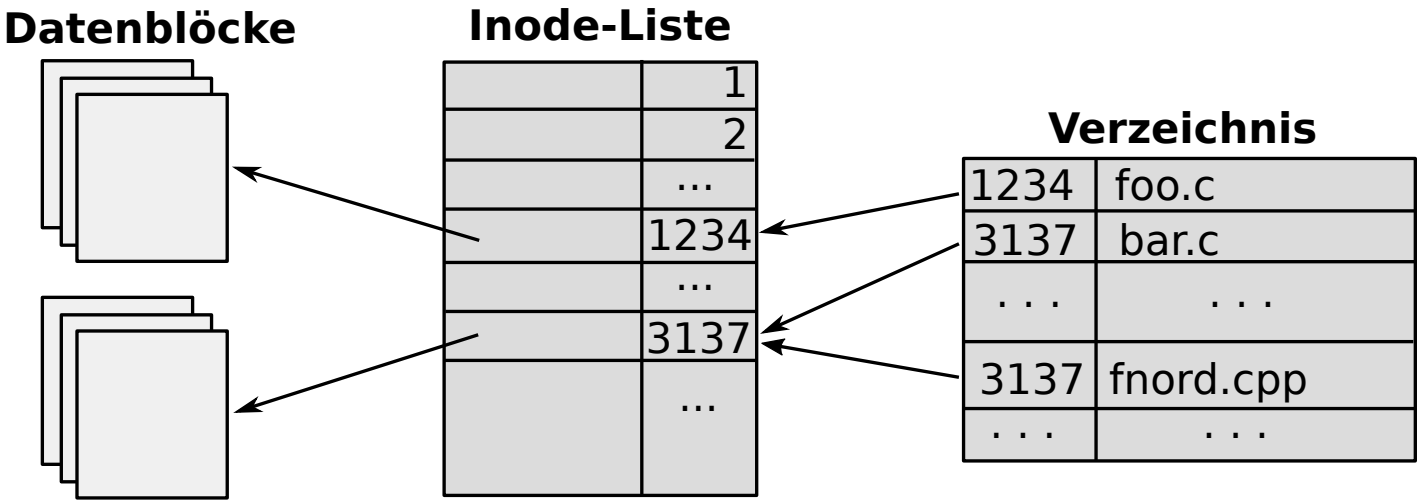
```
#include <dirent.h>

long telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
void rewinddir(DIR *dirp);
```

- **`telldir()`**: Gibt im Erfolgsfall die Position des Lesezeigers von `dirp` zurück, ansonsten **`-1`**
- **`seek()`**: Setzt die Position des Lesezeigers von `dirp` auf den Wert `loc`
- **`rewinddir()`**: Setzt die Position des Lesezeigers von `dirp` auf den ersten Eintrag des geöffneten Verzeichnisses zurück

Beispiel: Verzeichnispositionen verwalten

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <dirent.h>
4
5  void error(char *msg, int usage) {
6    if(usage) fputs(msg, stderr);
7    else      perror(msg);
8    exit(EXIT_FAILURE);
9  }
10
11
12 int main(int args, char *argv[]) {
13   DIR *dirp; struct dirent *entry;
14
15   if (args != 2) error("seekdir␣<directory>\n",1);
16   if(!(dirp = opendir(argv[1]))) error(argv[1],0);
17
18   while( (entry = readdir(dirp)) ) {
19     puts(entry->d_name);
20     seekdir(dirp, telldir(dirp) + 1);
21   }
22   closedir(dirp);
23
24   return EXIT_SUCCESS;
25 }
```

Hardlinks



- Anlegen eines Hardlinks: Beim Inode der Originaldatei wird der Linkzähler inkrementiert
- Löschen einer Datei: Der entsprechende Verzeichnisseintrag wird gelöscht und der Linkzähler des Inodes dekrementiert. Ist der Wert des Linkzählers danach 0, werden die Datenblöcke auf welche der Inode verweist sowie der Inode selbst freigegeben

Anmerkungen zu Hardlinks

- Hardlinks können mit dem Kommando `ln` angelegt werden
- Beispiel: `$ ln foo.c bar.c`
- Bei dem Verweis auf das Elternverzeichnis `..` handelt es sich in der Regel um ein Hardlink
- Das Anlegen von Hardlinks über Dateisystemgrenzen ist nicht möglich (**Warum?**)
- Wird eine Datei mittels `mv` innerhalb eines Dateisystem verschoben, wird einfach der alte Verzeichnisseintrag gelöscht und ein neuer mit der gleichen Inode-Nummer angelegt
- Verzeichnisseinträge gelten oftmals als gelöscht indem die Inode-Nummer auf 0 gesetzt wird
- **Frage:** Was passiert wenn man den Inhalt einer Datei verändert?

Erzeugen eines Hardlinks

```
#include <unistd.h>

int link(const char *pathname, const char *linkpath);
```

- Bei Erfolg wird `0`, ansonsten `-1`, zurückgegeben
- Hardlink `linkpath` wird von der Datei `pathname` erstellt
- Falls die Datei `linkpath` bereits existiert, wird diese **nicht** überschrieben

Beispiel: link

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  static void  usage() {
6    fputs("Usage: hardlink <src file> <dst file>", stderr);
7    exit(EXIT_FAILURE);
8  }
9
10 static void error(char *msg ) {
11   perror(msg);
12   exit(EXIT_FAILURE);
13 }
14
15 int main(int argc, char *argv[]) {
16   if (argc!=3)  usage();
17   if (access(argv[1], W_OK))  error(argv[1]);
18   if (link(argv[1], argv[2])) error(argv[2]);
19   return EXIT_SUCCESS;
20 }
```

