13. Implementation des Minix-Dateisystems (Projekt, 2. Teil)

Wir setzen die Implementierung des Minix-Dateisystems fort. Wenn Sie Aufgabe 12 nicht erfolgreich bearbeiten konnten, können Sie als Basis die rudimentäre Musterlösung von der Webseite verwenden (`u7-mini-loesung.c`), die allerdings keine Fehlerprüfung vornimmt und die Aufgabenstellung teilweise vereinfacht. (Sie sollten diese dann zu einem späteren Zeitpunkt mit Ihrem Lösungsansatz integrieren bzw. so ausbauen, dass sie auch Fehler abfängt und Situationen unterstützt, die in größeren Images auftauchen können.)

Sie können nun bereits den Superblock, die Inode-Bitmap und die Zone-Bitmap auslesen, in den Bitmaps einzelne Bits abfragen / setzen / löschen und generell ganze Blöcke lesen und schreiben. Mit diesen Mitteln lässt sich einiges anfangen, denn die Vorgehensweise beim Erstellen einer neuen Datei ist einfach:

1. Inode reservieren (mit `request_inode`)
2. Benötigte Blöcke reservieren (mit `request_block`) und Blocknummern im Inode vermerken (dafür gibt es noch keine Code)
3. Inhalt in die reservierten Blöcke schreiben (mit `writeblock`)
4. Eintrag im Verzeichnis erzeugen (Zuordnung Dateiname → Inode-Nummer)

Das einzige, was Sie bisher nicht können, ist das Bearbeiten von Inodes und Verzeichnissen – darum geht es in diesem Teil des Projekts. Nach dem Lösen der heutigen Aufgaben können Sie neue Dateien im Minix-Image anlegen (sofern deren Inhalte in sieben Datenblöcke passen, also maximal 7 KByte groß sind).

Vorab eine wichtige Erinnerung an die **Pointer-Arithmetik**: Sie werden bei den folgenden Aufgaben gelegentlich mit `memcpy()` Daten in die Mitte eines im Speicher gehaltenen Blocks schreiben müssen. Wenn Sie den Block als `char block[1024];` deklariert haben und jetzt innerhalb das Blocks z. B. ab Position 512 einen 32 Byte langen Inhalt schreiben möchten, wird der Aufruf

```
offset = 512; size = 32;
memcpy (&block + offset, &daten, size);
```

fehlschlagen. Wenn Sie `block` hingegen als `char* block;` deklarieren, funktioniert

```
offset = 512; size = 32;
memcpy (block + offset, &daten, size);
```

wie erwartet. Das folgende Beispielprogramm verdeutlicht den Unterschied.

```
#include <stdio.h>
int main () {
  char block[1024]; char* block2=(char*)&block; char daten[]="Test";
  int size = sizeof(daten); int offset = 512; long diff;

  printf ("&block:          %p \n", &block);
  printf ("&block + offset: %p \n", &block + offset);
  diff = (long)(&block+offset)-(long)&block;
  printf ("Differenz:       %ld \n", diff);

  printf ("block2:          %p \n", block2);
  printf ("block2 + offset: %p \n", block2 + offset);
  diff = (long)(block2+offset)-(long)block2;
  printf ("Differenz:       %ld \n", diff);
};
```

erzeugt die folgende Ausgabe:

```
esser@ubu64:/tmp/z2$ ./offset-test
&block:          0x7ffff31802b0
&block + offset: 0x7ffff32002b0
Differenz:       524288                    /* das ist 512 x 1024 ! */
block2:          0x7ffff31802b0
block2 + offset: 0x7ffff31804b0
Differenz:       512                       /* so soll es sein... */
```

Beim ersten Versuch wird also die Größe eines `char[1024]` berücksichtigt und mit dem Offset multipliziert (Pointer-Arithmetik). Darum müssen Sie in diesem Fall immer erst auf `(char*)` casten, sonst landen Sie an völlig falschen Speicherstellen.

a) Wir starten mit den Funktionen `read_inode()` und `write_inode()`, die Sie wie folgt implementieren sollen:

```
int read_inode  (int i, struct minix2inode* inodeptr);
int write_inode (int i, struct minix2inode* inodeptr);
```

Rückgabewert: 0 bei Fehler, sonst i (erlaubt Tests mit `if (!read_inode(...) { /* Fehler */ }`). Ein Fehler ist es auch, einen nicht belegten Inode anzufordern!

