



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Labor „Betriebssysteme Übung“

– Wintersemester 2019/20 –

Oliver P. Waldhorst

Zielsetzung

Zielsetzung des Labors

- Ergänzung / Vertiefung der Inhalte der Vorlesung „Betriebssysteme“
 - Insbesondere Funktionsweise von Dateisystemen und deren Verwendung in Linux (UNIX)
- Vertiefung des (Betriebs-)systemnahen Programmierens unter C++
- Entwickeln von Software im Team

Organisatorisches (1)

Umfang

- 3 ECTS / 2 SWS (entspricht einem Arbeitsaufwand pro Person von 90h!)
- Gruppenarbeit in Teams von 3 bis 4 Studierenden

Veranstaltungen

- Jeweils mittwochs von 11:30 – 13:00 und 14.00 – 17.10 Uhr (Li137)

Zeitplan

- Teil 1: 5 Termine (02.10.19 – 30.10.19)
- Teil 2: 10 Termine (06.11.19 – 22.01.20)
- **Letzte Möglichkeit zur Abgabe ist Mittwoch, 22.01.20!**

Organisatorisches (2)

Bewertung

- Unbenoteter Schein
- Bei Erledigung in diesem Semester: Eine Notenstufe Bonus in Betriebssysteme-Klausur

Melden Sie sich für den ILIAS-Kurs „Betriebssysteme Übung“ an

- **Anmeldeschluss für Kurs ist der 30.10.2019**
- Passwort „OS-LAB“
- Bitte spätestens zur Abgabe der Aufgaben Teams anmelden!

Organisatorisches (3)

Wichtige Inhalte, die vorausgesetzt werden, sind als „Warmup“ zusammengestellt

- <https://iz-gitlab-01.hs-karlsruhe.de/IWI-I/bslab-warmup>
(aus dem Hochschulnetz erreichbar)

Es wird ein begleitendes Tutorium angeboten

- Aufarbeitung von Grundlagen, Hilfestellungen zur Projektorganisation, allgemeine Fragen
- Dort wird auch der Warmup behandelt
- Mittwoch, 6. Block, LI137
- Start am 09.10.2019

