Dateisystem mit FUSE

Dokumentation Betriebssysteme Labor
Wintersemester 2018/2019

dosa1013, depa1016, odda1011

29. Januar 2019

Inhaltsverzeichnis

1. Aufgabenstellung und Vorgaben

1.1 Aufgabenstellung

Die uns gestellte Aufgabe in diesem Labor bestand darin, ein Dateisystem zu erstellen. Dieses soll verwendet werden, um einen Dateiträger zu formatieren und somit Dateien mit den gewöhnlichen Attributen zu unterstützen. Diese Attribute umschließen zum Beispiel den Namen, die Größe, die Zugriffsrechte und die verschiedenen Zeitstempel der Dateien. Weiterhin soll ermöglicht werden, dass ein Datenträger, der mit dem Dateisystem formatiert wurde, in einem Verzeichnisbaum eingetragen wird. Die Einbindung soll dann in einem freien, wählbarem und leerem Verzeichnis erfolgen, in welchem der Inhalt des Datenträgers anschließend erscheinen soll.

Anstatt wie bei traditionellen Dateisystemen mit Daten auf einem Datenträger zu arbeiten, verwendeten wir eine Containerdatei. Außerdem schreiben wir das Dateisystem nicht als Kernel Modul, sondern nutzen FUSE. FUSE, was für "File System In User Space" steht, hilft einem dabei, wie der Name schon sagt, Dateisysteme zu schreiben ohne dabei auf Kernel Ebene programmieren zu müssen und wird dementsprechend vor allem bei virtuellen Dateisystemen verwendet.

Während bei traditionellen Dateisystemen mit Kernel-Programmierung Anfragen aus dem User Space den VFS (Virtual Filesystem Switch) durchlaufen müssen, gefolgt vom Block-Layer, dem Input-Output-Layer, dem Gerätetreiber bis schließlich der Datenträger erreicht wird, wird das bei FUSE Dateisystemen vermieden.

Stattdessen werden Anfragen aus dem User Space vom VFS direkt an FUSE weitergegeben, welches sich um die betroffenen Bereiche kümmert. FUSE führt anschließend ein Programm aus, gibt diesem die Anfragen weiter und erhält eine Antwort, die es zum anfragenden Programm weiterleitet. Wodurch das virtuelle Dateisystem sich praktisch ebenfalls im User Space befindet.

Im ersten Aufgabenteil war das Ziel mittels dem Kommando `mkfs.myfs` eine Containerdatei zu erstellen, die alle nötigen Strukturen enthält. Beim Erstellen sollen ausgewählte Dateien einmalig in die Containerdatei kopiert werden. Nachdem diese durch FUSE in den Verzeichnisbaum eingebunden wurde, soll es möglich sein Dateien lesen zu können, aber noch nicht das Bearbeiten oder Löschen. Beim oben beschriebenen Aufgabenteil war vor allem das Design des Dateisystems entscheidend was den Aufbau und die Einteilung verschiedener Elemente angeht. Anschließend ging es ans Erstellen und Befüllen des Datenträgers durch das Kommando `mkfs.myfs`. Das Einbinden erfolgte über das Kommando `mount.myfs`. Schlussendlich deckten wir mögliche Fehlerquellen mit ausführlichen Testfällen ab.

Das Ziel des zweiten Aufgabenteils war es, nach dem Einbinden des Datenträgers, Dateien bearbeiten und löschen zu können. Hierfür mussten die FUSE-

Operationen zum Anlegen, Ändern, Schreiben und Löschen von Dateien implementiert werden.

Der dritte und letzte Teil der Laborabgabe ist das Anfertigen der Dokumentation, über das von uns verfasste Programm.

1.2 Vorgaben

Das zu erstellende Dateisystem soll eine Größe von mindestens 30 MB Platz für Dateien anbieten. Dementsprechend war die Größe aller Verwaltungsstrukturen zu beachten um sicherzugehen, dass genug Platz für Dateien übrig bleibt. Alle Dateien, die sich im Dateisystem befinden sollen sich im Rootverzeichnis befinden und es soll keine weiteren Verzeichnisse geben.

Als Konstanten wurde festgelegt: - Die maximal Länge eines Dateinamens `NAME_LENGTH` auf 255 Charaktere. - Die logische Blockgröße `BLOCK_SIZE` mit 512 Byte. - Die maximale Anzahl an Verzeichniseinträgen `NUM_DIR_ENTRIES` mit 64. - Die Anzahl der maximal geöffneten Dateien `NUM_OPEN_FILES` mit ebenfalls 64.

