Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Labor Betriebssysteme

– Sommersemester 2019 –

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 – 18.30 Uhr (Li137)

Zeitplan
- Warmup 27.03.19 – 10.04.19 (erledigt)
- Teil 1: 5 Termine (17.04.19 – 22.05.19)
- Teil 2: 5 Termine (29.05.19 – 03.07.19)
- **Letzte Möglichkeit zur Abgabe ist Mittwoch, 03.07.19!**

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 24.04.2019**
- Passwort „OS-LAB“
- Bitte spätestens zu den Abgabe der Aufgaben Teams anmelden!

Konkreter Inhalt des Labors

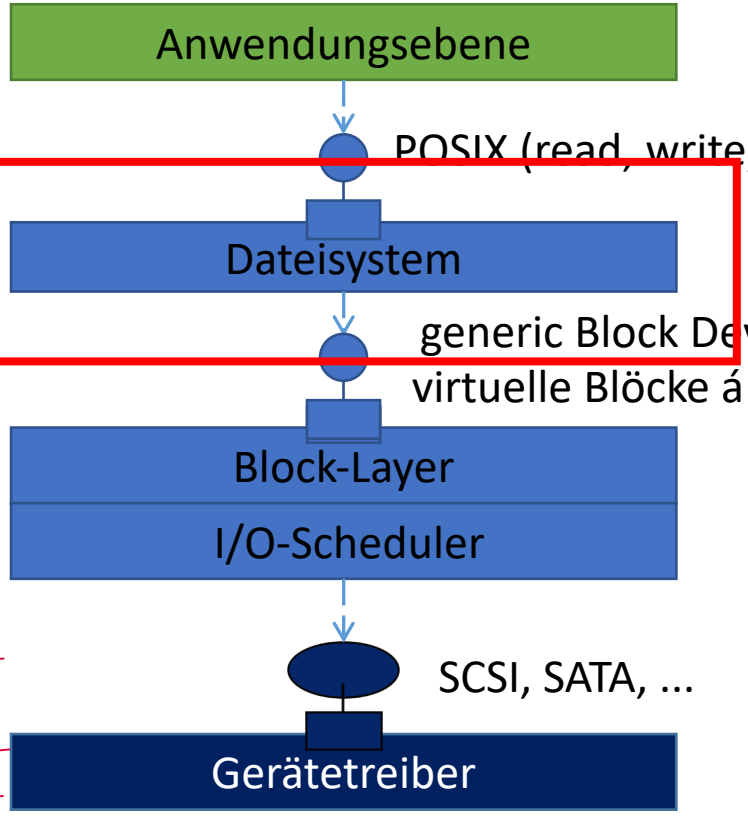
Erstellt werden soll ein Dateisystem *MyFS*

- 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)

- Eine 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!