Sie können sich diese Aufgaben erleichtern, wenn Sie überlegen, welche Unterschiede zwischen dem Lesen und dem Schreiben bestehen und zunächst eine allgemeine Funktion

```
int read_write_inode (int i, struct minix2_inode* inodeptr,
                      int wr_flag);
```

schreiben, die beides kann (abhängig vom Flag `wr_flag`). Erstellen Sie zudem eine Funktion

```
void show_inode (struct minix2_inode* inode);
```

welche die Felder eines vorher mit `read_inode()` eingelesenen Inodes ausgibt. Die Beschreibung der hier benötigten Datenstrukturen finden Sie auf dem letzten Aufgabenblatt.

b) Blocknummern sind vom Typ `uint32_t`, also 32 Bit große vorzeichenlose Integers. Im Inode stehen die absoluten Blocknummern der verwendeten Datenblöcke. Einträge, die auf keinen Block zeigen sollen, enthalten die (ungültige) Blocknummer 0.

Sie können eine Liste von zu belegenden Blöcken als Liste verwalten (`uint32_t[]` oder `uint32_t*`). Die Liste soll immer mit 0 terminiert sein. Schreiben Sie zwei Funktionen

```
int read_block_list  (int i, uint32_t* blocklist);
int write_block_list (int i, uint32_t* blocklist);
```

welche zu einem angegebenen Inode (`i`) die zugehörigen Blöcke zurückgeben bzw. diese in den Inode schreiben. Der Rückgabewert beider Funktionen ist die Anzahl der Blöcke, im Fehlerfall `-1`. (Der Fehlercode ist nicht 0, weil es zulässig ist, eine Datei ohne Blöcke zu verwenden: eine leere Datei).

c) Jetzt können Sie neue Datenblöcke, die Sie mit `request_block()` reserviert haben, mit `write_block_list()` in einen neuen Inode schreiben, welchen Sie mit `request_inode()` angefordert haben; der Pseudocode für das Erzeugen einer neuen (mit Nullen gefüllten) Datei sieht also so aus:

```
create_null_file (int size, char* filename) {
  int nblocks = ...  // Anzahl benötigter Blöcke bestimmen
  uint32_t* blist = malloc ((nblocks+1)*sizeof(uint32_t));
  for (int i = 0; i<nblocks; i++)
    *blist++ = request_block ();
  *blist = 0;
  int inodenr = request_inode ();
  write_block_list (inodenr, blist);
  // weitere Eintraege im Inode
```

```
  struct minix2_inode inode;
  read_inode (inodenr, &inode);
  ... // inode->uid, inode->gid, ...
  write_inode (inodenr, &inode);
  // Datenblöcke mit Nullen füllen
  for (int i = 0; i<nblocks; i++)
    ...
  write_link (inodenr, filename);  // Verzeichniseintrag anlegen
  free (blist);
};
```

Implementieren Sie diese Funktion vollständig (bis auf den Verzeichniseintrag, darum geht es in Aufgabe **d)** – die Funktion `write_link()` können Sie zunächst als Rumpf ohne Funktion ergänzen.

Hinweis: Beschränken Sie sich zunächst auf Dateien, die in sieben Datenblöcke passen – für größere Dateien müssen Sie indirekte und ggf. doppelt indirekte Blockadressierung verwenden, siehe nächste Übung. (Dabei enthält `inode->i_zone[7]` die Adresse eines Indirektionsblocks, der Adressen von Datenblöcken speichert; `inode->i_zone[8]` enthält die Adresse eines Doppelindirektionsblocks, welcher Adressen von Indirektionsblöcken speichert. `inode->i_zone[9]` ist für künftige Verwendungen, etwa dreifache Indirektion, reserviert und wird nie benutzt. Mehr dazu in der nächsten Woche.)

Der Inode soll auch sinnvolle Angaben bei den Zugriffsrechten (`mode=0644`, oktal) und Besitzer und Gruppe (`uid=gid=0`) enthalten. Berücksichtigen Sie auch `nlinks`! Die drei Timestamps können Sie auf einen beliebigen Wert setzen (z. B. 1338847929 für ein aktuelles Datum).

Testen Sie die Funktion, indem Sie `create_null_file()` (mit einer Größe von maximal 7 KByte) aufrufen und sich dann den (neuen) Inode Nr. 5 ausgeben lassen.

d) Jetzt fehlt noch die Funktion `write_link()`, welche dem Wurzelverzeichnis `/` einen neuen Eintrag hinzufügt: den Verweis von `filename` auf den Inode `inodenr`.

Verzeichnisse sind, wie schon erwähnt, spezielle Dateien; das Wurzelverzeichnis hat die Inode-Nummer 1. Die von dort verlinkten Datenblöcke enthalten je bis zu 32 Verzeichniseinträge vom Typ:

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

(denn 32 x 32 = 1024). Ein nicht verwendeter Eintrag im Verzeichnis ist durch `inode=0` gekennzeichnet.

