12. Implementation des Minix-Dateisystems (Projekt, 1. Teil)

In dieser Aufgabe geht es darum, das Minix-Dateisystem unter Linux nachzuimplementieren: Minix ist ein Unix-System, das von Prof. Andrew S. Tanenbaum an der Freien Universität Amsterdam für Lehrzwecke entwickelt wurde, und es bringt ein eigenes Unix-Dateisystem mit, für das es auch unter Linux Support gibt.

Für diese Aufgabe werden Sie nur eine Auswahl der echten Features von Minix implementieren, u. a. verzichten wir hier darauf, Unterverzeichnisse anzulegen.

Ziel ist es, dass Sie auf eine Disketten-Image-Datei, die mit `mkfs.minix` (unter Linux) formatiert wurde, u. a.

- Dateien kopieren und diese wieder auslesen,
- das Inhaltsverzeichnis anzeigen,
- Hard-Links erzeugen und
- Dateien löschen

können. Da Sie dabei kompatibel zu anderen Minix-Implementierungen bleiben sollen, müssen Sie zunächst den exakten Aufbau eines Minix-Dateisystems verstehen.

Wenn Sie alle Projektaufgaben bearbeitet haben, ist Folgendes (bzw. eine Teilmenge davon möglich): Sie können Programme der folgenden Form schreiben:

```
#include "myminix.h"
int main () {
  mx_mount ("minix01.img","/");              // Image-Datei mounten
  mx_mkdir ("/mnt");                         // create folder /mnt
  mx_mount ("minix02.img","/mnt");           // noch ein Image
  fd1 = mx_open ("/test.txt", O_RDONLY);     // Datei öffnen
  fd2 = mx_open ("/mnt/test.txt", O_WRONLY); // noch eine
  char buf[1024]; int count;
  do {
    count = mx_read (fd1, &buf, 1024);       // lesen
    mx_write (fd2, &buf, count);             // und schreiben
  } while (count>0);
  mx_close (fd1); mx_close (fd2);            // Dateien schließen
  mx_umount ("/mnt"); mx_umount ("/");       // Images aushängen
}
```

und Sie können in Ihrer selbst geschriebenen Shell Befehle der folgenden Form eingeben:

```
spsh$ minix mount minix01.img /
spsh$ minix mkdir /mnt
spsh$ minix mount minix02.img /mnt
spsh$ minix cp /test.txt /mnt/test.txt
spsh$ minix ls -l /mnt
... test.txt ...
spsh$ minix umount /mnt
spsh$ minix umount /
```

Auch ein Austausch von Dateien zwischen Minix-Image und dem normalen Dateisystem soll über die Shell möglich sein. Was genau hier zu tun ist, wird sich aus diesem und den restlichen Übungsblättern ergeben. Keine Sorge: Wir werden uns der Aufgabe Schritt für Schritt annähern, am Anfang steht ein Verständnis für den Aufbau des Minix-Dateisystems.

Es gibt das Minix-Dateisystem in vier verschiedenen Varianten, welche sich u. a. in der maximalen Dateigröße und der maximalen Länge von Dateinamen (14 oder 30) unterscheiden. Sie können für ein Minix-Dateisystem-Image herausfinden, um welche Version es sich handelt. Wir behandeln hier nur die beiden Varianten, bei denen Dateinamen bis zu 30 Zeichen lang sein dürfen. Wenn Sie `mkfs.minix` nur den Image-Dateinamen übergeben, wird ein Dateisystem vom ersten Typ (`minix1`, 30 Zeichen pro Dateiname, ca. 256 MByte theoretische max. Dateigröße) erstellt, über die Option `-v` erzeugen Sie ein Dateisystem vom zweiten Typ (`minix2`, 30 Zeichen pro Dateiname, ca. 2 GByte max. Dateigröße).

Ein Minix-Dateisystem ist in **Blöcke** der Größe 1 KByte (1024 Byte) unterteilt. Der erste Block ist der Bootsektor (den werden wir ignorieren), der zweite Block ist der sog. **Superblock**, der alle wichtigen Informationen über das Dateisystem enthält.

Da Sie häufig mit Blöcken arbeiten werden, brauchen Sie zunächst zwei Funktionen

```
void readblock  (int blockno, char* block);
void writeblock (int blockno, char* block);
```

mit denen Sie blockweise auf das Dateisystem zugreifen können. Ob Sie das Dateisystem-Image einfach über `open()` öffnen und dann mit `lseek()`, `read()` und `write()` darauf zugreifen oder die Datei über eine `mmap()` in den Prozessspeicher einblenden und die Transfers mit `memcpy()` erledigen, bleibt Ihnen überlassen.