Die Übung soll von 2-4 Studenten durchgeführt werden.

Zur Bearbeitung des Projektes wurde ein Template bereitgestellt, welches folgendes beinhaltet:

- Makefile
- Testframework “Catch”
- diverse `.cpp` und `.h` Dateien, die nützliche Funktionalitäten und Vorlagen enthalten.

2. Read-Only Filesystem

2.1 Container Datei Aufbau

Grundbestandteil des Dateisystems ist die Erstellung der Containerdatei und diese anschließend zu befüllen. Die Containerdatei muss mindestens eine Größe von 30 MB besitzen und wird in 512 Byte große Blöcke aufgeteilt. Maximal sollen bis zu 64 Dateien gespeichert werden können. Als Gesamtgröße wählten wir 33.324.544 Bytes (32 MB) welche 65.536 Blöcke entsprechen. Das hat den Vorteil, dass wir über ein `uint16_t` (16 Bit Integer) alle Blöcke adressieren können.

Die Blöcke werden wie folgt genutzt:

Name	Superblock	DMAP	FAT	Rootverzeichnis	Dateien
Größe	1	128	256	64	65087
Blockindex	0	1-128	129-384	385-448	449-65535

- **Block 0** Enthält den Superblock, welcher nicht benutzt wird.
- **Block 1-128** Enthält die DMAP in der abgelegt wird, welche Blöcke frei oder belegt sind. Dabei wird entweder der Charakter F für free, oder A für allocated an die Stelle des Datenblocks geschrieben. Um alle 65.536 Blöcke über ein Byte abzubilden, muss dieser Bereich 128 Blöcke groß sein.
- **Block 129-384** Enthält eine File Allocation Table (FAT) in der an der Stelle einer Blocknummer die Nummer des darauf folgenden Blocks abgelegt wird, welche die nächste Blocknummer beinhaltet. Beim letzten Block wird eine 0 geschrieben. Da pro Blocknummer werden 2 Byte benötigt werden, muss dieser Bereich 256 Blöcke groß sein.
- **Block 385-448** Enthält das Rootverzeichnis mit Einträgen für jede im Dateisystem gespeicherte Datei. Diese Einträge beinhalten den Dateinamen, die Dateigröße, Benutzer/Gruppen-ID, Zugriffsberechtigungen, Zeitstempel für den letzten Zugriff/Veränderung/Statusänderung und den Zeiger auf den ersten Datenblock. Pro Datei wird hier ein Block benutzt. Da in unserem Dateisystem höchstens 64 Dateien Platz finden, wird dieser Bereich mit 64 Blöcken bemessen.
- **Block 449-65.535** Datenblöcke für Dateien. Abhängig von der Größe der Datei werden mehrere Blöcke verwendet.

2.2 Eigene Klassen und Methoden

Für jeden der oben genannten Sektoren wird eine eigene Klasse bereitgestellt. Diese Klassen haben Methoden für jegliche Interaktion mit dem Sektor und werden in `mkfs.myfs.cpp` und `myfs.cpp` verwendet.

2.2.1 DMAP-Methoden

`void DMAP::create()`

Erstellt die DMAP, mit allen Bytes auf Free gesetzt.

`int DMAP::getFree(uint16_t *pos)`

Kann verwendet werden um einen leeren Block aus der DMAP zu bekommen.

`int DMAP::getFree(uint16_t num, uint16_t *arr)`

Über diese Methode können mehrere freie Blöcke auf einmal erhalten werden.

`void DMAP::allocate(uint16_t pos)`

Setzt einen bestimmten Block in der DMAP auf Allocated.

`void DMAP::allocate(uint16_t num, uint16_t *arr)`

Setzt mehrere Blöcke auf Allocated.

2.2.2 FAT-Methoden

`uint16_t FAT::read(uint16_t curAddress)`

Liest die nächste Adresse aus der angegebenen Adresse.

`void FAT::write(uint16_t curAddress, uint16_t nextAddress)`

Schreibt die nächste Adresse an die angegebene Adresse.

2.2.3 dpsFile-Struct

Um alle Informationen zu einer Datei zu speichern, wird ein eigenes Struct verwendet, welches pro Datei im Rootverzeichnis gespeichert wird.

```
struct dpsFile {
    char name[NAME_LENGTH]; // Name der Datei
    struct stat stat;       // Stat-Struct
    uint16_t firstBlock;    // Addresse der ersten Datenblocks
};
```

2.2.4 Rootdir-Methoden

`int RootDir::get(const char *name, dpsFile *fileData)`

Ruft die Dateiinformationen über den Dateiname ab.