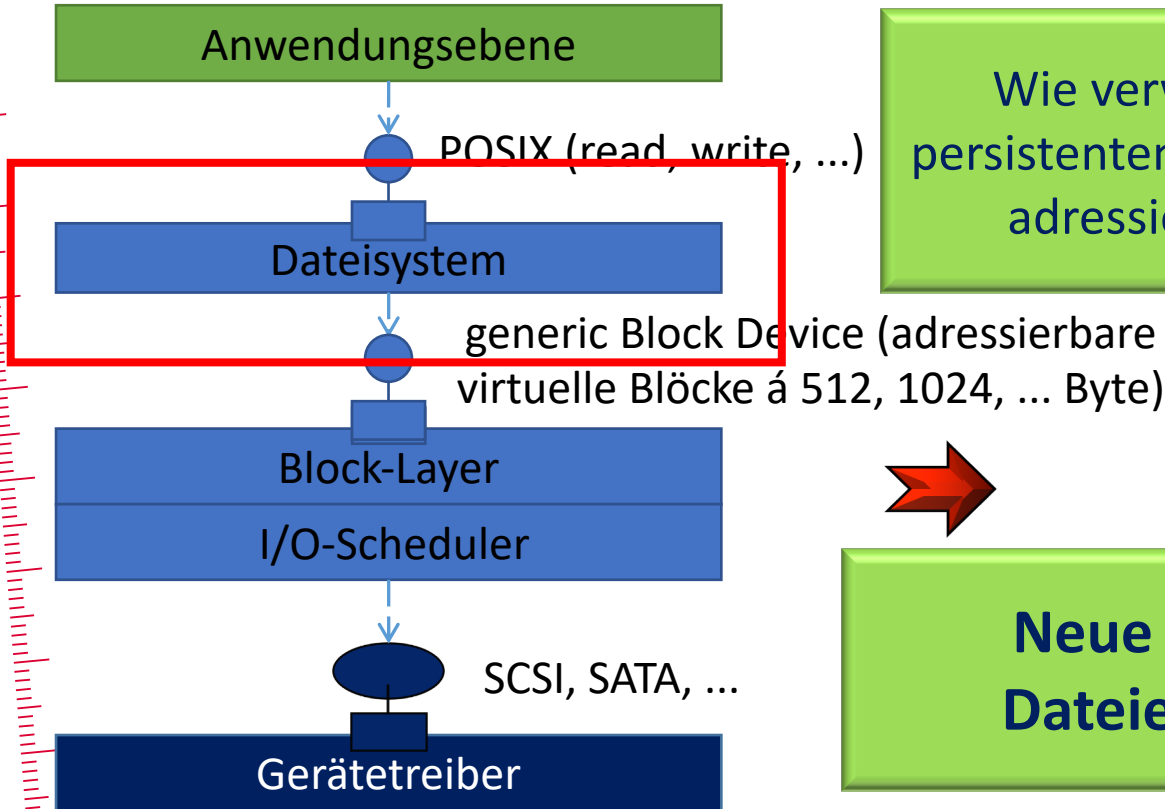
Konkreter Inhalt des Labors

Erstellt werden soll ein Dateisystem *MyFS*

- Wird verwendet, um Datenträger zu „formatieren“
 - Unterstützt Dateien mit den „üblichen“ Attributen (Name, Größe, Zugriffsrechte, Zeitstempel...)
 - Dateien sind in einem einzigen Verzeichnis angeordnet (d.h. es gibt keine Unterverzeichnisse)
- Ein mit MyFS formatierter Datenträger kann (wie jeder Datenträger mit einem bekannten Dateisystem) in den Verzeichnisbaum eingebunden werden
 - Ort der Einbindung ist ein frei wählbares, leeres Verzeichnis
 - Der Inhalt des Datenträgers erscheint in diesem Verzeichnis
- Wie soll das erreicht werden?



