Betriebssysteme Labor

Eigenes Dateisystem mit FUSE

Prof. Dr. rer. nat. Oliver P. Waldhorst

Cédric Uden - 56869

Dominik Schneider - 72374

Felix Schneider - 72306

Marlon Zarnke - 71667

14. Oktober 2020 - 03. Februar 2021

Inhaltsverzeichnis

Einleitung

Ziel

Im Rahmen des Labors zur Vorlesung Betriebssysteme im dritten Semester des Informatik Studiums ist es die Aufgabe der Studenten ein eigenes Dateisystem mit Hilfe der Bibliothek `FUSE` zu konzeptionieren und daraufhin umzusetzen. Ziel dessen ist es die Studenten praxisnah an das systemnahe Programmieren heranzuführen.

Aufgabe

Die Aufgabe wurde in zwei Teile gegliedert:

Im ersten Teil ging es darum erste Einblicke in die Funktionsweise von `FUSE` zu bekommen und den gegebenen Code zu verstehen. Daraufhin sollte die Umsetzung unseres eigenes Dateisystems stattfinden, welches schlussendlich eine RAM-Disk repräsentierte. Allerdings gab es ein Problem dabei: Sobald der Code des `My-In-Memory-FS` gestoppt wurde, zum Beispiel durch absichtliches Beenden des Programms oder auch durch einen Absturz des Programms, sind alle Dateien sofort verfallen ohne ein Backup.

Im zweiten Teil lag der Fokus auf der Persistenz der Dateien. Hier war der Ansatz das Verfallen der Dateien bei Beendung des Programmes zu verhindern, indem wir die Dateien nicht in den Arbeitsspeicher des Systems geschrieben haben, sondern in einen Container, welcher ein normales Speichermedium, wie zum Beispiel eine HDD oder einen USB-Stick, darstellte.

Unser Dateisystem sollte keine Unterordner haben, sondern nur ein `Root`-Verzeichnis. Weiterhin sollte das Dateisystem bis zu 64 Dateien gleichzeitig speichern können.

Ein weiterer Teil der Aufgabenstellung war es diese Dokumentation zu schreiben.

Umgebung

Uns wurde eine virtuelle Maschine auf der Ubuntu läuft bereitgestellt, auf der `FUSE` schon installiert war. Weiterhin erhielten wir Zugang zu einem GitHub-Repository von Prof. Dr. Waldhorst, in dem schon eine Projektvorlage mit Skeletten der zu implementierenden Funktionen lag. Das Vervollständigen dieser war unsere Aufgabe.

FUSE

`FUSE` (Filesystem in Userspace) ist ein Kernel-Modul für Linux, welches auch nicht-privilegierten (nicht `root`) Nutzern ermöglicht eigene Dateisysteme mit dem Kommando `mount` zu verwenden. Dabei läuft das eigene Dateisystem nicht wie üblich im Kernel-Mode, sondern im User-Mode. Dies vereinfacht die Programmierung ungemein, da sich der Programmierer nicht mit Beschränkungen und Besonderheiten des Kernel-Mode auseinandersetzten muss.

Weiteres zu `FUSE` lässt sich hier finden: https://wiki.ubuntuusers.de/FUSE/

My-In-Memory-FS

Dateistruktur

Da wir für jede Datei nicht nur den eigentlichen Inhalt speichern mussten, sondern auch andere wichtige Metainformationen, wie zum Beispiel Besitzer, Rechte und Modifikationszeit, haben wir uns entschieden all diese Informationen für eine Datei zu bündeln und in einer `struct` zu speichern:

```
typedef struct {
    char name[NAME_LENGTH];

    int size = 0;

    uid_t uid;
    gid_t gid;
    mode_t permission = S_IFREG | 0777;

    char* data;

    time_t atime;
    time_t ctime;
    time_t mtime;

} MyFSFileInfo;
```

Alle Dateien, die wir angelegt haben, wurden in einem Array `files` gespeichert mit

```
#define NUM_DIR_ENTRIES 64
files = new MyFSFileInfo [NUM_DIR_ENTRIES];
```

und lagen somit als Instanz unserer `struct` im Arbeitsspeicher.