Entfernen eines Verzeichnisseintrages

```
#include <unistd.h>

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

- Bei Erfolg wird **0**, ansonsten **-1**, zurückgegeben
- **unlink()** entfernt auch symbolische Links (Verknüpfungen)
- **unlink()** entfernt den Verzeichnisseintrag mit dem Dateinamen `name` aus dem Verzeichnis und dekrementiert den Linkzähler des Inodes um 1. War dies der letzte Link auf eine Datei, so wird diese gelöscht
- Ist die zu löschende Datei noch in Benutzung, bleibt sie bestehen bis der letzte auf sie weisende Dateideskriptor gelöscht wurde
- Bei temporären Dateien sollte nach **open()** direkt **ulink()** aufgerufen werden

Beispiel: Umgang mit Tempfiles

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <pwd.h>
5
6  static void error(char *msg) {
7    perror(msg);
8    exit(EXIT_FAILURE);
9  }
10
11 /*****************************************************************************/
12
13 int main() {
14   char *tmpfile = "ZZZZZtmp3t34g";
15   char *line = NULL;
16   size_t len = 0;
17   FILE *pfile = fopen(tmpfile, "w+");
18
19   if(!pfile) error(tmpfile);
20   if( unlink(tmpfile) ) error(tmpfile);
21
22   fputs(getpwuid(getuid())->pw_name, pfile);
23   fseek(pfile, 0, SEEK_SET);
24   sleep(5);
25   getline(&line, &len, pfile);
26   puts(line);
27   fclose(pfile);
28   return EXIT_SUCCESS;
29 }
```

Symbolische Links (Symlinks)

- Bei Symbolic Links (Symlinks) handelt es sich um Verknüpfungen (Verweis auf eine andere Datei)
- Symlinks können auch über Dateisystemgrenzen hinweg erzeugt werden
- Symlinks können mit dem Kommando
  `# ln -s <dst> <slink>` angelegt werden
- Bei einem Symlink wird ein neuer Inode *angelegt*. Der Inhalt des Symlinks ist der Dateiname der referenzierten Datei
- **Frage:** Warum verweist der Inode des Symlinks nicht einfach auf den Inode der referenzierten Datei?
- Beim Löschen der referenzierten Datei tritt beim Zugriff mittels Symlink ein Fehler auf
- Im Gegensatz zu **`open()`** folgt **`unlink()`** dem Symlink nicht

Spaß mit Symlinks

- **Frage:** Was macht die folgende Kommandosequenz?

```
$ touch fnord; ln -s fnord a; ln -s a b
```

- **Frage:** Was macht die folgende Kommandosequenz?

```
$ mkdir foo; cd foo; ln -s . bar
```

- **Frage:** Was macht die folgende Kommandosequenz?

```
$ mkdir foo; cd foo; ln -s . bar; ln -s .. foo
```

- Die meisten Systemcalls erkennen Endlos-Rekursionen und setzen `errno` auf `ELOOP`

Anlegen eines symbolischen Links

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);
```

- Bei Erfolg wird `0` zurückgegeben, ansonsten `-1`.
- **`symlink()`** erstellt einen symbolischen Link `linkpath` welcher die Zeichenkette `target` enthält
- Ein symbolischer Link kann auf eine nicht-existierende Datei zeigen. In diesem Fall handelt es sich um einen toten Link
- Falls `linkpath` bereits existiert, wird die vorhandene Datei **nicht** überschrieben

Inhalt eines symbolischen Links lesen

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buf, size_t
    bufsiz);
```

- Bei Erfolg wird die Anzahl gelesener Bytes, ansonsten **-1** zurückgegeben
- **open()** öffnet immer die Basisdatei auf welche der Symlink `pathname` zeigt
- **readlink()** öffnet den Symlink `pathname`, kopiert dessen Inhalt bis zu `bufsize` bytes nach `bufsize`. Im Anschluss wird der Symlink wieder geschlossen

Beispiel Symlinks

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define MAXLEN 4096

static void error(char *msg ) {
  perror(msg);
  exit(EXIT_FAILURE);
}

static void  usage() {
  fputs("Usage: symlink <target> <symlink>", stderr);
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
  char buf[MAXLEN];

  if(argc!=3)  usage();

  if( access(argv[1], F_OK) ) error(argv[1]);

  if( symlink(argv[1], argv[2]) ) error(argv[2]);

  if( readlink(argv[2], buf, MAXLEN) == -1 ) error(argv[2]);
  puts(buf);

  return EXIT_SUCCESS;
}
```

Zusammenfassung

Nach diesem Kapitel sollte Sie . . .

- . . . wissen was eine Gerätdatei ist.
- . . . wie man den Eigentümer und die Gruppe einer Datei ermittelt.
- . . . was ein Inode ist und wie ext3 bzw. ext4 auf Daten verweisen.
- . . . wie der Linux-Bootvorgang funktioniert.
- . . . was der Unterschied zwischen einem Hardlink und einem symbolischen Link ist.
- . . . wie man (symbolische) Links anlegt.
- . . . wie man Verzichnisse anlegt, liest und löscht.