Details zur Aufgabenstellung

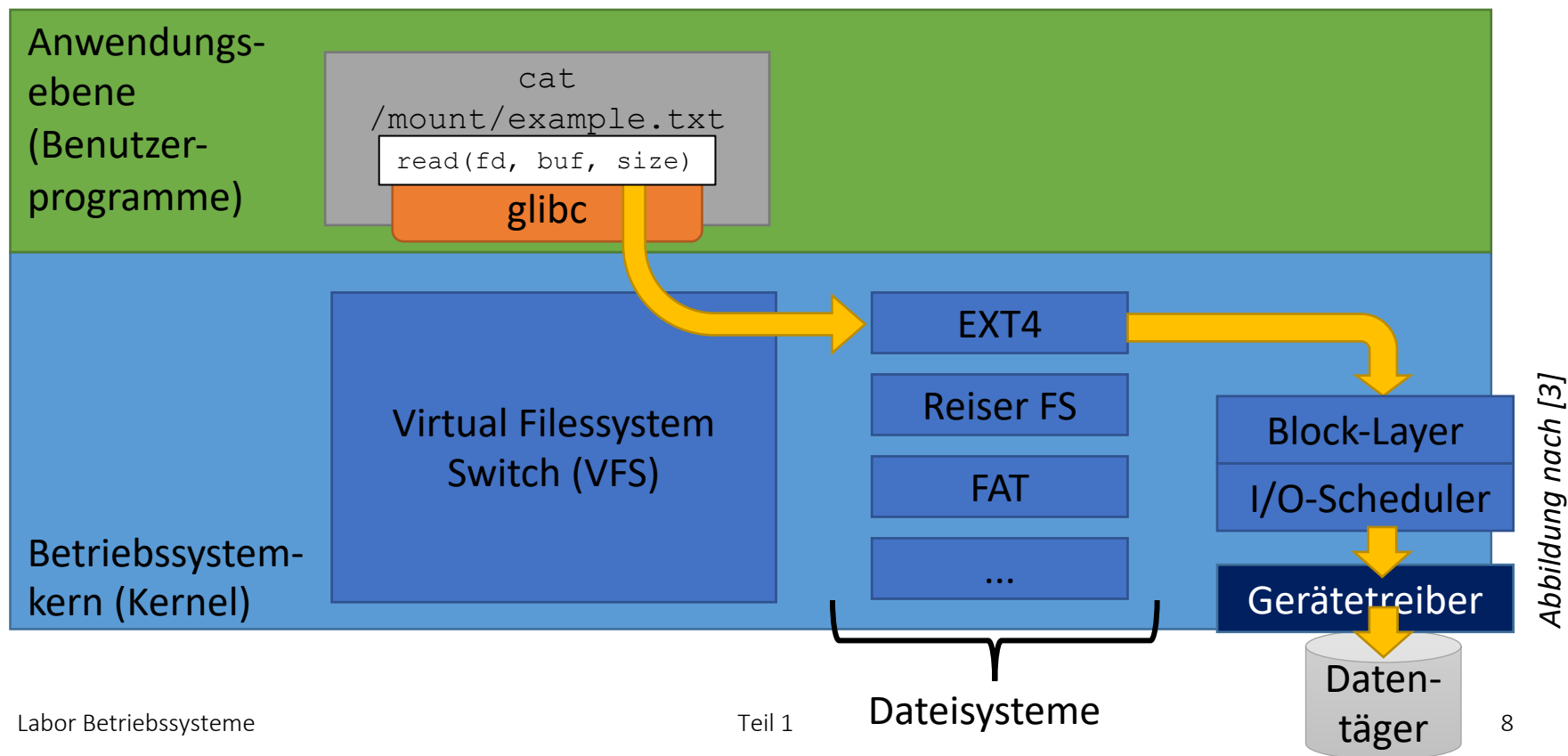


Wie verwaltet und organisiert man persistenten Speicher (, der aus einzelnen, adressierbaren Blöcken besteht)?



**Neue Abstraktionsebene:
Dateien und Verzeichnisse**

„Echte“ Dateisysteme erfordern Kernel-Programmierung!



Eine einfachere Alternative: *File System In User Space (FUSE)* <https://github.com/libfuse/libfuse/>