Umzusetzende Funktionen

Für das Entwickeln eines simplen Dateisystems mit `FUSE` sind nicht alle angebotenen Methoden von `FUSE` notwendig. Für die gestellte Aufgabe wurde vorgegeben, dass wir folgende Methoden zu implementieren haben:

```
fuseInit(struct fuse_conn_info *)

fuseGetattr(const char*, struct stat*)

fuseMknod(const char*, mode_t, dev_t)

fuseUnlink(const char*)

fuseOpen(const char*, struct fuse_file_info*)

fuseRead(const char*, char*, size_t, off_t, struct
fuse_file_info*)

fuseWrite(const char*, const char*, size_t, off_t, struct
fuse_file_info*)

fuseRelease(const char*, struct fuse_file_info*)

fuseReaddir(const char*, void*, fuse_fill_dir_t, off_t, struct
fuse_file_info*)

fuseTruncate(const char*, off_t)

fuseTruncate(const char*, off_t, struct fuse_file_info*)
```

fuseInit

Diese Funktion wird beim Start des Programms aufgerufen. Sie ist dafür zuständig das Logging zu starten für das einfachere Debuggen und jeden Eintrag des Array `files` mit `'\0'` zu initialisieren, um so dem Programm zu signalisieren, dass die Einträge alle leer sind und noch befüllt werden können.

fuseGetattr

Diese Funktion wird aufgerufen, wenn die Metadaten einer Datei benötigt werden, zum Beispiel bei einer Abfrage mit `ls filename`. Die Informationen werden in den übergebenen Puffer `statbuf` übergeben, wenn eine Datei mit dem übergebenen Namen existiert. Andernfalls gibt die Funktion einen Fehler zurück.

fuseMknod

Diese Funktion wird aufgerufen, wenn eine neue Datei angelegt werden soll. Dabei wird eine neue Instanz des `MyFSFileInfo`-Structs erstellt und mit den nötigen Informationen, also dem Namen, dem Inhalt, der UID, der GID und der aktuellen Zeit, befüllt. Dies geschieht allerdings nur, wenn nicht schon eine Datei mit dem gleichen Namen existiert oder und der Name nicht länger als `NAME_LENGTH` mit

```
#define NAME_LENGTH 255
```

ist.

fuseUnlink

Diese Funktion ist für das Löschen der Dateien zuständig. Dabei wird der Verweis des Namens der zu löschenden Datei im Array `files` mit `'\0'` überschrieben und somit wieder als frei verfügbar deklariert. Weiterhin wird der Speicherplatz des Inhalts der Datei mit `free` freigegeben.

fuseOpen

Diese Funktion für das Öffnen einer Datei zuständig. Sie sorgt nicht, wie der Name vermuten lässt, für das Lesen des Inhalts einer Datei, sondern sperrt die Datei. Das heißt bevor ein Prozess eine Änderung an einer Datei vornehmen möchte muss der Prozess erst nach einer Berechtigung dafür fragen. Diese Berechtigung wird nur einem Prozess gleichzeitig gegeben. Dies sorgt dafür, dass nicht mehrere Prozesse gleichzeitig dieselbe Datei beschreiben oder ein Prozess den Inhalt der Datei liest, während ein anderer sie beschreibt. Dafür wird dem Prozess ein `fileHandle`, also ein eindeutiger und einzigartiger Identifikator, gegeben, mit dem er auf die gewünschte Datei zugreifen kann.

fuseRead

Diese Funktion ist für das Lesen des Inhalts einer Datei zuständig. Sie wird also beispielsweise aufgerufen bei einem `cat filename`. Dabei hat sie als Parameter die `size` und den `offset` bekommen, um zu wissen, ab wo sie wie lange lesen muss. Die Funktion hat den resultierenden String, an den in den übergebenen Puffer kopiert und gibt die Anzahl der tatsächlich gelesenen Bytes zurück.

fuseWrite

Wie der Name der Funktion schon vermuten lässt, ist diese Funktion für das Schreiben einer bestehenden Datei mit neuem Inhalt zuständig. Wie bei `fuseRead` wird auch dieser Funktion die `size` und der `offset` übergeben. Zusätzlich wird ihr noch der Inhalt des Puffers übergeben, da in diesem der neue gewünschte Inhalt steht.

