Universität Bremen

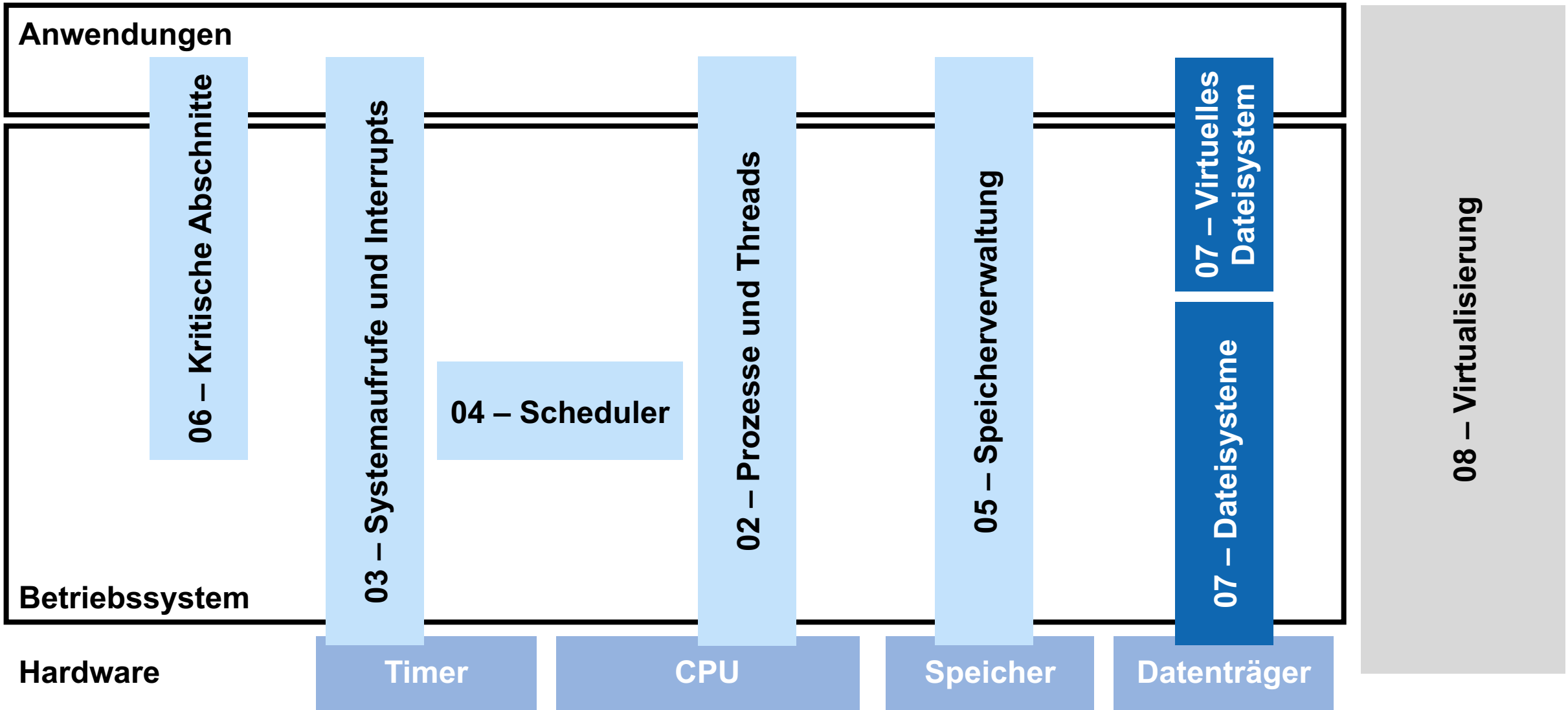
Betriebssysteme

Dateisysteme

Dr. Bernhard J. Berger
AG Sichere Systeme

Universität Bremen

Themenübersicht



Universität Bremen

Themenübersicht

- FS als API
- Terminologie
- Management

Grundlagen

- Grundidee
- API-Typen
- Laufzeitsicht
- Dateisysteme registrieren
- Strukturen
  - Superblock
  - Inode
  - Dentry
  - File