Der Superblock (also die absoluten Bytes 1024-2047 der Image-Datei) hat folgenden Aufbau:

```
struct minix_superblock {
        uint16_t s_ninodes;
        uint16_t s_nzones;
        uint16_t s_imap_blocks;
        uint16_t s_zmap_blocks;
        uint16_t s_firstdatazone;
        uint16_t s_log_zone_size;
        uint32_t s_max_size;
        uint16_t s_magic;
        uint16_t s_state;
        uint32_t s_zones;
};
```

Die Typen `uint16_t` und `uint32_t` sind in der Datei `/usr/include/stdint.h` definiert und 16 Bit bzw. 32 Bit breite vorzeichenlose Integer-Zahlen. Der Superblock verwendet also nur 24 Bytes.

Wenn Sie den Superblock in einen `struct minix_superblock` kopieren, können Sie die Werte ausgeben lassen; eine Analyse dieser Werte sollte der erste Schritt sein.

Am Eintrag `s_magic` erkennen Sie, um welche Version des Minix-Dateisystems es sich handelt; uns interessieren nur die beiden Fälle

- Minix v1 (30 Zeichen pro Dateiname): 5007 = 0x138F
- Minix v2 (30 Zeichen pro Dateiname): 9336 = 0x2478

(Für die Varianten mit kürzeren Dateinamen gibt es zwei weitere Magic-Nummern.) Ohne diese Information auszuwerten, ist kein Zugriff auf das Dateisystem möglich, weil die beiden Versionen sich in Größe und Inhalt eines Inodes unterscheiden.

Bei einem v1-Dateisystem steht die Anzahl der Blöcke im Eintrag `s_nzones`, ein v2-Superblock verwendet dafür den Eintrag `s_zones`. Der jeweils andere Wert ist 0.

Statt von Blöcken spricht Minix immer von Zones, dabei ist i. d. R. eine Zone genau ein Block; theoretisch könnte aber eine Zone auch aus mehreren Blöcken bestehen: Es gilt

$$\text{Zonengröße} = \text{Blockgröße} \times 2^{\texttt{s\_log\_zone\_size}}$$

(und der Wert `s_log_zone_size` ist normal 0). Für diese Aufgabe können Sie ungeprüft davon ausgehen, dass immer Zonengröße = Blockgröße = 1024 gilt.

`s_imap_blocks` und `s_zmap_blocks` geben an, wie viele Blöcke von der **Inode-Bitmap** und der **Zone-Bitmap** belegt sind: In diesen Bitmaps steht für jeden Inode bzw. für jeden Datenblock (jede Daten-Zone), ob diese frei oder belegt sind: 0 bedeutet frei, 1 bedeutet belegt.

Die Inode-Bitmap folgt direkt auf den Superblock, dahinter kommt die Zone-Bitmap. Gleich danach folgt die Inode-Tabelle, und schließlich beginnen die Datenblöcke.

In der Zone-Bitmap steht das erste Bit (Bit 0) für keinen Datenblock und ist immer gesetzt; das zweite Bit (Bit 1) steht für den ersten Datenblock, welcher immer die ersten Verzeichniseinträge des Wurzelverzeichnisses enthält und nie frei sein kann, also ist auch das zweite Bit immer gesetzt; die noch davor liegenden Blöcke werden hier nicht erfasst, weil sie keine Dateiinhalte aufnehmen können.

Ein Inode sieht – abhängig von der Minix-Dateisystem-Version – wie folgt aus:

```
struct minix1_inode {
        uint16_t i_mode;
        uint16_t i_uid;
        uint32_t i_size;
        uint32_t i_time;
        uint8_t  i_gid;
        uint8_t  i_nlinks;
        uint16_t i_zone[9];
};
```

```
struct minix2_inode {
        uint16_t i_mode;
        uint16_t i_nlinks;
        uint16_t i_uid;
        uint16_t i_gid;
        uint32_t i_size;
        uint32_t i_atime;
        uint32_t i_mtime;
        uint32_t i_ctime;
        uint32_t i_zone[10];
};
```

Die Größe eines Inodes ist also 14 + 2*9 = 32 Bytes (v1) bzw. 24 + 4*10 = 64 Bytes (v2), damit passen in einen Inode-Block 1024/32 = 32 Inodes (v1) bzw. 1024/64 = 16 Inodes (v2).