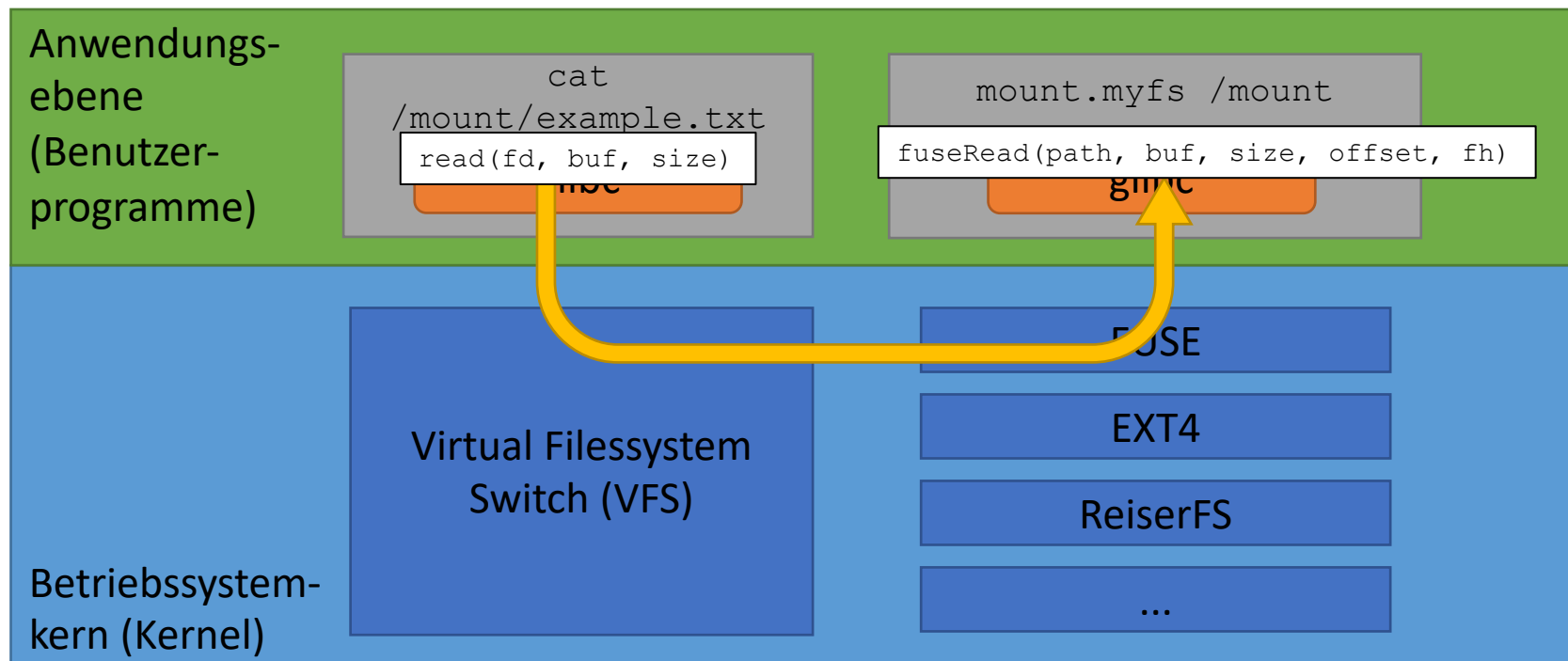


Abbildung nach [3]

Wie schreibt man ein FUSE-Dateisystem [3]

- ... eigentlich wie ein „normales“ C-Programm
- Das Programm muss bestimmte Funktionen / Methoden bereitstellen, die dann von FUSE aufgerufen werden
 - „Callbacks“, potentiell rund 35 Operationen
 - Keine Sorge, es müssen nicht für alle Operationen Funktionen definiert werden!
 - Sinnvolle Dateisysteme kommen bereits mit wenigen Operationen aus (s.u.)

Operationen in FUSE [3]

FUSE-Operationen, für die Funktionen definiert werden können, gliedern sich wie folgt:

- Verzeichnisoperationen
- Dateioperationen
- Operationen auf Metadaten
- Sonstige Operationen

Verzeichnisoperationen

`readdir(path)`

- Liefern der Verzeichniseinträge für jede Datei in einem Verzeichnis (inkl. dem Verzeichnis selber „.“ und dem übergeordneten Verzeichnis „..“)

`mkdir(path, mode)`

- Verzeichnis erzeugen

`rmdir(path)`

- Verzeichnis löschen

Dateioperationen (1)

`mknod(path, mode, dev)`

- Erzeugen einer Datei

`unlink(path)`

- Löschen einer Datei

`rename(old, new)`

- Datei verschieben oder umbenennen

`open(path, flags)`

- Datei öffnen

`read(path, buf, length, offset, fh)`

- Daten aus Datei lesen

Dateioperationen (2)

`write(path, buf, size, offset, fh)`

- Daten in Datei schreiben

`truncate(path, len, fh)`

- Datei an Stellen `len` abschneiden

`flush(path, fh)`

- Datei zurückschreiben

`release(path, fh)`

- Datei (endgültig) schließen

Operationen auf Metadaten

`getattr(path)`

- Metadaten für Datei lesen

`chmod(path, mode)`

- Zugriffsrechte setzen

`chown(path, uid, gid)`

- Besitzer ändern

`fsinit(self)`

- Datenstrukturen initialisieren, ...

Andere Operationen siehe

https://libfuse.github.io/doxygen/structfuse__operations.html

Sinnvolle Fehlercodes bei der Rückgabe (vgl. **errno.h**)

Von FUSE aufgerufene Methoden geben i.d.R. zurück:

- Erfolg: Rückgabe ≥ 0
- Fehler: -(Fehlercode)

ENOSYS	Funktion nicht implementiert
EROFS	Nur lesbares Dateisystem
EPERM	Operation nicht erlaubt
EACCES	Zugriff verweigert
ENOENT	Datei oder Verzeichnis existiert nicht
EIO	I/O Fehler
EEXIST	Datei existiert
ENOTDIR	Datei ist kein Verzeichnis
EISDIR	Datei ist ein Verzeichnis
ENOTEMPTY	Verzeichnis ist nicht leer

Definition von FUSE-Funktionen

Funktionen werden in C definiert und Zeiger auf die Funktionen in einer Struktur vom Typ `fuse_operations` an FUSE übergeben

```
struct fuse_operations {  
    int (*getattr) (const char *, struct stat *);  
    int (*readlink) (const char *, char *, size_t);  
    int (*getdir) (const char *, fuse_dirh_t,  
        fuse_dirfil_t);  
    int (*mknod) (const char *, mode_t, dev_t);  
    int (*mkdir) (const char *, mode_t);  
    ...  
};
```

(Vgl. https://libfuse.github.io/doxygen/structfuse_operations.html)

Beispiele für FUSE-Dateisysteme

SSFS: Simple & Stupid File System [4]

- Liefert Verzeichnis mit genau zwei Dateien und deren Inhalt

BBFS: Big Brother File System [5]

- Erlaubt Zugriff per FUSE auf „normales“ Verzeichnis
- Gibt bei jedem Zugriff die verwendeten FUSE-Operationen aus
- Interessant, um herauszufinden, was eigentlich genau passiert!