`int RootDir::exists(const char *name)`

Prüft, ob eine Datei im Dateisystem vorhanden ist.

`int RootDir::read(uint16_t num, dpsFile *fileData)`

Liest Dateiinformationen aus dem Rootverzeichnis für die Datei mit der Nummer `num`.

`int RootDir::write(dpsFile *fileData)`

Schreibt Dateiinformationen in das Rootverzeichnis. Wenn der gleiche Namen im Rootverzeichnis schon vorhanden ist, wird dieser überschrieben, sonst wird ein neuer Eintrag angelegt.

2.2.5 Files-Methoden

`int Files::read(uint16_t *blocks, uint16_t num, uint16_t offset, char *buf)`

Liest eine Anzahl `num` von Blöcken aus der Containerdatei.

`int Files::write(uint16_t *blocks, uint16_t num, uint16_t offset, size_t size, const char *buf)`

Schreibt eine Anzahl von Blöcken in die Containerdatei.

2.3 Container Erstellung und Befüllung

Die Containerdatei wird mit einem Kommandos erstellt mit folgendem Aufbau: `mkfs.myfs containerfile [input-file ...]`. Die einzelnen Dateien zum Befüllen werden an den Befehl angehängt. Wenn keine Dateien angehängt werden, wird eine leere Containerdatei erzeugt.

2.4 Einbinden und Lesen

Das Einbinden des Dateisystems erfolgt durch das Kommando `mount.myfs container logfile mountdir -s`. Der Parameter `-s` steht für Single-Threaded-Mode, der verwendet werden soll um potentielle Fehler zu vermeiden, die durch Multithreading auftreten können. Im Template wurde uns das Programm `mount.myfs` übergeben, welches sicherstellt, dass das Kommando die richtige Syntax befolgt und benötigten Dateien die Voraussetzungen erfüllen. Sofern der Zugriff auf den Container und die Logdatei erfolgreich war und ein Mountpoint existiert, kann die Initialisierung erfolgen.

2.4.1 FUSE-Methoden

`*MyFS::fuseInit`

Öffnet die Logdatei und Containerdatei. Danach werden alle Container-Objekte (DMAP, FAT, Root-Ordner und Dateien) erstellt.

`MyFS::fuseReaddir`

In der Methode muss für jeden Eintrag im Verzeichnis die übergebene Funktion `filler` aufgerufen werden. Erst wird das für die Einträge `.` und `..` getan. Diese stehen für das aktuelle und das darüberliegende Verzeichnis. Dann wird über alle Dateien im Rootberzeichnis iteriert und deren Namen an den filler übergeben.

`MyFS::fuseGetattr`

Befüllt statbuf mit Attributen aus dem Eintrag des Rootverzeichnis. Wenn die angefragte Datei nicht existiert wird `-ENOENT` zurückgegeben. Bei der Abfrage des Rootverzeichnisses selbst wird das Stat-Struct mit den Werten:

```
st_uid = getuid();        // UID des aktuellen Nutzers
st_gid = getgid();        // GID des aktuellen Nutzers
st_atime = time(NULL);    // alle Zeiten ...
```

```
st_mtime = time(NULL);    // auf den ...
st_ctime = time(NULL);    // aktuellen Zeitstempel
st_mode = S_IFDIR | 0555; // Zugriffsberechtigungen auf 0555
st_nlink = 2;             // Anzahl der Links auf 2
st_size = 4096;           // Größe in Bytes auf 4096
st_blocks = 8;            // Anzahl der 512 Byte Blöcke auf 8
```

befüllt.

`MyFS::fuseOpen`

Öffnet eine Datei zum lesen oder schreiben. Beim Aufruf der Methode wird die Anzahl der offenen Dateien um eins erhöht und das Structfeld `fuse_file_info->fh` wird mit dem Zeiger auf den Ersten Datenblock der Datei beschrieben.

`MyFS::fuseRead`

Diese Methode wird immer direkt nach dem erfolgreichen fuseOpen aufgerufen. Das übergebene Struct `fuse_file_info` beinhaltet den vorher gespeicherten Filehandle. Nun können über die FAT alle Blocknummern abgefragt und ausgelesen. Die gelesenen Bytes werden in den Puffer `buf` kopiert und die Anzahl zurückgegeben.

`MyFS::fuseRelease`

Wird beim schließen einer Datei aufgerufen. Die Anzahl der offenen Dateien wird um eins reduziert und alle `toFile` Methoden der Container-Objekte werden aufgerufen.