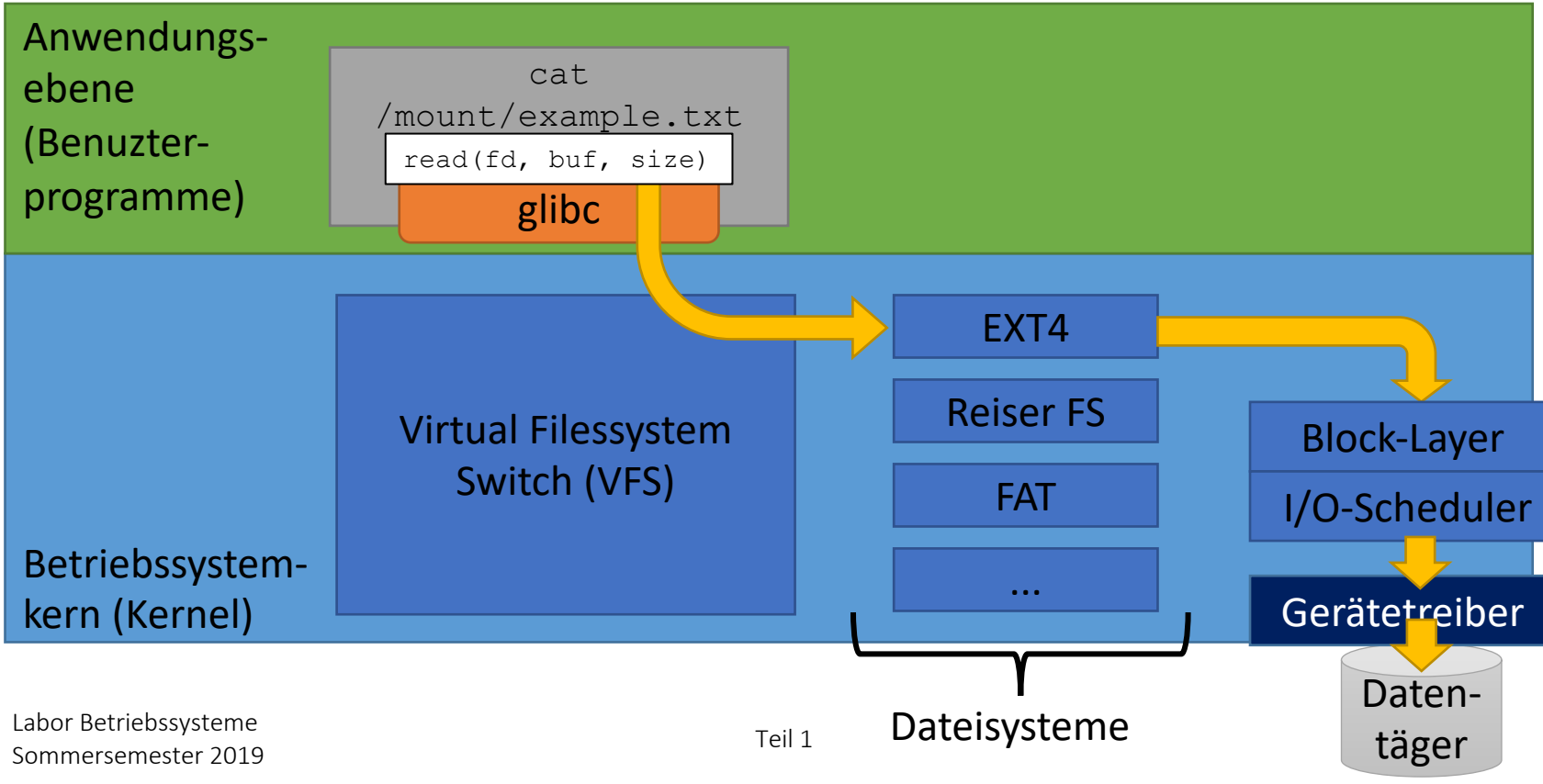


Abbildung nach [3]

Eine einfachere Alternative: *File System In User Space* (*FUSE*)

