15. Implementation des Minix-Dateisystems (Projekt, 4. Teil)

Wir setzen die Implementierung des Minix-Dateisystems fort. Wenn Sie Aufgabe 14 nicht erfolgreich bearbeiten konnten, können Sie als Basis die rudimentäre Musterlösung von der Webseite verwenden (`u9-mini-loesung.c`) – mit den gleichen Einschränkungen wie bei der letzten Aufgabe.

Es fehlt unserem Dateisystem noch die Funktion `mx_write()`, welche das Verhalten von `write()` nachbildet.

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

- Wenn die Datei nur zum Lesen 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 geschrieben werden müssen,
- dann anhand des Inode feststellen, ob die zu schreibenden Blöcke bereits belegt sind (also alte Daten verändert werden) oder ob die Datei durch das Schreiben größer wird (also neue Blöcke mit `request_block()` angefordert werden müssen). Im zweiten Fall reservieren Sie zunächst die nötigen Blöcke und aktualisieren den Inode.
- Dann muss die Funktion aus dem Inode die zugehörigen Blocknummern extrahieren und mit `writeblock()` aus dem Puffer die Daten in die Blöcke des Dateisystems schreiben. (Dieser und der vorherige Schritt lassen sich auch zusammenfassen.)
- Schließlich muss noch die Dateiposition angepasst werden, der neue Wert ist die Position hinter dem letzten geschriebenen Byte.
- Der Rückgabewert ist die Anzahl der tatsächlich geschriebenen Zeichen.
- Sie müssen keine Indirektion erlauben; beim Versuch, eine Datei so zu verändern, dass sie größer als 7 KByte wird, soll die Funktion die Datei bis zur 7-KByte-Grenze füllen und durch den Rückgabewert deutlich machen, dass nicht alle Daten geschrieben wurden. Die Dateiposition ist danach 7 KByte + 1.

b) Testen Sie `create_empty_file()` (aus der letzten Übung) und `mx_write()`.

Projektaufgaben

Bearbeiten Sie – abhängig von der Matrikelnummer – eine der folgenden Aufgaben:

- Aufgabe **A**, wenn Ihre Matrikelnummer ohne Rest durch 3 teilbar ist
- Aufgabe **B**, wenn beim Teilen Ihrer Matrikelnummer durch 3 der Rest 1 bleibt
- Aufgabe **C**, wenn beim Teilen Ihrer Matrikelnummer durch 3 der Rest 2 bleibt

(Beispiel: Matr. Nr. 875321 → 875321:3 = 291773, Rest 2 → Aufgabe C). Wenn Sie einen Tauschpartner finden, dürfen Sie mit einem anderen Teilnehmer die Aufgabe tauschen. (Ziel ist, dass jede Aufgabe ungefähr gleich oft bearbeitet wird.)

Sie dürfen Aufgaben auch gemeinsam bearbeiten (maximal Zweiergruppen), in dem Fall werden aber höhere Ansprüche an Ihre Lösung gestellt. Beachten Sie bitte, dass beide Mitglieder einer Zweiergruppe denselben Vortragstag haben müssen.

Es folgen die Aufgabenstellungen für die drei Projektaufgaben A, B und C. Lassen Sie sich Zeit für die Planung und zerlegen Sie die Aufgabe(n) in sinnvolle Teilprobleme.

A. Minix-Dateisystem in der Mini-Shell

Integrieren Sie die Mini-Shell und die Implementation des Minix-Dateisystems: Wenn einer der zwei Befehle `mv` oder `cp` verwendet wird, soll die Shell die Argumente untersuchen. Beginnt eines der Argumente (oder beide) mit dem Präfix `mx:`, dann soll dieses Argument als Name einer Datei in einem Minix-Image interpretiert werden.

Beispiele:

`cp /tmp/datei.txt mx:datei.txt`	kopiert `datei.txt` in das Minix-Image
`mv mx:datei.txt /tmp/verschoben`	kopiert `datei.txt` aus dem Image nach `/tmp/verschoben` und löscht das Original
`cp mx:1.txt mx:2.txt`	kopiert im Image `1.txt` nach `2.txt`

Damit das funktioniert, müssen Sie erkennen, ob Minix-Dateien gelesen oder geschrieben werden sollen. Entsprechend müssen Sie vor oder beim Ausführen des eigentlichen Kommandos eine temporäre Datei erzeugen, die im Dateisystem liegt. Überlegen Sie, welches die richtigen Reihenfolgen für die verschiedenen Varianten sind.

Das einzubindende Image können Sie fest verdrahten. Für „Bonuspunkte“ spendieren Sie der Mini-Shell zwei neue Kommandos `mxmount` und `mxumount`, über die Sie auswählen, welche Image-Datei verwendet wird. Wenn schon ein Image verwendet wird, sollen weitere Aufrufe von `mxmount` so lange fehlschlagen, bis Sie `mxumount` (ohne Parameter) aufrufen. Wenn aktuell kein Image verwendet wird, sollen Befehle, die das Präfix `mx:` verwenden, fehlschlagen.

B. Verzeichnisse

Bisher haben wir nur mit dem Wurzelverzeichnis gearbeitet. Erweitern Sie die Funktion `open()`, so dass sie auch Pfade in Dateinamen akzeptiert (z. B. `/a/b/datei.txt`).

Der Dateiname muss dazu zunächst in seine Pfadbestandteile (`a`, `b`, `datei.txt`) zerlegt werden. Dann müssen Sie die verschiedenen Inodes für die Verzeichnisse suchen und das Inhaltsverzeichnis lesen (im Beispiel: zunächst `a` im Wurzelverzeichnis, dann `b` im Verzeichnis `a`).