FUSE und C++ [6]

Zur Verwendung in C++ müssen C-Funktionen als Wrapper geschrieben werden

Beispiel: Klasse MyFS (myfs.h/myfs.cpp) mit Wrapper (wrap.h/wrap.cpp)

In myfs.h:

```
class MyFS {  
private:  
    static MyFS *_instance;  
    ...  
public:  
    static MyFS *Instance();  
    ...  
    // --- Methods called by FUSE ---  
    int fuseGetattr(const char *path, struct stat *statbuf);  
    ...  
};
```

In wrap.cpp:

```
int wrap_getattr(const char *path, struct stat *statbuf) {  
    return MyFS::Instance()->fuseGetattr(path, statbuf);  
}
```

Die Aufgabenstellung

Aufgabe Teil 0: Verstehen, was FUSE tut!

Aufgabe Teil 1: In-Memory File System (RAM-Disk)

- Beim Starten des Programms `mount .myfs` wird ein leeres Dateisystem erstellt
- Dateien, die in dem Dateisystem abgelegt werden, werden ausschließlich im Arbeitsspeicher des Rechners gehalten
- Nach Programmende, -absturz, Neustart des Rechners etc. sind die Dateien nicht mehr vorhanden

Aufgabe Teil 2: On-Disk File System

- Das von `mount .myfs` verwaltete Dateisystem wird auf einen Datenträger geschrieben.
- Nach Programmende und Neustart sind alle Dateien noch vorhanden
- Mehr dazu in der Vorlesung Betriebssysteme

Aufgabe Teil 3: Dokumentation

Teil 0: Verstehen, was FUSE tut

- Richten Sie Ihre Arbeitsumgebung ein
- Klonen Sie das Projekt-Template, übersetzen Sie es und führen Sie es aus (siehe auch README.md im Projekt)
- Versuchen Sie zu verstehen, was die folgenden Funktionen tun:
 - `MyFS::fuseGetattr()`
 - `MyFS::fuseRead()`
 - `MyFS::fuseReaddir()`

Teil 1: Read-Only File System

Teilaufgabe 1a: Design: Definition von Datenstrukturen und Software-Architektur ihrer Lösung

Teilaufgabe 1b: Umsetzen der FUSE-Funktionen für `mount.myfs`

Teilaufgabe 1c: Ausführliches Testen!

Die Aufgabe 1a sollte von der gesamten Gruppe bearbeitet werden. Für die übrigen Aufgaben empfiehlt sich eine Aufteilung!

1a: Definition von Datenstrukturen

Für MyFS-Dateien sollte (mindestens) folgendes gespeichert werden:

- Dateiname (max. Länge NAME_LENGTH)
- Dateigröße
- Benutzer / Gruppen-ID
- Zugriffsberechtigungen (mode)
- Zeitpunkt letzter Zugriff (atime) / letzte Veränderung (mtime) / letzte Statusänderung (ctime)
- Daten der Datei

In einem MyFS-Dateisystem können NUM_DIR_ENTRIES Dateien gespeichert werden

Hinweise zum Speichern der Daten (1)

Dateien sind im allgemeinen eine Folge von Byte fester Länge

- Beliebige Bytes, auch "`\0`"!
- Unterschied zu Strings (z.B. Dateiname) beachten – diese enden mit "`\0`"!
- Es bietet sich an, Daten in einem Array vom Typ `char` zu speichern:

```
struct MyFsFileInfo {  
    char name[NAME_LENGTH];  
    size_t size;  
    char * data;  
    ...  
}
```


Hinweise zum Speichern der Daten (2)