Sofern die alte Größe und die neue Größe des Inhalts nicht gleich sind wird die entsprechende Eigenschaft der Datei dem neuen Inhalt entsprechend angepasst.

fuseRelease

Diese Funktion ist analog zu `fuseOpen` für das Schließen einer Datei zuständig. Dabei gibt ein Prozess den `fileHandle`, nachdem er fertig mit seiner Schreib- oder Leseoperation fertig ist, wieder zurück. Somit kann nun die Datei wieder von einem anderen Prozess geöffnet werden.

fuseReaddir

Diese Funktion ist für das Ausgeben der Inhalte eines Ordners verantwortlich. Da unser Dateisystem keine Unterordner unterstützt muss die Funktion nur nach Dateien im aktuellen Verzeichnis suchen. Diese Funktion wird zum Beispiel bei einem `ls directory/` aufgerufen.

Zusätzlich zu den Dateien liefert die Funktionen auch noch die Verweise auf das aktuelle und das übergeordnete Verzeichnis:

```
filler( buf, ".", NULL, 0 );  // Current Directory
filler( buf, "..", NULL, 0 ); // Parent Directory
```

fuseTruncate

Diese Funktion beschäftigt sich mit dem Ändern der Größe einer Datei. Dabei wird der Parameter `size` verändert. Wenn sich die Größe verkleinert wird ein Teil des Inhalts abgeschnitten. Da dieser aber zu diesem Zeitpunkt schon leer ist, gehen keine Daten verloren. Wird die Größe größer wird die der Parameter `size` angepasst, allerdings ist der neue Platz noch nicht mit neuem Inhalt definiert.

My-On-Disk-FS

Bei der zweiten Teilaufgabe ging es darum das volle Potential unsers Dateisystems auszunutzen. Dafür war es notwendig, dass die gespeicherten Dateien persistiert werden und somit länger als bis zu einem Programmstop oder Systemneustart gespeichert bleiben. Um dies zu erreichen mussten wir nicht nur den Inhalt der Dateien und deren Metainformationen speichern, sondern auch wie diese verwaltet werden. Um dies zu ermöglichen haben wir ein vorgegebenes Blockgerät, in der Literatur auch `Blockdevice`, simuliert.

Dateistruktur

DMAP

Die DMAP ist eine Klasse, welche den Überblick über die freien Datenblöcke behält und Auskunft gibt über den Status einzelner Blöcke im Dateisystem. Ein Array beschreibt, ob ein Block `free` ist, also nicht genutzt wird, oder den Status `used` hat, was so viel bedeutet, dass der Block bereits verwendet wird. Dies ist notwendig, da man natürlich nur freie Blöcke verwenden möchte, um neue Information zu speichern.

In unserem Falle der Implementierung der DMAP haben wir ein Array `blocks[DATA_BLOCKS]` mit

```
#define DATA_BLOCKS 4096  // 4096*512KB = 2097152 which
approximates to 2 GB total FS size
```

vom Typ `Boolean` verwendet in welchem die Information über freie Blöcke gespeichert wird. Dies Funktioniert nun wie folgt:

Ist an der Indexstelle `dataBlockNum` der Eintrag `true` so ist der Block an der Stelle `dataBlockNum` frei und nicht `isUsed`. Bei einem Eintrag von `false` wird der Block mit der Nummer `dataBlockNum` bereits benutzt.
Des Weiteren besitzt unsere Klasse, welche sich um die DMAP kümmert, eine Variable

```
int freeBlockCounter
```

welche stehts mitzählt, wie viele freie Blöcke zu einem gegebenen Zeitpunkt existieren.

Wir haben Folgende Methoden implementiert, welche die DMAP bereitstellen:

`initialInitDMap(): Initialisiert die DMAP in einen Container für das Dateisystem`

`initDMap(): Falls es schon eine bestehende DMAP gibt, da das Filesytem nicht leer ist, wird diese Funktion aufgerufen und lädt die Daten des bestehenden Contaiers ein`

`getNextFreeBlock(): Gibt den Index des nächsten freien Blockes zurück`

`getXAmountOfFreeBlocks(int amount): Gibt ein Array mit der Länge amount zurück, welches die Indexe aller freien Datenblöcke beinhaltet`

`setBlockState(int dataBlockNum, bool isUsed): Setzt den Status eines Blockes auf false für free oder true für isUsed`

`getBlockState(int dataBlockNum): Gibt den Status eines Blockes zurück`

`increaseFreeBlockCounterBy(int amount): Inkrementiert den freeBlockCounter um amount`

`decreaseFreeBlockCounterBy(int amount): Dekrementiert den freeBlockCounter um amount`

`persists(): Dient zum Schreiben aller Änderungen auf den Container um diese zu persistieren`

File Allocation Table (FAT)