3 Read and Write Filesystem

3.1 Erstellen und Löschen von Dateien

3.1.1 Eigene Klassen und Methoden

Für das löschen von Dateien mussten die Klassen `DMAP` und `RootDir` um Methoden erweitert werden.

`void DMAP::setFree(uint16_t pos)`

Setzt einen Block auf “Free”.

`int RootDir::del(const char *name)`

Löscht den Eintrag aus dem Bootverzeichnis.

3.1.2 FUSE-Methoden

`MyFS::fuseMknod`

Wird nur dann aufgerufen wenn die Methode `MyFS::fuseGetattr` den Fehlercode `-ENOENT` zurück gibt. Beim erstellen einer neuen Datei wird der gewünschte Name und die Zugriffsberechtigungen übergeben. Die Restlichen Filestats werden mit den Werten:

```
st_blksize = 512;
st_size = 0;
st_blocks = 0;
st_nlink = 1;
st_atime = time(NULL);
st_mtime = time(NULL);
st_ctime = time(NULL);
st_uid = getgid();
st_gid = getuid();
```

befüllt.

`MyFS::fuseWrite`

Schreibt vom `offset` bis `offset+size` den Inhalt aus dem Puffer `buf` in eine Datei. Dabei wird zuerst ermittelt ob neue Blöcke für den Schreibvorgang nötig sind oder ob die vorhandenen Blöcke überschrieben werden. Wenn neue Blöcke verwendet werden, werden diese in der FAT an die vorhandenen Blöcke angehängt. Nach dem Schreiben werden die Filestats `st_size` und `st_blocks` auf die neue Größe und `st_mtime` und `st_ctime` auf den aktuellen Timestamp angepasst. Sollte der Schreibvorgang nicht möglich sein, weil keinen Platz im Dateisystem vorhanden ist, wird der Fehlercode `-ENOSPC` zurückgegeben.

`MyFS::fuseUnlink`

Entfernt eine Datei indem der Eintrag im Rootverzeichnis gelöscht wird und die entsprechenden Datenblöcke in der DMAP freigegeben werden.

4. Testfälle

4.1 Testen des Containers

Das Testen der Containerdatei, also das Testen des Erzeugens und Befüllens der Containerdatei erfolgt ohne jegliche Abhängigkeit zu FUSE. Dadurch konnte das Testen bereits parallel zum Schreiben des Codes für die anderen Aufgabenstellungen erfolgen. Die Testcases sind mit dem Testframework “Catch” geschrieben.

4.1.2 Test-DMAP

In diesem Testcase werden die Methoden der DMAP-Klasse getestet. Erst die, die einzelne Blöcke zurückgeben und diese allokieren und dann die, die mehrere Blöcke aus der DMAP zurückgeben und allokieren können.

4.1.1 Test-FAT

Diese Tests überprüfen ob alle Lese- und Schreib-Methoden der Fat-Klasse funktionieren.

4.2 Testen der Fuse Operationen

Die Fuse Operationen wurden mit `Bash`-Scripten ausführlich getestet. Dabei werden Linux-Commands ausgeführt und der Exit-Code dieser überprüft:

```
<command>
if [ $? -eq 0 ]; then
    echo -e "command successful"
else
    echo -e "command failed"
fi
```

`read.sh`

Das Skript erstellt eine Containerdatei und übergibt dabei eine Testdatei. Dann wird die Containerdatei gemountet und die Datei über `diff` mit dem Original verglichen.

`write.sh`

Mit diesem Script werden verschiedene Schreiboperationen getestet: - Anlegen einer leeren Datei - Beschreiben der Datei - Text anhängen - Datei überschreiben

`delete.sh`

Dieses Script testet die unlink Methode. Dazu wird eine Datei angelegt und gleich wieder gelöscht.

5. Optimierungen

Beim ersten programmieren der Container-Klassen haben alle Methoden immer direkt die Daten aus der Containerdatei gelesen und geschrieben. Das hatte zufolge, dass Lese und Schreiboperationen bei großen Dateien lange gedauert haben. Um das zu beschleunigen wurden für die Klassen `DMAP`, `FAT` und `RootDir` jeweils ein Array angelegt, welches im Konstruktor aus der Containerdatei befüllt

und durch den Aufruf der `toFile` Methode in die Container zurückgeschrieben wird.

Dadurch konnte die Zeit, die zum Schreiben von 30 MB gebraucht wird um das 42-Fache reduziert werden.