14. Implementation des Minix-Dateisystems (Projekt, 3. Teil)

Wir setzen die Implementierung des Minix-Dateisystems fort. Wenn Sie Aufgabe 13 nicht erfolgreich bearbeiten konnten, können Sie als Basis die rudimentäre Musterlösung von der Webseite verwenden (`u8-mini-loesung.c`) – mit den gleichen Einschränkungen wie bei der letzten Aufgabe.

Diesmal geht es darum, die Funktionen `open()`, `read()`, `lseek()` und `close()` nachzubilden. Es soll damit ein Zugriff auf Dateien im Wurzelverzeichnis des Minix-Dateisystems möglich sein, und Sie sollen bis zu 16 Dateien gleichzeitig geöffnet halten können. Die neuen Funktionen werden analog `mx_open()`, `mx_read()`, `mx_lseek()` und `mx_close()` heißen.

Datenstrukturen, die Sie im Programm benötigen, sind die folgenden:

- **interner Inode:** das ist eine Kopie des Inodes aus dem Minix-Dateisystem, aber mit zusätzlichen Elementen, z. B. einem Reference Count `refcount`, der mitzählt, wie oft die darüber erreichbare Datei geöffnet ist. Sobald Sie eine Datei öffnen, legen Sie einen solchen internen Inode an; Änderungen an den Metadaten speichern Sie zunächst nur im internen Inode, erst beim Schließen der Datei werden diese Informationen ins Dateisystem zurückgeschrieben. (Für sofortiges Sichern der Daten wird es eine Funktion `mx_sync()` geben, siehe Aufgabe **d**.) Eines der nötigen Zusatzfelder im internen Inode heißt `clean` und ist 1, solange keine Änderungen am internen Inode vorgenommen wurden. Jede Änderung setzt den Wert auf 0, jeder Aufruf von `mx_sync()` setzt den Wert (am Ende) wieder auf 1.

  Für interne Inodes wird bei Bedarf mit `malloc()` Speicherplatz organisiert und mit `free()` wieder freigegeben, wenn der interne Inode nicht länger verwendet wird.

- **MFD** (Minix File Descriptor): ein nicht-negativer Integer-Wert, den `mx_open()` zurückgibt und den die übrigen Funktionen für den Dateizugriff verwenden.

- **Dateistatus:** Ein Struct, das die folgenden Elemente enthält:

  - Zeiger auf einen internen Inode: Steht hier ein Nullzeiger, ist der Eintrag unbenutzt. Ansonsten handelt es sich um den internen Inode einer geöffneten Datei.
  - Dateiposition: Der Standardinhalt (für unbenutzte Einträge) ist -1. Beim Öffnen einer Datei wird der Wert 0 eingetragen; Ausnahme: Beim Öffnen im Append-Modus wird `i_size` eingetragen.
  - Modus: `O_RDONLY`, `O_WRONLY`, `O_RDWR` oder `O_APPEND`

  Für die folgenden Beschreibungen nehmen wir an, dass der Aufbau so aussieht:

```
struct filestat {
  int_inode_t* int_inode;
  int pos;
  short mode;
}
```

- **Statusliste:** Array, das 16 Dateistatus-Structs enthält:

```
struct filestat status[16];
```

a) In Aufgabe 13 haben Sie die Funktion `create_null_file()` implementiert, mit der Sie eine neue Datei mit angegebener Größe und angegebenem Dateinamen erzeugen können. Erstellen Sie eine Kopie dieser Funktion namens `create_empty_file()`, welche weniger leistet: Sie erzeugt einfach eine leere Datei, muss also keine Datenblöcke reservieren und in `i_zone[i]` für alle `i` Nullen eintragen. Wir brauchen diese Funktion in der nächsten Übung für `mx_write`.

b) Schreiben Sie eine Funktion `mx_open()`, welche im Wesentlichen wie der System-Call-Wrapper `open()` arbeitet:

- Es wird der erste freie MFD verwendet (0..15). Die Nummer sei `n`.
- Zunächst werden die MFDs durchsucht:
  - **Fall 1:** Wenn die Datei bereits mit dem MFD `i` geöffnet ist, wird `status[n].int_inode` auf `status[i].int_inode` gesetzt und im internen Inode `refcount` um 1 erhöht. (Der interne Inode wird also mehrfach genutzt.)
  - **Fall 2:** War die Datei noch nicht geöffnet, wird das Wurzelverzeichnis durchsucht – wenn der angegebene Dateiname gefunden wird, wird der zugehörige Inode in den zum MFD gehörenden internen Inode kopiert. (Interne Inodes sollten darum am Anfang exakt wie normale Inodes aufgebaut sein, die Zusatzfelder folgen erst dahinter.) Im internen Inode werden `clean` auf 1 und `refcount` (zunächst) auf 0 gesetzt. Speichern Sie im internen Inode auch die Nummer des „echten" Inodes im Dateisystem.
- In `status[n].pos` wird ein geeigneter Wert (0 oder `i_size`) eingetragen.
- Im internen Inode wird `refcount` erhöht.
- `status[n].mode` nimmt den Öffnen-Modus auf. (So ist es möglich, dieselbe Datei z. B. 1x zum Lesen und 1x zum Schreiben zu öffnen – dabei wird nur einziger interner Inode verwendet, Modus und aktuelle Lese-/Schreibposition werden aber getrennt gespeichert.)
- Danach gibt die Funktion den Wert `n` zurück – Ende.
- Wenn das Öffnen fehlschlägt, gibt die Funktion -1 zurück. `errno` wird nicht gesetzt.

c) Ergänzen Sie eine Funktion `mx_close()`, die das Gegenstück zu `mx_open()` ist, also geöffnete Dateien wieder schließt. Beachten Sie dabei, dass eine Datei mehrfach geöffnet sein kann. Der `refcount` wird beim Schließen dekrementiert – wenn er den Wert 0 erreicht, soll der zugehörige interne Inode freigegeben werden. Ist `clean = 0`, wird vorher der „externe Anteil" des internen Inodes in das Dateisystem zurückgeschrieben.

d) Implementieren Sie eine Funktion `mx_sync(int mfd)`, welche Änderungen am internen Inode zum MFD `mfd` sofort zurück schreibt und danach im internen Inode `clean` auf 1 setzt.

e) Implementieren Sie die Funktion `mx_lseek()`, welche dieselben Parameter wie `lseek()` akzeptiert. Anstelle des File Descriptors wird ein Minix File Descriptor (MFD) verwendet. Wenn der angegebene MFD gültig ist, wird die zugehörige Dateiposition geeignet gesetzt. Für erste Tests reicht es, die Variante mit absoluter Positionierung (`whence = SEEK_SET`) zu implementieren.

f) Schreiben Sie eine Funktion `mx_read()`, welche dieselben Parameter wie `read()` akzeptiert. Anstelle des File Descriptors wird wieder ein MFD verwendet. Diese Funktion ist komplex:

- Wenn die Datei nur zum Schreiben oder im Append-Modus geöffnet ist oder der angegebene MFD nicht gültig ist, gibt die Funktion direkt den Wert -1 zurück.
- Ansonsten muss sie die aktuelle Dateiposition berücksichtigen und zunächst bestimmen, welche (logischen) Blöcke der Datei gelesen werden müssen,
- dann aus dem Inode die zugehörigen Blocknummern extrahieren und mit `readblock()` die Blöcke vom Dateisystem lesen
- und schließlich die relevanten Teile in den angegebenen Puffer kopieren.
- Schließlich muss noch die Dateiposition angepasst werden, wobei die Funktion berücksichtigen muss, dass evtl. beim Lesen das Dateiende erreicht wurde – in dem Fall ist die Dateiposition `i_size`.
- Der Rückgabewert ist die Anzahl der tatsächlich gelesenen Zeichen

g) Testen Sie Ihre Funktionen mit den Dateien, die sich im Test-Image befinden, und schicken Sie mir per E-Mail eine dokumentierte Version Ihres Programms (h.g.esser@gmx.de).