In der FAT wird verwaltet, in welchen Datenblöcken die Daten einer Datei gespeichert werden. Die Datenblöcke werden als `Integer` Array organisiert. Dabei entsprechen die Indizes der FAT den Datenblöcken und die Werte sind eine Referenz auf den nächsten Block der Datei. Das Ende einer Datei wird mit dem Wert `-1` gekennzeichnet. Dieser Aufbau ist nötig, da eine Datei größer als ein Block sein kann und diese Datei auch nicht zwingend in aufeinanderfolgenden Blöcken gespeichert werden muss. Man benötigt für einen Integer Wert 4 Byte, das heißt unsere FAT für alle 4.096 Blöcke einen Speicherbedarf von 4 x 4.096 Bytes, was 16.384 Bytes oder 32 Blöcken à 512 Byte entspricht. Um bei einer Änderung in der FAT nicht immer die gesamte Struktur zu persistieren, merken wir uns welche Blöcke sich in der FAT geändert haben und schreiben nur diese neu.

Wir haben folgende Methoden implementiert, welche die FAT bereitstellen:

`void insertModifiedBlock(int index): Prüft, ob der Block schon modifiziert wurde, schreibt den Index des geänderten Blocks in modifiedBlocks[] und erhöht den modifiedBlocksCounter`

`void clearModifiedBlocks(): Setzt den modifiedBlocksCounter auf 0 und leert das modifiedBlocks[] Array`

`int setNextBlock(int currentBlock, int nextBlock): Erstellt den Verweis auf den nächsten Block der Datei`

`int getNextBlock(int currentBlock): Gibt den Index des nächsten zu Datei gehörigen Block zurück`

`void freeBlock(int index): Setzt den Wert des Blocks zurück`

`persists(): Dient zum Schreiben aller Änderungen auf den Container um diese zu persistieren`

`void initFAT(): Falls es schon eine bestehende FAT gibt, da das Filesystem nicht leer ist wird diese Funktion aufgerufen`

`void initialInitFAT(): Initialisiert die FAT für ein noch nicht beschriebenes Filesystem.`

RootDir

Das RootDir beinhaltet die Metainformationen von `Files`, welche sich im Dateisystem befinden. Dies haben wir mit dem von `GNU C Library` bereitgestelltem `Struct stat` realisiert.

Zu Metainformationen zählen in diesem Falle der `Mode`, welcher Auskunft gibt wie die Zugriffsberechtigungen der Datei sind, die Blockgröße, die totale Größe in Bytes, Anzahl allokierter 512 Byte Blöcke, Anzahl von `hard links` sowie Zeitpunkt der letzten Änderung, der letzten Statusänderung und des letzten Zugriffes auf die Datei. Desweitern kommen hinzu Gruppen ID und Benutzer ID. Diese Informationen sind wichtig, um Auskunft über Dateien zu erlangen.

Wir haben für die Implementierung des RootDir ein Array `files[NUM_DIR_ENTRIES]` vom Typen `rootfile*` verwendet. In diesem sich die Dateien befinden, welche das `Struct` beinhalten, welches wiederum die Metainformationen zu den entsprechenden Dateien beinhaltet.

Wir haben folgende Methoden implementiert, welche die RootDir Klasse bereitstellt:

```
initialInitRootDir(): Initialisiert das RootDirector für ein
noch nicht beschriebenes Filesystem

initRootDir(): Falls es schon ein bestehendes Rootdirectory
gibt, da das Filesystem nicht leer ist, wird diese Funktion
aufgerufen und lädt die benötigten Files

createFile(const char *path): Ein neues File Objekt erstellt
welches das stat Struct beinhaltet das die nötigen Metadaten
enthält. Anschließend wird es an der nächsten freien Stelle des
files Array eingefügt. Zuletzt wird der existingFilesCounter um
eins inkrementiert

deleteFile(rootFile *file): Löscht das File Objekt und setzt
anschließend den Wert des Arrays an der Stelle auf nullptr.
Anschließend wird der existingFilesCounter um eins
dekrementiert

getFile(const char *path): Gibt das File am gegebenen Pfad
zurück falls die nicht möglich wird nullptr zurück gegeben

getFiles(): Gibt Array zurück welches alle existierenden Files
beinhaltet zurück

load(int index): Gibt das File an der übergebenen Stelle index
aus dem files-Array zurück

persist(rootFile *file): Dient zum schreiben aller Änderungen
auf den Container um diese zu persistieren
```

Funktionen

Die Funktionen `fuseRead`, `fuseWrite` und `fuseTruncate` mussten wir anpassen, da diese die Funktionen sind, die direkt mit den Dateien interagieren. Somit müssen sie mit der neuen Dateistruktur zurechtkommen und arbeiten.