Speicher kann mit `malloc()` reserviert werden:

```
MyFsFileInfo files[NUM_DIR_ENTRIES];  
files[1].size= 1024;  
files[1].data= malloc(files[1].size);
```

Speicher kann mit `free()` wieder freigegeben werden:

```
free(files[1].data);  
files[1].size= 0;
```

Hinweise zum Speichern der Daten (3)

Benötigen Sie mehr Speicher, können sie `realloc()` verwenden:

```
files[1].size= 2048;  
files[1].data= realloc(files[1].data,  
                        files[1].size);
```

Zum Kopieren von Speicher kann `memcpy()` verwendet werden:

```
char buffer[256], buffer2[256];  
memcpy(buffer, files[1].data, 256);  
...  
memcpy(buffer, files[1].data + 256, 256);
```

Hinweise zum Umgang mit Dateinamen

FUSE übergibt alle Dateinamen in `path` mit führendem `" / "`

- Um diesen zu ignorieren, kann `path+1` als Zeiger auf den Dateinamen verwendet werden, z.B. `printf("%s", path+1);`

Ein String kann mit `strcpy()` in einen anderen kopiert werden

```
char path= "/path";  
strcpy(files[1].name, path+1);
```

Ein String kann mit `strcmp()` mit einem anderen verglichen werden

```
if(strcmp(files[1].name, path+1) == 0) {  
    // Strings sind gleich  
}
```

1b: Umsetzen der FUSE-Funktionen

MyFS-Datenträger sollen mit folgendem Kommando eingebunden werden:

```
mount.myfs mount_point -l logdatei
```

Dabei ist

- *mount-dir* Das Verzeichnis, in das der Datenträger eingebunden werden soll
- *logdatei* Dateiname einer Datei, in die Log-Meldungen geschrieben werden, wenn FUSE Operationen auf MyFS ausführt (hilfreich, da FUSE im Hintergrund arbeitet)

Beispiel:

```
mount.myfs mount -l log.txt
```

Die Einbindung kann mit dem Kommando `fusermount --unmount mount` (Linux) bzw. `umount mount` (Max OS-X) gelöst werden

Hinweise zu 1b

Folgende Operationen in der Klasse `MyFS` müssen (mindestens) implementiert werden:

- Zum Initialisieren / Freigeben eines Dateisystems

```
MyFS::fuseInit()
```

```
MyFS::fuseDestroy() (*)
```

- Zum Anzeigen eines Verzeichnisses

```
MyFS::fuseOpendir() (**)
```

```
MyFS::fuseReaddir()
```

```
MyFS::fuseReleasedir() (**)
```

```
MyFS::fuseGetattr()
```

- Zum Lesen/Schreiben einer Datei

```
MyFS::Mknod()
```

```
MyFS::Truncate()
```

```
MyFS::fuseOpen()
```

```
MyFS::fuseRead()
```

```
MyFS::fuseWrite()
```

```
MyFS::fuseRelease()
```

```
MyFS::fuseUnlink()
```

() ggf. nicht notwendig*

*(**) Da wir nur ein einziges Verzeichnis pro Containerdatei verwenden, kann dieses auch beim Initialisieren / Freigeben der Containerdatei geöffnet bzw. geschlossen werden*

Hinweise zu FUSE Operationen (1)

```
int MyFS::fuseReaddir(const char *path,  
    void *buf, fuse_fill_dir_t filler, off_t  
    offset, struct Fuse_File_info *fileInfo)
```

- Liefert Namen der Dateien im Verzeichnis zurück
- `filler` wird zum Befüllen der Verzeichniseinträge verwendet
 - Einträge für das aktuelle und übergeordnete Verzeichnis werden erzeugt mit:

```
filler( buf, ".", NULL, 0 );  
filler( buf, "..", NULL, 0 );
```
 - Z.B. wenn sich `file1.txt` im Verzeichnis befindet:

```
filler( buf, "file1.txt", NULL, 0 );
```
- Sinnvolle Fehlercodes
 - `ENOTDIR` – Funktion wurde nicht für das Root-Verzeichnis aufgerufen

Hinweise zu FUSE Operationen (2)

```
int MyFS::fuseGetattr(const char *path,  
                      struct stat *statbuf)
```