Schließlich finden Sie (hier) im Verzeichnis `b` den Eintrag für `datei.txt` und damit die Inode-Nummer dieser Datei.

Schreiben Sie Funktionen `mkdir()` und `rmdir()`: `mkdir()` erzeugt ein neues, leeres Unterverzeichnis (und akzeptiert nur absolute Pfadangaben). `rmdir()` löscht ein Verzeichnis (wenn es leer ist) und erwartet ebenfalls eine absolute Pfadangabe.

Ergänzen Sie außerdem eine Funktion `chdir()`, mit der sich das aktuelle Arbeitsverzeichnis ändern lässt. Speichern Sie das aktuelle Arbeitsverzeichnis in einer globalen Variable (Voreinstellung: /). Die `open()`-Funktion soll das Arbeitsverzeichnis berücksichtigen, wenn Sie einen relativen Pfad angeben (also z. B. `b/datei.txt`, wenn `/a` das aktuelle Arbeitsverzeichnis ist). open() kann dabei einfach prüfen, ob der verwendete Pfad mit / beginnt (dann ist er absolut) oder nicht (dann ist es relativ). Bei einem relativen Pfad setzt `open()` zunächst das aktuelle Arbeitsverzeichnis und den relativen Pfad zu einem absoluten Pfad zusammen (`/a` + `b/datei.txt` → `/a/b/datei.txt`). (Beim Aneinanderhängen mehrerer Strings hilft die `strncat()`-Funktion.)

C. Defragmentierung

Minix-Dateisysteme können fragmentieren: Wenn Sie z. B. zwei Dateien zum Schreiben öffnen und dann im ständigen Wechsel jeweils 1 KByte Daten an die erste und dann 1 KByte an die zweite Datei anhängen, gehören die Datenblöcke im Dateisystem jeweils im Wechsel den beiden Dateien.

Bestimmen Sie zunächst für jede Datei (im Wurzelverzeichnis), ob diese fragmentiert ist – sie ist fragmentiert, wenn mindestens ein Datenblock nicht direkt hinter dem vorherigen Datenblock liegt. Implementieren Sie eine Funktion, welche den Grad der Fragmentierung ausgibt (den prozentualen Anteil der fragmentierten Dateien).

Schreiben Sie dann eine Funktion, welche das Dateisystem defragmentiert: Die Blöcke einer Datei sollen im Dateisystem unmittelbar aufeinander folgen. Sie dürfen zum Zwischenspeichern den Arbeitsspeicher verwenden. (Ein echter Defragmentierer könnte das nicht, weil Dateisysteme viel größer als der Arbeitsspeicher sein können.)

Der Defragmentierer soll die Datenböcke dabei großzügig über das Image verteilen, so dass hinter jeder Datei noch einige freie Blöcke verbleiben. Passen Sie dann die Funktionen `mx_write()` und `request_block()` so an, dass beim Anhängen weiterer Daten an eine Datei diese direkt auf das Ende folgenden, freien Blöcke verwendet werden. `request_block()` benötigt dazu einen neuen Parameter, der die von `mx_write()` „gewünschte“ Blocknummer angibt.

- Wenn hier 0 übergeben wird, soll `request_block()` wie gewohnt arbeiten.
- Wird eine Blocknummer ungleich 0 übergeben, soll `request_block()` versuchen, diesen Block zu reservieren.
- `mx_write()` wird also beim Vergrößern einer Datei immer `request_block()` mit Argument `n+1` aufrufen, wobei `n` die Nummer des letzten bereits von der Datei belegten Blocks ist.

Bei einem bereits fragmentierten Dateisystem kann es vorkommen, dass es keine solchen Blöcke gibt; in dem Fall wird beim Schreiben einfach (wie vorher) der nächste freie Block verwendet.

Testen Sie die Funktion, indem Sie zunächst (wie oben beschrieben) ein stark fragmentiertes Minix-Image erzeugen. Lassen Sie sich die Blocknummern der Dateien ausgeben, rufen Sie dann die Defragmentierfunktion auf und prüfen Sie, dass danach die Fragmentierung aufgehoben ist. Die Dateiinhalte dürfen dabei nicht zerstört werden.

Abgabeformalitäten

Die fertig bearbeitete und gut dokumentierte Lösung Ihrer Projektaufgabe soll mit dem folgenden Header beginnen:

```
// Systemprogrammierung, Projektaufgabe A  (oder B, C)
// Bearbeiter:
// 1. Max Mustermann, Matr.Nr. 12345
// 2. Martha Musterfrau, Matr.Nr. 23456
```

Wenn Sie die Möglichkeit des Aufgabentauschs genutzt haben, setzen Sie hinter die Matr.-Nr. noch in Klammern den Namen des Tauschpartners, etwa so:

```
// 2. Martha Musterfrau, Matr.Nr. 23456 (Tausch mit Martin Muster)
```

Wenn Sie Ihre Aufgabe nicht vollständig lösen konnten, beschreiben Sie in Ihrer Mail, was Sie versucht haben und welche Probleme / Fehler dabei aufgetreten sind.

Die Lösungsdatei(en) schicken Sie bitte per E-Mail mit einem kurzen Vorschlag, über welchen Aspekt Ihrer Lösung Sie vortragen möchten, bis zum **29.06.2012** an mich (h.g.esser@gmx.de).

Bereiten Sie einen ca. zehn Minuten dauernden Vortrag vor, in dem Sie Ihren Lösungsansatz beschreiben. Wenn Sie mit einem Partner zusammen gearbeitet haben, werden Sie gemeinsam einen ca. 15 Minuten dauernden Vortrag halten, bei dem Sie sich abwechseln. Nach jedem Vortrag ist ca. fünf Minuten Zeit für Fragen der anderen Teilnehmer (und von mir).