Um einen Verzeichniseintrag auszulesen bzw. zu schreiben, erstellen Sie die beiden Funktionen

```
int read_dir_entry  (int inodenr, int entrynr,
                     struct minix_dir_entry* entry);
int write_dir_entry (int inodenr, int entrynr,
                     struct minix_dir_entry* entry);
```

Hier können Sie auch wieder den Trick aus Aufgabe **a)** verwenden und zunächst eine gemeinsame Funktion `int read_write_dir_entry (..., int wr_flag)` erstellen. Das erste Argument der Funktion gibt den Inode des Verzeichnisses an (damit lässt sich die Funktion später auf andere Verzeichnisse als / erweitern), danach folgt die Nummer des Verzeichniseintrags und schließlich ein Pointer auf einen Verzeichniseintrag (`minix_dir_entry`).

Wie schon beschrieben, passen in einen Block nur 32 Verzeichniseinträge – wenn Sie mehr als 32 Einträge in einem Verzeichnis haben, müssen Sie

- einen weiteren Block reservieren und in den Verzeichnis-Inode eintragen
- in diesem Block das Anlegen der Einträge fortsetzen

Genauso ist es später beim Löschen von Einträgen evtl. nötig, leer gewordene Zusatzblöcke wieder

zu entfernen. Sie erleichtern sich die Lösung, indem Sie dieses Problem zunächst ignorieren und erst nach der erfolgreichen Implementierung für ≤ 32 Einträge die Funktionen so erweitern, dass sie mit mehr als 32 Einträgen zurecht kommen.

Jetzt können Sie die in Aufgabe **c)** bereits erwähnte Funktion `write_link()` implementieren: Sie müssen nur in einer Schleife mit `read_dir_entry()` einen freien Eintrag suchen (erkennbar an `entry.inode = 0`) und diesen dann für den neuen Eintrag verwenden.

Während `write_dir_entry()` keine Überprüfungen durchführt, muss `write_link()` prüfen, ob ein Dateiname bereits vorhanden ist, denn derselbe Dateiname darf nicht mehrfach in einem Verzeichnis auftauchen.

Ihr Programm darf zur Vereinfachung annehmen, dass ein Verzeichnis nie mehr als 224 (= 7 x 32) Einträge enthält – dann passen die Einträge alle in die ersten sieben Datenblöcke, welche über den Verzeichnis-Inode direkt erreichbar sind (`i_zone[0]` bis `i_zone[6]`).

`write_link()` muss am Ende auch die Größe des Verzeichnisses anpassen (Eintrag `i_size` im Inode des Verzeichnisses)! Sie können dazu am Ende

```
struct minix2_inode inode;
read_inode (1, &inode);
if (inode.i_size < 32*(i+1)) {   // i ist Nummer des neuen
  inode.i_size = 32*(i+1);       // Eintrags
  write_inode (1, &inode);
};
```

ausführen.

e) Implementieren Sie eine Funktion `void list_dir (int i)`, die als Argument die Inode-Nummer eines Verzeichnisses erhält (diese Nummer wird im Beispiel immer 1 für das Wurzelverzeichnis sein). Sie soll dann alle Verzeichniseinträge auslesen. Wenn ein Eintrag belegt ist, soll sie über die darin gespeicherte Inode-Nummer den zugehörigen Inode auslesen und dann in einer Zeile Inode-Nummer, Dateiname, Link-Count und Dateigröße ausgeben:

```
 1 .                            2     192
 1 ..                           2     192
 2 Testdatei1.txt               2    6144
 3 Testdatei2.txt               1    6144
 2 Hardlink.txt                 2    6144
 4 Symlink.txt                  1      14
```

f) Testen Sie Ihr Programm mit

```
create_null_file (5300, "hallo.txt");
create_null_file (5300, "hallo.txt");   // Fehler!
create_null_file (5300, "hallo2.txt");
printf ("Verzeichnis /:\n");
list_dir (1);
```

Sie sollten folgende Ausgabe erhalten:

```
FEHLER: Name hallo.txt schon vorhanden!
Verzeichnis /:
 1 .                            2     192
 1 ..                           2     192
 2 Testdatei1.txt               2    6144
 3 Testdatei2.txt               1    6144
 2 Hardlink.txt                 2    6144
 4 Symlink.txt                  1      14
 5 hallo.txt                    1    5300
 9 hallo2.txt                   1    5300
```