- Struktur `statbuf` (vgl. `man 2 stat`) kann mit Attributen aus dem Verzeichniseintrag der Datei befüllt werden
- Füllen Sie dabei die Zugriffsberechtigungen in `statbuf->st_mode` für das Verzeichnis „/“ mit `S_IFDIR | 0755`
- Geben Sie für die Anzahl von Links `statbuf->st_nlink` für das Verzeichnis „/“ den Wert „2“ zurück (<http://unix.stackexchange.com/a/101536>)
- Geben Sie für die Anzahl von Links `statbuf->st_nlink` für alle Dateien im Verzeichnis den Wert „1“ zurück
- Dateinamen werden in `path` mit „/“ am Anfang übergeben – ggf. beachten beim Durchsuchen des Verzeichnisses!
- Sinnvolle Fehlercodes
 - `ENOENT` – Dateien nicht gefunden

Hinweise zu FUSE Operationen (3)

```
int MyFS::fuseOpen(const char *path,  
                    struct fuse_file_info *fileInfo)  
int MyFS::fuseRead(const char *path,  
                    char *buf, size_t size, off_t offset,  
                    struct fuse_file_info *fileInfo)  
int MyFS::fuseRelease(const char *path,  
                       struct fuse_file_info *fileInfo)
```

- In `fileInfo->fh` kann in `fuseOpen()` ein File Handle gespeichert werden, mit dem Sie später in `fuseRead()` und `fuseRelease()` auf die (geöffnete) Datei zugreifen können
- Sinnvolle Fehlercodes
 - `EMFILE` – zu viele geöffnete Dateien
 - `ENOENT` – Datei nicht gefunden
 - `EBADF` – Wert in `fileInfo->fh` zeigt nicht auf eine geöffnete Datei

Hinweise zu den FUSE Operationen (4)

```
int MyFS::fuseMknod(const char *path,  
                    mode_t mode, dev_t dev)
```

- Erzeugt eine neue Datei `path`
- Wird nur aufgerufen, wenn `MyFS::fuseGetattr()` für `path` zuvor `-ENOENT` zurückgeliefert hat!
- Sinnvolle Fehlercodes
 - `EEXIST` – Datei existiert bereits
 - `ENOSPC` – Kein freier Platz im Dateisystem

Hinweise zu den FUSE Operationen (5)

```
int MyFS::fuseWrite(const char *path,  
                    const char *buf, size_t size, off_t offset,  
                    struct fuse_file_info *fileInfo)
```

- Schreibt `size` Bytes aus `buf` ab Position `offset` in die durch `path` oder `fileInfo` gegebene Datei
 - Datei wird vorher geöffnet, daher bietet sich die Verwendung von `fileInfo` an
- Rückgabewert ist die Anzahl der geschriebenen Bytes
- Sinnvolle Fehlercodes
 - `EBADF` – Datei nicht zum Schreiben geöffnet
 - `ENOSPC` – Kein freier Platz in Container-Datei

Hinweise zu den FUSE Operationen (6)

```
int MyFS::fuseTruncate(const char *path,  
off_t newSize)
```

- Setzt neue Länge `newSize` der Datei
- Ggf. wird die Datei abgeschnitten oder mit zufälligen Bytes aufgefüllt
- Rückgabewert bei Erfolg ist 0.
- Sinnvolle Fehlercodes
 - `ENOENT` – Datei nicht gefunden

Hinweise zur Logdatei

Es sind bereits drei Makros zum Schreiben in die Logdatei vorgegeben:

- `LOGM ()`
 - Schreibt den Namen der aktuellen Methode
- `LOG ("Text")`
 - Schreibt den Text
- `LOGF ("Form. Text", ...)`
 - Schreibt den Text mit Formatierungen wie `printf()`
- Die Logdatei kann mittels `tail -f logdatei` kontinuierlich ausgegeben werden

1c: Ausführliches Testen

Entwerfen Sie Testfälle und führen Sie diese aus

- Ein eingebundener MyFS-Datenträger kann über alle Programme getestet werden, die auf das Dateisystem zugreifen
 - Shell-Kommandos (`ls`, `cat`, ...), Text-Editoren, eigenen Programme in C/C++, Java(-script), Python, ...
- Es empfiehlt sich aber, die Funktionen der Klasse `MyFS` zunächst unabhängig von FUSE zu testen (s.u.)