Betrachten wir das Layout einer 1,4-MByte-Diskette, die mit Minix v2 formatiert (`mkfs.minix -v`) wurde. Eine Analyse des Superblocks ergibt:

```
s_ninodes:        480
s_nzones:         0
s_imap_blocks:    1
s_zmap_blocks:    1
s_firstdatazone: 34
s_log_zone_size: 0
s_max_size:       2147483647
s_magic:          9336
s_state:          1
s_zones:          1440
```

Wir haben 480 Inodes. Bei 16 Inodes pro Block benötigt die Inode-Tabelle 480/16 = 30 Blöcke. Die Inode-Bitmap besteht nur aus 480 Bits (=60 Bytes) und passt in einen Block, der Rest dieses Blocks wird nicht verwendet (und beim Formatieren mit 1-Bits gefüllt).

Bei 1440 Blöcken werden 1440 Bits (=180 Bytes) für die Zone-Bitmap benötigt, auch das passt wieder in einen Block. Auch hier wird der Rest des Blocks mit 1-Bits gefüllt.

Daraus ergibt sich folgende Aufteilung:

• Block 0	unbenutzt (Bootsektor)	0- 1023	0x0000-0x03ff
• Block 1	Superblock	1024- 2047	0x0400-0x07ff
• Block 2	Inode-Bitmap	2048- 3071	0x0800-0x0bff
• Block 3	Zone-Bitmap	3072- 4095	0x0c00-0x0fff
• Block 4–33	Inode-Tabelle (30 Blöcke)	4096-34815	0x1000-0x87ff
• Block 34-1439	Datenblöcke	34816- ...	0x8800- ...

Dass es in Block 34 mit den Daten losgeht, steht auch im Superblock im Feld `s_firstdatazone`.

Um das Wurzelverzeichnis des Dateisystems zu lesen, muss man nun wissen, dass Minix für das Wurzelverzeichnis `/` die Inode-Nummer 1 verwendet und auch bei 1 zu zählen beginnt; Inode 1 findet sich also an Position 0 in der Tabelle, der zugehörige Bitmap-Eintrag ist aber Bit 1 (*nicht* Bit 0, siehe unten)!

Im Inode finden Sie nun in `i_zone[0]` die Nummer des ersten Datenblocks. Der Block besteht aus Verzeichniseinträgen, welche die folgende Struktur haben:

```
struct minix_dir_entry {
        uint16_t inode;
        char name[30];
};
```

Jeder Eintrag ist also 2+30 = 32 Bytes groß und enthält zuerst die Inode-Nummer und dann den zugeordneten Dateinamen. Achtung: Wenn der Dateiname 30 Zeichen lang ist, dann ist er nicht (!) 0-terminiert. Für die interne Speicherung von Dateinamen (in Ihrem Programm) sollten Sie also 31 Zeichen reservieren und immer für eine korrekte 0-Terminierung des Strings sorgen.

In einen Block passen 1024 / 32 = 32 solche Verzeichniseinträge. Gibt es mehr als 32, werden weitere Blöcke für dieses Verzeichnis verwendet. Wie groß das Verzeichnis insgesamt ist, steht in seinem Inode-Eintrag `i_size`.

Nützliche Tools: Für die folgenden Aufgaben können Sie einige hilfreiche Tools verwenden:

- `dd`: zum Erzeugen leerer (unformatierter) Disketten-Images:
  `dd if=/dev/zero of=image.img bs=1024 count=1440`
- `mkfs.minix`: erstellt ein Minix-Dateisystem – auch in einer Image-Datei. Option `-v` zum Erzeugen eines v2-Systems
- `hexdump`: zeigt Inhalte von Binärdateien (wie z. B. Image-Dateien) an. Sie können das Tool auch als `hd` aufrufen, um die meist gewünschte gemischte Hex-/ASCII-Ausgabe zu erzeugen:

  ```
  $ hd image1.img | head -3
  00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
  *
  00000400  e0 01 00 00 01 00 01 00  22 00 00 00 ff ff ff 7f  |........".......|
  ```