`getNextFreeIndexOpenFiles()`: Gibt den Index des nächsten freien Eintrags in dem openFiles-Array zurück

`fuseRead(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fileInfo)`: Diese Funktion analysiert die zu lesenden Datenstücke und gibt die Information an die readFile Funktion weiter

`fuseWrite(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fileInfo)`: Analog zur Funktion fuseRead analysiert fuseWrite die zu lesenden Datenstücke und gibt die Information an die writeFile Funktion weiter

`fuseTruncate(const char *path, off_t newSize)`: Die Funktion analysiert die zu bearbeitenden Datenstücke: Sie löscht die nicht mehr gebrauchten Daten und gibt gegebenenfalls neue Daten an writeFile weiter

`readFile(int *blocks, int blockCount, int offset, size_t size, char *buf, openFile *file)`: Diese Funktion bekommt ihre Informationen von fuseRead und liest daraufhin aus dem Blockdevice und somit mit den einzelnen Blöcken

`writeFile(int *blocks, int blockCount, int offset, size_t size, char *buf, openFile *file)`: Analog zu readFile bekommt auch diese Funktion ihre Informationen aus fuseWrite und schreibt die gewünschten Daten auf das Blockdevice

Tests

Das Testen lässt sich unterteilen in die Tests für die beiden Teilaufgaben:

My-In-Memory-FS

Das Testen unseres Dateisystems erfolgte über Unit-Tests, die uns davor schon in der Datei `./testing/itest.cpp` gegeben waren. Diese prüften unser Dateisystems auf die wichtigsten Funktionen. Genauer wurde folgendes geprüft:

```
Testcase 1.1:  Create & remove a single file
Testcase 1.2:  Write and read a file
Testcase 1.3:  Overwrite a part of a file
Testcase 1.4:  Append at the end of a file
Testcase 1.5:  Append beyond the end of a file
Testcase 1.6:  Append before the end of a file
Testcase 1.7:  Truncate a file
Testcase 1.8:  Change mode of a file
Testcase 1.9:  Write to multiple files
Testcase 1.10: Write a very large file
```

My-On-Disk-FS

Bei der zweiten Teilaufgabe war die Testsituation etwas anders: Uns wurde gesagt, dass unser Programm getestet wird, aber was genau war uns vor der Abgabe nicht klar. Somit mussten wir unser Dateisystem selbst testen, um Bugs zu finden und diese zu beheben.

Fazit

Uns ist es gelungen ein simples Dateisystem mit Hilfe von `FUSE` in `C++` zu implementieren. Dabei konnten wir alle geforderten Strukturen und Funktionsweisen aus der Aufgabenstellung und aus dem "üblichen" Verhalten eines Dateisystems einhalten.

Unser eigenes Dateisystem ist allerdings nicht alltagstauglich. Ganz vorne mit dabei, wenn es um nicht gewährleistete Alltagsfähigkeit geht, ist die fehlende Möglichkeit Unterverzeichnisse anzulegen, um bestehende Dateien besser zu untergliedern. Zusätzlich wäre es wünschenswert mehr als 64 Dateien gleichzeitig speichern zu können. Weiterhin wäre es praktisch die Größe des Dateisystems (dynamisch) zu erhöhen, wohl möglich über einen Parameter beim Start des Programms. Ein weiterer Kritikpunkt unseres Dateisystems ist die Performance, welche wir bei unserer Implementierung vernachlässigt haben. Bei Vergrößerung des Dateisystems oder bei Vergrößerung der Anzahl der möglichen Dateien wäre es erstrebenswert die Geschwindigkeit zu verbessern und für eine erhöhte Stabilität zu sorgen. Wenn es keine festen Vorgaben gibt, würde sich hier die Implementierung eines Superblocks anbieten.

Abschließend kann man sagen, dass uns dieses Labor einen sehr interessanten Einblick in das systemnahe Programmieren mit `C++` gegeben hat. Allerdings war das Labor sehr anspruchsvoll und so zeitintensiv, dass wir uns, zusätzlich zu den Laborterminen, noch an vielen Samstagen treffen mussten, um das Labor zum geforderten Zeitpunkt fertig zu stellen.