https://github.com/libfuse/libfuse/

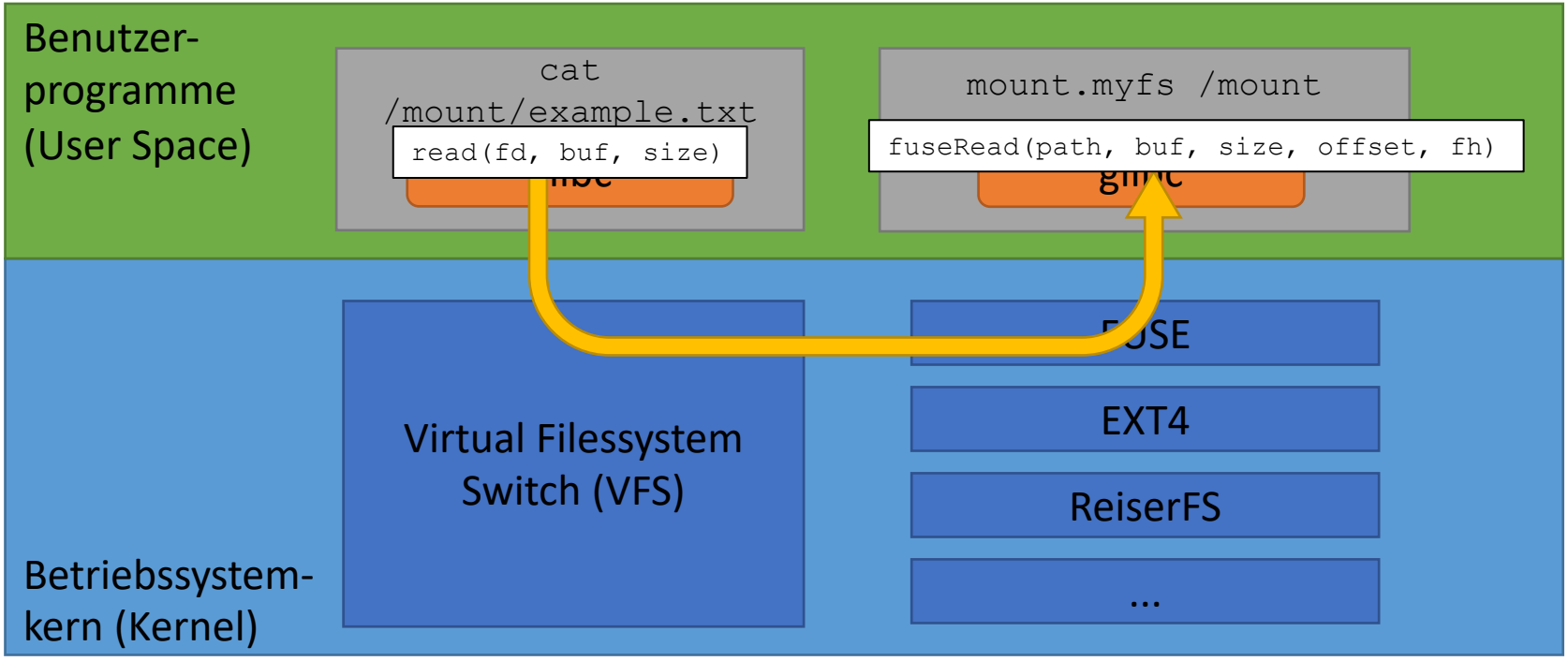


Abbildung nach [3]

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
  - 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

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

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);}
```

Woher wir den Datenträger?

Gar nicht! Datenträger wird simuliert durch Klasse `BlockDevice`

- Verwendet Containerdatei: Inhalt des Datenträgers wird in einer (binären) Datei im herkömmlichen Dateisystem gespeichert

- Bereitgestellte Methoden:
  - `BlockDevice::create(path)`
    – neuen Datenträger / Containerdatei erzeugen
  - `BlockDevice::open(path)`
    – existierenden Datenträger / Containerdatei öffnen
  - `BlockDevice::read(blockNo, buffer)`
    – Block mit Nummer `blockNo` lesen
  - `BlockDevice::write(blockNo, buffer)`
    – Block mit Nummer `blockNo` schreiben
  - `BlockDevice::close()`
    – Datenträger schließen

- **Blöcke haben immer eine feste Größe (hier: 512 Byte)**

Die Aufgabenstellung

Annahme: Wir gehen davon aus, dass wir virtuelle Datenträger in Form von Containerdateien verwenden, auf die wir nur mittels der Klasse `BlockDevice` in Blöcken fester Größe zugreifen können

Aufgabe Teil 0: Verstehen, was FUSE tut!

Aufgabe Teil 1: Read-Only File System

- Eine Datenträger wird mittels eines Kommandos `mkfs.myfs` erstellt und alle notwendigen Verwaltungsstrukturen für das MyFS-Dateisystem angelegt („formatiert“)
- Beim Erstellen werden ausgewählte Dateien auf den Datenträger kopiert (einmalig)
- Wenn ein Datenträger mittels FUSE in den Verzeichnisbaum eingebunden wird, können enthaltene Dateien gelesen, aber (noch) nicht verändert oder gelöscht werden

Aufgabe Teil 2: Read-Write File System

- `mkfs.myfs` erstellt leere Datenträger mit fester Größe (optional kann Kopieren beibehalten werden)
- Wenn ein Datenträger mittels FUSE in Verzeichnisbaum eingebunden wird, können enthaltene Dateien gelesen, verändert und gelöscht werden, neue Dateien können eingefügt werden

Aufgabe Teil 3: Dokumentation

Teil 0: Verstehen, was FUSE tut

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

- 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 des Aufbaus von MyFS-Datenträgern und der Software-Architektur ihrer Lösung

Teilaufgabe 1b: Erstellen und Befüllen von Datenträgern mittels `mkfs.myfs`

Teilaufgabe 1c: Einbinden („mounten“) von Datenträgern mittels `mount.myfs`

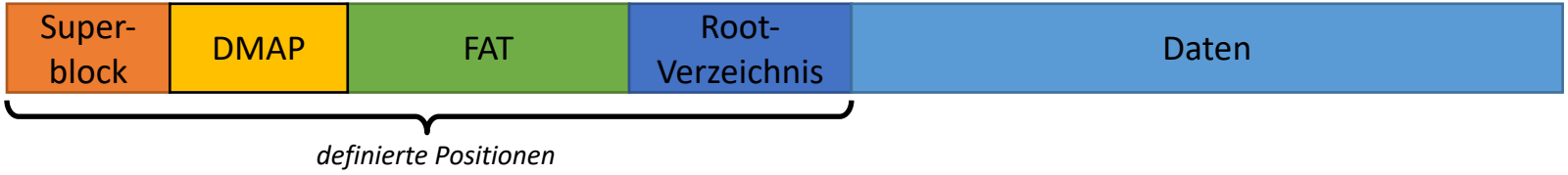
Teilaufgabe 1d: Ausführliches Testen!

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

1a: Aufbau von MyFS-Containerdateien

In MyFS-Datenträgern sollte (mindestens) folgendes abgelegt werden:

- *Superblock*: Informationen zum File-System (z.B. Größe, Positionen der Einträge unten...)
- *DMAP*: Verzeichnis der freien Datenblöcke
- *File Allocation Table* (FAT, vgl. Vorlesung Betriebssysteme)
- *Root-Verzeichnis* für Dateien im Dateisystem mit folgenden Einträgen:
  - Dateiname
  - Dateigröße
  - Benutzer / Gruppen-ID
  - Zugriffsberechtigungen (mode)
  - Zeitpunkt letzter Zugriff (atime) / letzte Veränderung (mtime) / letzte Statusänderung (ctime)
  - Zeiger auf ersten Datenblock
- Datenblocks der Dateien



Hinweis: Der Datenträger soll nur ein einziges (Root-)Verzeichnis haben, d.h. es müssen keine Verzeichnisbäume implementiert werden

1a: Aufbau von MyFS-Containerdateien

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Überlegen Sie auch, wie die entsprechenden Datenstrukturen im Speicher gehalten werden! Beispiel:

```
class MyFS {
...
SuperBlock mySuperBlock;
Dmap myDmap;
FAT myFAT;

int fuseInit(...) {
...
    // read Superblock, D-Map, FAT
    readStructures();
...
}
```

1b: Erstellen und Befüllen von MyFS-Containerdateien

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

MyFS-Datenträger sollen mit folgendem Kommando erstellt werden:

```
mkfs.myfs containerdatei [input-datei ...]
```

Dabei ist

- *Containerdatei* — Pfad- und Dateiname der Containerdatei des Datenträgers
- *[input-datei ...]* — Pfad- und Dateiname(n) der auf den Datenträger zu kopierenden Datei(en)

Beispiel:

```
mkfs.myfs container.bin text1.txt
   input/text2.txt /input2.txt
```

Hinweise zu `mkfs.myfs`

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

- Beim Kopieren sollen alle Dateien ins Root-Verzeichnis des Datenträgers kopiert werden, d.h. Pfadnamen werden entfernt
  - Doppelte Dateinamen sollen erkannt und ein Fehler gemeldet werden
- Die Informationen, die für die Befüllung der Verzeichniseinträge benötigt werden, können durch die C-Funktion `stat()` abgefragt werden (vgl. `man 2 stat`)
  - Die Attribute `st_atime`, `st_ctime` sollen dabei auf die aktuelle Zeit gesetzt werden, `st_mtime` soll von der Originaldatei übernommen werden.
  - Setzen Sie die Attribute `st_uid` und `st_gid` auf Benutzer- und Gruppen-ID des aktuellen Benutzers
- Füllen Sie dabei die Zugriffsberechtigungen (Mode) für die Dateien mit `S_IFREG | 0444`
- **Für diese Aufgabe benötigen Sie noch keine FUSE-Operationen!**
  - Sie können direkt auf der Klasse `BlockDevice` arbeiten
  - Ggf. können Sie relevante Methoden bereits in der Klasse `MyFS` implementieren, um diese in Teil 1c wiederverwenden zu können

1c: Einbinden von MyFS-Datenträgern

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

MyFS-Datenträger sollen mit folgendem Kommando eingebunden werden:

```
mount.myfs containerdatei logdatei mount-dir
```

Dabei ist

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

Beispiel:

```
mount.myfs container.bin logfile mount
```

Die Einbindung kann mit dem Kommando `fusermount --unmount` gelöst werden

Hinweise zu 1c

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

- Zum Initialisieren / Freigeben einer Containerdatei
  - `MyFS::fuseInit()`
  - `MyFS::fuseDestroy()` (*)
- Zum Anzeigen eines Verzeichnisses
  - `MyFS::fuseOpendir()` (**)
  - `MyFS::fuseReaddir()`
  - `MyFS::fuseReleasedir()` (**)
  - `MyFS::fuseGettatr()`
- Zum Lesen einer Datei
  - `MyFS::fuseOpen()`
  - `MyFS::fuseRead()`
  - `MyFS::fuseRelease()`

(*) 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 | 0555`
- 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)

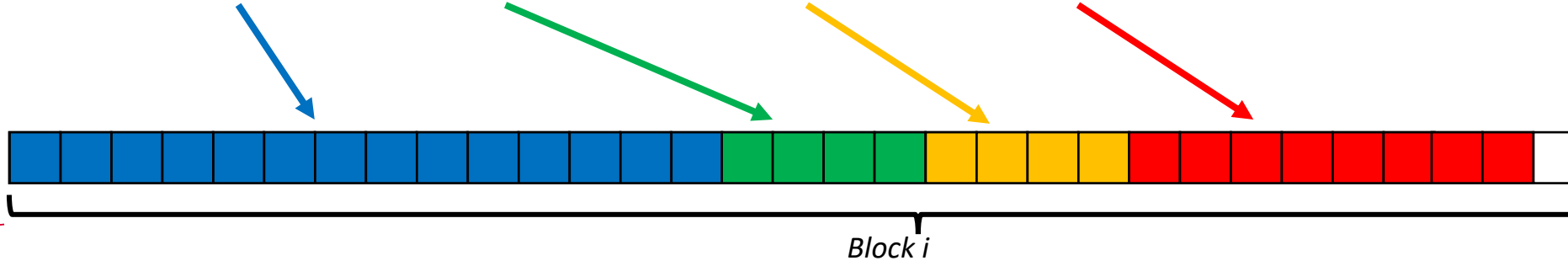
```
int MyFS::fuseOpen(const char *path,
        struct fuse_file_info *fileInfo)
int MyFS::fuseRead(const char *path,
        char *buf, size_t size, off_t offset,
        struct fuse_file_info *fileInfo)
int MyFS::fuseRelease(const char *path,
        struct fuse_file_info *fileInfo)
```

- 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
- Puffern Sie in der Funktion `fuseRead()` den aktuellen Block, um Leseinformationen einzusparen – siehe nächste Seite!
- Sinnvolle Fehlercodes
  - `EMFILE` – zu viele geöffnete Dateien
  - `ENOENT` – Datei nicht gefunden
  - `EBADF` – Wert in `fileInfo->fh` zeigt nicht auf eine geöffnete Datei
  - `ENXIO` – Zugriff über Dateiende hinaus

Hinweise zur Pufferung

Anwendungsprogramme lesen häufig nur jeweils wenige Bytes in mehreren aufeinanderfolgenden Leseoperationen aus einer Datei

- Bsp.: [`read(p,n)` liefert `n` Bytes ab der absoluten Position `p` in der Datei]:
  `read(0,13)` – `read(13,4)` – `read(17,4)` – `read(21,8)` – …



In solchen Fällen würde durch eine unbedarfte Implementierung der betroffene Block viermal (oder öfter) aus der Containerdatei gelesen

→ Für jede geöffnete Datei soll der zuletzt gelesene Block gepuffert werden

→ Bei jedem Zugriff wird geprüft, ob die gewünschten Bytes im gepufferten Block liegen, in diesem Fall wird Lesen aus der Containerdatei vermieden

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

1d: 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)

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Speicherplatz im Dateisystem

- Ihr Dateisystem sollte mindestens 30 MB freien Platz für Dateien bieten
  - Größe der Verwaltungsstrukturen berücksichtigen und Größe der Containerdatei entsprechend wählen!

Konstanten

- `#define NAME_LENGTH 255`
  - Max. Länge eines Dateinamens
- `#define BLOCK_SIZE 512`
  - Logische Blockgröße
- `#define NUM_DIR_ENTRIES 64`
  - Anzahl der Verzeichniseinträge
- `#define NUM_OPEN_FILES 64`
  - Anzahl offener Dateien pro MyFS Containerdatei

Vorgaben (2)

- Ein Makefile-Projekt für die Ziele `mkfs.myfs` und `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 Eclipse, CLion, XCode

Vorgaben (3)

- Verwenden Sie für alle Zugriffe auf die Containerdatei die Klasse `BlockDevice`
  - Ebenfalls in bereitgestellten Projekt enthalten
  - Simuliert ein generisches Block Device, das Blöcke fester Größe `BLOCK_SIZE` mit Hilfe von logischen Blocknummern {0, 1, ..., *LAST_BLOCK*} adressieren kann
  - Stellt Operationen zum Erzeugen, Öffnen, Lesen, Schreiben von Containerdateien zur Verfügung

- 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

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 eigenen 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 abgebgeben 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-examplefs. https://code.google.com/archive/p/fuse-examplefs/ (abgerufen 06.10.2017)

[7] libfuse API documentation. https://libfuse.github.io/doxygen/index.html (abgerufen 06.11.2017)