- `bindump`: Manchmal ist auch eine binäre Ausgabe nützlich. Da ich kein passendes Tool dafür gefunden habe, habe ich Ihnen ein kleines Programm (bindump) gebastelt, das Sie auf der Kurs-Webseite finden. Nach dem Kompilieren können Sie es als Filter verwenden:

  ```
  $ ./bindump < bindump.c | head -5
  00000000  00101111 00101111 00100000 01100010 01101001 01101110 01100100 01110101  // bindu
  00000008  01101101 01110000 00001010 00101111 00101111 00100000 01001000 01100001  mp.// Ha
  00000010  01101110 01110011 00101101 01000111 01100101 01101111 01110010 01100111  ns-Georg
  00000018  00100000 01000101 01110011 01110011 01100101 01110010 00101100 00100000   Esser,
  00000020  01010011 01111001 01110011 01110100 01100101 01101101 01110000 01110010  Systempr
  ```

- `bindump` hat eine Option `-r`, welche die Reihenfolge der Bits umdreht (z. B. `11000000` statt `00000011` für den Byte-Wert 3). Sie ist nützlich für die Anzeige der Bitmaps.
- `mount`: Wenn Sie Zugang zu einem Linux-Rechner haben, auf dem Sie Root-Rechte besitzen, können Sie Minix-Images auch mounten. Um die Datei `image1.img` unter `/mnt` einzubinden, würden Sie z. B. den folgenden Befehl verwenden:

  ```
  sudo mount -o loop image1.img /mnt
  ```

  (Die Option `-o loop` ist nötig, damit Linux das Mounten einer Datei erlaubt.) Das Dateisystem erkennt `mount` dabei selbständig.
- Wenn auf einem Linux-Rechner (ohne Root-Rechte) `qemu` installiert ist, können Sie ein kleines Linux-Live-System (z. B. die CD-Variante von Knoppix) herunterladen und mit

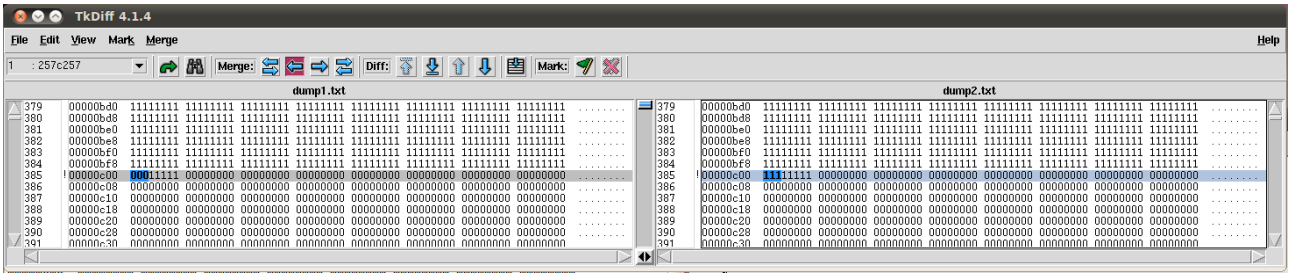
  ```
  qemu -cdrom linux.iso -fda image1.img
  ```

  starten; dann können Sie im virtuellen Linux-System mit

  ```
  mount /dev/fd0 /mnt
  ```

  das Minix-Image einbinden. Statt `qemu` können Sie auch andere Virtualisierungssoftware (VirtualBox, VMware etc.) verwenden.

- `tkdiff` und `diff`: Mit den diff-Tools können Sie zwei Dateien vergleichen und sich z. B. die Unterschiede zwischen Hexdumps von zwei Image-Dateien anzeigen lassen. `tkdiff` (falls installiert) ist dabei deutlich übersichtlicher.



Aufgaben

Für die folgenden Aufgaben können Sie die Minix-Image-Dateien `minix1.img` und `minix2.img` von der Webseite herunterladen (siehe auch folgende Seite).

a) Implementieren Sie die Funktionen `readblock()` und `writeblock()`, die oben schon vorgestellt wurden:

```
void readblock  (int blockno, char* block);
void writeblock (int blockno, char* block);
```

Ihr Programm muss anfangs herausfinden, welche Image-Datei zu verwenden ist; dafür können Sie eine beliebige Methode wählen, z. B. feste Angabe in einer Dateinamen-Konstanten oder Auswertung von `argv[1]`.

