Praktikum angewandte Systemsoftwaretechnik (PASST)

Dateisysteme / Aufgabe 6

21. Juni 2018

Stefan Reif, Peter Wägemann, Florian Schmaus, Michael Eischer, Andreas Ziegler, Bernhard Heinloth und Benedict Herzog

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Dateisysteme

Was ist ein Dateisystem?

- Beispiele: ext{2,3,4}, ntfs, ...
  - Verwaltet persistenten Speicher
  - Einheitliche Schnittstelle
  - Implementierung von `open()`, `read()`, `write()`, ...
- Pseudo-Dateisysteme: procfs, sysfs
  - Exportieren Kernel-Datenstrukturen
  - Zugriff über Dateischnittstelle
- Dateisystem-Instanzen: `/`, `/proc/`, `/mnt/.../`

Was ist ein Dateisystem?

- Beispiele: ext{2,3,4}, ntfs, ...
  - Verwaltet persistenten Speicher
  - Einheitliche Schnittstelle
  - Implementierung von `open()`, `read()`, `write()`, ...
- Pseudo-Dateisysteme: procfs, sysfs
  - Exportieren Kernel-Datenstrukturen
  - Zugriff über Dateischnittstelle
- Dateisystem-Instanzen: `/`, `/proc/`, `/mnt/.../`

Aufgabe 6

Implementierung eines Dateisystems im Linux-Kernel

Der Dateibaum

- Dateien sind als Baum organisiert
  - $\rightarrow$ „hierarchisches” Dateisystem
- Wurzelverzeichnis: /
- Manche Einträge sind „Mountpoints”
  - Dahinter verbirgt sich eine weitere Dateisysteminstanz
  - Operationen können dort eine vollständig andere Semantik haben

Das Linux Virtual Filesystem (VFS)

- Abstraktionsebene, die ...
  - Die Koexistenz von Dateisystemen und -instanzen erlaubt
  - Manche Aufgaben für Dateisysteme übernimmt
- Die Struktur erinnert stark an Objektorientierte Programmierung
  - Datenstrukturen, die Funktionszeiger beinhalten
    $\rightarrow$ Abstraktion
  - Teilweise Standardimplementierungen vorhanden
    $\rightarrow$ Vererbung

```
01 #include <linux/fs.h>
02
03 extern int register_filesystem(struct file_system_type *);
04 extern int unregister_filesystem(struct file_system_type *);
```

Begriffsklärung

- `struct file_system_type`
  → Ein Dateisystem
- `struct superblock`
  → Eine Dateisysteminstanz
- `struct inode`
  → Eine Datei auf der Festplatte
- `struct dentry`
  → Ein Directory-Entry
  - Wird in einem Cache gespeichert
  - Enthält einen Verweis auf ein `struct inode`
- `struct file`
  → Eine geöffnete Datei
  - Enthält einen Verweis auf ein `struct dentry`
- `ino_t`
  → Skalarer Wert, der eine Inode-Nummer enthält

Dateisysteme im Linux-Kernel (1/3)

- Das `struct file_system_type` repräsentiert ein Dateisystem

```
01 struct file_system_type {
02   const char *name;
03   int fs_flags;
04   struct dentry *(*mount) (struct file_system_type *, int, const
       ↪ char *, void *);
05   void (*kill_sb) (struct super_block *);
06   /* ... */
07 }
```

Dateisysteme im Linux-Kernel (2/3)

- Hilfsfunktionen sind vorhanden
- `fill_super`-Callback initialisiert die Datenstruktur

```
01 extern struct dentry *mount_bdev(
02   struct file_system_type *fs_type,
03   int flags, const char *dev_name, void *data,
04   int (*fill_super)(struct super_block *, void *, int));
```

Dateisysteme im Linux-Kernel (3/3)

- Superblock-Operationen müssen gesetzt werden
- Ein Info-Zeiger kapselt dateisystemspezifische Informationen

```
01 int myfs_fill_super(struct super_block *sb, /* ... */)
02 {
03   /* ... */
04   sb->s_op = &myfs_s_ops;
05   sb->s_fs_info = myinfo;
06 }
```

```
01 static const struct super_operations myfs_sops = {
02   .put_super = myfs_put_super,
03   /* ... */
04 }
```

Blöcke lesen und schreiben

- Dateisysteme verwenden Caches
- Schreiben passiert in der Regel asynchron

```
01 struct buffer_head *bh;
02 struct super_block *sb = ...;
03 bh = sb_read(sb, num);
04 memcpy(read_buffer, bh->b_data, read_buffer_size);
05
06 memcpy(bh->b_data, write_buffer, write_buffer_size);
07 mark_buffer_dirty(bh);
08 brelse(bh);
```

Inodes lesen und schreiben

- Hilfsfunktionen: `iget_locked`, `new_inode`, `unlock_new_inode`, `mark_inode_dirty`, `insert_inode_hash`
- Inode-Operationen müssen initialisiert werden

```
01 inode->i_op = myfs_iops;
02 inode->i_mapping->a_ops = myfs_aops;
```

- Inode-Operationen haben eine Default-Implementierung:

```
01 const struct inode_operations myfs_iops = {
02   .llseek     = generic_file_llseek,
03   .read_iter  = generic_file_read_iter,
04   /* ... */
05 }
```

- Wie funktionieren diese Default-Implementierungen?

Addressraum-Operationen (1/3)

- Die meisten Inode-Operationen basieren auf Adressraumoperationen

```
01 const struct address_space_operations myfs_aops = {
02   .readpage    = myfs_readpage,
03   .writepage   = myfs_writepage,
04   /* ... */
05 }
```

Addressraum-Operationen (2/3)

- Die Adressraumoperationen benötigen einen `get_block`-Callback

```
01 int myfs_writepage(struct page *page, struct writeback_control *
       ↪ wbc)
02 {
03   return block_write_full_page(page, myfs_get_block, wbc);
04 }
05
06 int myfs_readpage(struct file *file, struct page *page)
07 {
08   return block_read_full_page(page, myfs_get_block);
09 }
```

Addressraum-Operationen (3/3)

- `get_block` muss das Dateisystemlayout verstehen

```
01 int myfs_get_block(struct inode *inode, sector_t block, struct
       ↪ buffer_head *bh_result, int create)
02 {
03   struct super_block *sb = inode->i_sb;
04   unsigned long phys = ...;
05   map_bh(bh_result, sb, phys);
06 }
```

Aufgabe 6

Dateisystem

- Einarbeiten in die benötigten APIs im Linux-Kern
  - Dokumentation, Codebeispiele
  - Empfohlene Dateisysteme zum Verständnis: bfs, minix
- Block-Layout entwerfen
- Programmieren des Dateisystems
- Programmieren eines `mkfs`-Programms

Abgabe: bis 2018-07-13 durch Vorführung in einer Rechnerübung

Fragen?