Vorgaben (1)

Konstanten

- `#define NAME_LENGTH 255`
 - Max. Länge eines Dateinamens
- `#define BLOCK_SIZE 512`
 - Logische Blockgröße (wird für Teil 2 benötigt)
- `#define NUM_DIR_ENTRIES 64`
 - Anzahl der Verzeichniseinträge
- `#define NUM_OPEN_FILES 64`
 - Anzahl offener Dateien pro MyFS-Dateisystem

Vorgaben (2)

- Ein Makefile-Projekt für das Ziel `mount.myfs` wird über den Gitlab-Server der Fakultät bereitgestellt
 - <https://iz-gitlab-01.hs-karlsruhe.de/IWI-I/bslab>
 - Enthält bereits die C++ Klasse `MyFS` inkl. Wrapper-Funktionen
 - **Implementieren Sie alle Funktionalität Ihres Dateisystems in dieser Klasse!**
 - Verändern Sie keine Dateien, die beginnen mit:
`// DO NOT EDIT THIS FILE!!!`
 - Hinweise dazu, wo etwas zu implementieren ist stehen hinter:
`// TODO: ...`
 - Implementieren Sie alle benötigten Operationen in C++
 - Sie können das Projekt in IDE Ihrer Wahl importieren
 - ... wir unterstützen CLion, Xcode, (Eclipse)

Vorgaben (3)

- Für das Testen der Klasse `MyFS` können Unit Tests in die Datei `test-myfs.cpp` im Unterverzeichnis `unittests` implementiert werden
 - `test-blockdevice.cpp` zeigt beispielhaft, wie Unit Tests mit Hilfe des Frameworks *catch* (<https://github.com/catchorg/Catch2>) geschrieben werden können – dabei wird die Klasse `BlockDevice` (vgl. Teil 2) als Beispiel verwendet

Vorgaben (4)

Verwenden Sie Git für die Versionskontrolle

- Sie können den Gitlab-Server der Fakultät verwenden (<https://iz-gitlab-01.hs-karlsruhe.de>)
 - Dort kann ein Team-Mitglied ein Projekt anlegen und unter „Settings – Members“ die übrigen Team-Mitglieder einladen
 - Das Template kann wie in der Projektdokumentation beschrieben in ein eigenes Projekt importiert werden

Bitte keine öffentlichen Projekte!

Bewertung

Teil 1 + 2 (je 10 Punkte)

- Testfälle (8x 1 Punkt)
- Erklärung des Codes, Einhaltung von Vorgaben, ... (2 Punkte)
- **Auch wenn Teil 1 und 2 zusammen abgegeben werden können, wird frühe Abgabe von Teil 1 empfohlen!**

Teil 3 / Dokumentation – 10 Seiten (10 Punkte)

- Vollständigkeit und Verständlichkeit der Dokumentation
 - Aufgabenstellung (in eigenen Worten)
 - Lösungsansatz und Umsetzung
 - Programmausführung und Testfälle
 - ...

Das Labor ist bestanden, wenn...

- ... alle drei Teile jeweils mit mindestens 5 Punkten bewertet wurden und...
- ... insgesamt 20 Punkte erreicht wurden

Fragen?

Literatur

- [1] R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, (V. 0.90). Arpaci-Dusseau Books, 2015. <http://pages.cs.wisc.edu/~remzi/OSTEP/> (Kapitel 39 und 40).
- [2] R. Stevens, S. Rago, Advanced Programming the UNIX Environment (3rd Edition). Addison Wesley, 2013. (Kapitel 3 und 4)
- [3] X. Pretzer, Building File Systems with FUSE. <https://stuff.mit.edu/iap/2009/fuse/fuse.ppt> (abgerufen 06.11.2017)
- [4] M. Q. Hussain, Writing a Simple Filesystem Using FUSE in C. <http://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/> (abgerufen 06.11.2017)
- [5] J. Pfeiffer, Writing a FUSE Filesystem: a Tutorial. <https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/> (abgerufen 12.10.2017)
- [6] fuse-examples. <https://code.google.com/archive/p/fuse-examples/> (abgerufen 06.10.2017)
- [7] libfuse API documentation. <https://libfuse.github.io/doxygen/index.html> (abgerufen 06.11.2017)