Die Methoden können davon ausgehen, dass `block` ein Zeiger auf einen Puffer der Größe 1024 ist.

b) Nutzen Sie `readblock()`, um den Superblock (den zweiten Block) einzulesen. Deklarieren Sie ein `struct minix_superblock` und kopieren Sie die Nutzdaten aus dem Puffer in dieses Struct. Geben Sie alle Elemente (`s_ninodes` etc.) aus.

c) Erweitern Sie Ihr Programm um Funktionen, welche die Inode-Bitmap und die Zone-Bitmap ausgeben können. Die Bitmaps sollen dabei in zwei Formen angezeigt werden:

– als Liste von Nullen/Einsen mit Angabe der jeweiligen Position (wie im Programm `bindump`):
`111001000000000...`

– als Liste der belegten Inodes bzw. Zones:
`belegte Inodes: 0, 1, 2, 5`

d) Schreiben Sie Funktionen `get_imap_bit()`, `set_imap_bit()` und `clear_imap_bit()` sowie `get_zmap_bit()`, `set_zmap_bit()` und `clear_zmap_bit()`. Sie sollen einzelne Bits aus den Inode- und Zone-Bitmaps auslesen bzw. auf 0 (`clear`) oder 1 (`set`) setzen.

Um aus einer 8-Bit-Integer-Zahl `n` das Bit `i` auszulesen ($0 \le i \le 7$), können Sie die Formel

`(n >> i) % 2` (Rechts-Shift um `i` Positionen, dann mit `%2` das unterste Bit lesen)

verwenden.

e) Implementieren Sie zwei Funktionen `request_inode()` und `request_block()`, welche mit Hilfe der jeweiligen Bitmap den ersten freien Inode (bzw. den ersten freien Block) finden, diesen als belegt markieren und die Inode-Nummer (bzw. die Blocknummer / Zone-Nummer) zurückgeben. Wenn alle Inodes bzw. alle Blöcke belegt sind, sollen die Funktionen `-1` zurückgeben.

Beachten Sie, dass (wie weiter oben beschrieben) die Zone-Bitmap nicht mit dem Eintrag für Block 0 beginnt, sondern mit dem Zustand von Block 33. (Der erste Datenblock ist im Beispiel 34.) Um den Zustand von Block `n` abzufragen, können Sie also `get_zmap_bit (n-s_firstdatazone+1)` verwenden. (Überlegen Sie sich, warum das so ist – setzen Sie `n=34` ein.)

Beachten Sie auch die Hinweise zu den Image-Dateien auf den folgenden Seiten.

Anhang

Im Folgenden erhalten Sie ein paar Informationen über die Image-Dateien `minix1.img` und `minix2.img` von der Webseite, die Sie verwenden können, um die Ausgabe Ihrer Tools zu überprüfen.

`minix1.img` wurde mit `dd` und `mkfs.minix -v` erstellt:

```
# dd if=/dev/zero of=minix1.img bs=1024 count=1440
1440+0 Datensätze ein
1440+0 Datensätze aus
1474560 Bytes (1,5 MB) kopiert, 0,00353603 s, 417 MB/s
# mkfs.minix -v minix1.img
480 inodes
1440 Blöcke
Firstdatazone=34 (34)
Zonesize=1024
Maxgröße=2147483647
```

Dann wurde `minix1.img` nach `/mnt` gemountet und die Datei `Testdatei1.txt` (6144 Bytes, hex.: 0x1800, das sind genau 6 Blöcke) hinein kopiert. Mit `sed -e 's/ei1.txt/ei2.txt' < Testdatei1.txt > Testdatei2.txt` wurde eine leicht veränderte Kopie (`Testdatei2.txt`), mit `ln` ein Link (`Hardlink.txt`) und mit `ln -s` ein Symlink `Symlink.txt` (jeweils von Textdatei1.txt) erstellt.

Bei der Anzeige des Wurzelverzeichnis erscheint:

```
/mnt# ls -il
insgesamt 19
2 -rw-r--r-- 2 root root 6144 2012-06-04 23:32 Hardlink.txt
4 lrwxrwxrwx 1 root root   14 2012-06-04 23:33 Symlink.txt ->
                                                    Testdatei1.txt
2 -rw-r--r-- 2 root root 6144 2012-06-04 23:32 Testdatei1.txt
3 -rw-r--r-- 1 root root 6144 2012-06-04 23:32 Testdatei2.txt
/mnt# ls -ild /mnt
1 drwxr-xr-x 2 root root 192 2012-06-04 23:33 /mnt
```

Es sind also die Inodes 1 – 4 belegt (siehe erste Spalte).

Wenn Sie das Image mit `hd minix1.img` betrachten, werden Sie das Inhaltsverzeichnis und den Inhalt der beiden Dateien entdecken.

Zu den belegten Blöcken:

- Das Wurzelverzeichnis `/` belegt Block 34.
- Die Datei `Testdatei1.txt` belegt Block 35-40.
- Die Datei `Testdatei2.txt` belegt Block 41-46.
- Der Symlink `Symlink.txt` belegt Block 47. (Auch Symlinks brauchen Datenblöcke!)
- Weitere Blöcke sind nicht belegt, der Hardlink ist nur ein weiterer Eintrag im Wurzelverzeichnis.

Die Inode-Bitmap sieht wie folgt aus (Ausschnitt aus `bindump -r < minix1.img`):

```
00000800  11111000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000808  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000810  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
```

Die fünf Einsen stehen für Inode-Nummern 0 – 4; Inode 0 gibt es aber nicht. Die Zone-Bitmap sieht so aus:

```
00000c00  11111111 11111110 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c08  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c10  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
```

(Die 15 gesetzten Bits beziehen sich auf die Blöcke 33-47. Block 33 gehört noch zur Inode-Liste und ist

kein Datenblock!)

Bei einem frisch formatierten Minix-Dateisystem (keine Dateien, aber es gibt das leere Wurzelverzeichnis) sehen Inode- und Zone-Bitmap so aus:

```
00000800  11000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000808  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
...
00000c00  11000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000c08  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
...
```

Dort sind also zwei Inode-Nummern (0, 1) und die Blöcke 33 und 34 als belegt markiert. Inode 0 gibt es nicht, Inode 1 ist das Wurzelverzeichnis. Ein Verzeichnis ist nie leer, denn es enthält immer die Einträge „`.`“ und „`..`“.

In der letzten Zeile des Hexdumps von `minix1.img` finden Sie den Text `Testdatei1.txt` – hier handelt es sich um den Inhalt des Symlinks: In dessen Datenblock wird der verlinkte Dateiname gespeichert.

`minix2.img` ist als Kopie des oben beschriebenen Image `minix1.img` entstanden, es wurden dann noch zwei weitere Dateien (`Testdatei3.txt` und `bindump.c`) hinzugefügt und diese schließlich alle mit `cp -a *.c *.txt subdir/` in ein mit `mkdir` erzeugtes Unterverzeichnis `subdir/` kopiert. Sie können mit diesem Image zusätzlich arbeiten, um ein komplexeres Beispiel zu sehen.

```
root@ubu64:/mnt# ls -liRd
1 drwxr-xr-x 3 root root 288 2012-06-05 00:23 .
root@ubu64:/mnt# ls -liR
.:
insgesamt 36
6 -rw-r--r-- 1 root root  1236 2012-06-05 00:23 bindump.c
2 -rw-r--r-- 2 root root  6144 2012-06-05 00:11 Hardlink.txt
7 drwxr-xr-x 2 root root   256 2012-06-05 00:23 subdir
4 lrwxrwxrwx 1 root root    14 2012-06-05 00:12 Symlink.txt -> Testdatei1.txt
2 -rw-r--r-- 2 root root  6144 2012-06-05 00:11 Testdatei1.txt
3 -rw-r--r-- 1 root root  6144 2012-06-05 00:11 Testdatei2.txt
5 -rw-r--r-- 1 root root 13072 2012-06-05 00:23 Testdatei3.txt

./subdir:
insgesamt 35
12 -rw-r--r-- 1 root root  1236 2012-06-05 00:23 bindump.c
 8 -rw-r--r-- 2 root root  6144 2012-06-05 00:11 Hardlink.txt
 9 lrwxrwxrwx 1 root root    14 2012-06-05 00:23 Symlink.txt -> Testdatei1.txt
 8 -rw-r--r-- 2 root root  6144 2012-06-05 00:11 Testdatei1.txt
10 -rw-r--r-- 1 root root  6144 2012-06-05 00:11 Testdatei2.txt
11 -rw-r--r-- 1 root root 13072 2012-06-05 00:23 Testdatei3.txt
```

Die Ausgabe mit `bindump` zeigt für die Inode- und Zone-Bitmaps:

```
# bindump -r < minix2.img
00000800  11111111 11111000 00000000 00000000 00000000 00000000 00000000 00000000  ........
00000808  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
[...]
00000c00  11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000  ........
00000c08  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  ........
...
```

Also belegte Inodes: 0 – 12, belegte Blöcke: 33 – 93.