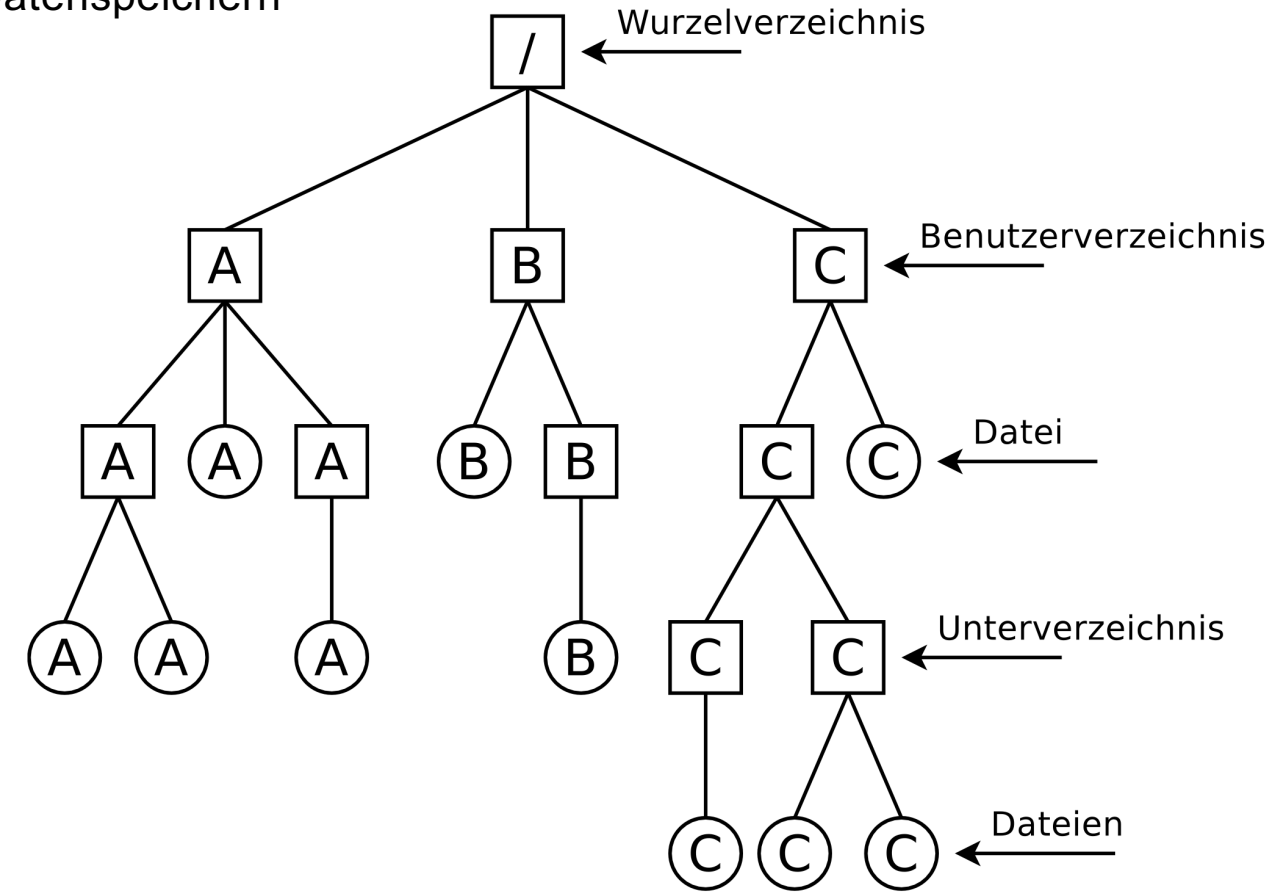
VFS

- Minix
- FAT-Dateisysteme
- ext-Dateisysteme
- Journaling
- Extents
- Copy-On-Write

Dateisysteme

Dateisystem als API

- Dateisystem
  - Hierarchische Organisationsstruktur für Dateien auf Datenspeichern
  - Dateien sind zusammengehörende Dateninhalte
- Dateisystem verwaltet
  - Dateinamen
  - Attribute (Metadaten)
- Dateisystem dient als
  - API des Betriebssystems
  - Dateinamen als Schnittstelle
- Wichtige Eigenschaften
  - Geschwindigkeit
  - Geringer Verschnitt



Hardware

- Dateisysteme adressieren Cluster
  - Jede Datei belegt eine ganzzahlige Menge an Clustern
  - Alternativbegriffe: Zonen oder Blöcke
  - Achtung: Sektoren von Festplatten werden auch manchmal Blöcke genannt
- Größe der Cluster
  - Abhängig vom Dateisystem und von Konfiguration
  - Beeinflusst Effizienz des Dateisystems
    - Steigender Verwaltungsaufwand für große Dateien
    - Abnehmender Kapazitätsverlust durch interne Fragmentierung
  - Wird beim Anlegen des Dateisystems festgelegt

Hinweis

Unter Linux wird empfohlen die Page-Size des Speichers bei der Wahl der Clustergröße zu berücksichtigen, damit Cluster effizient in den Speicher abgebildet werden können.

Kommandos

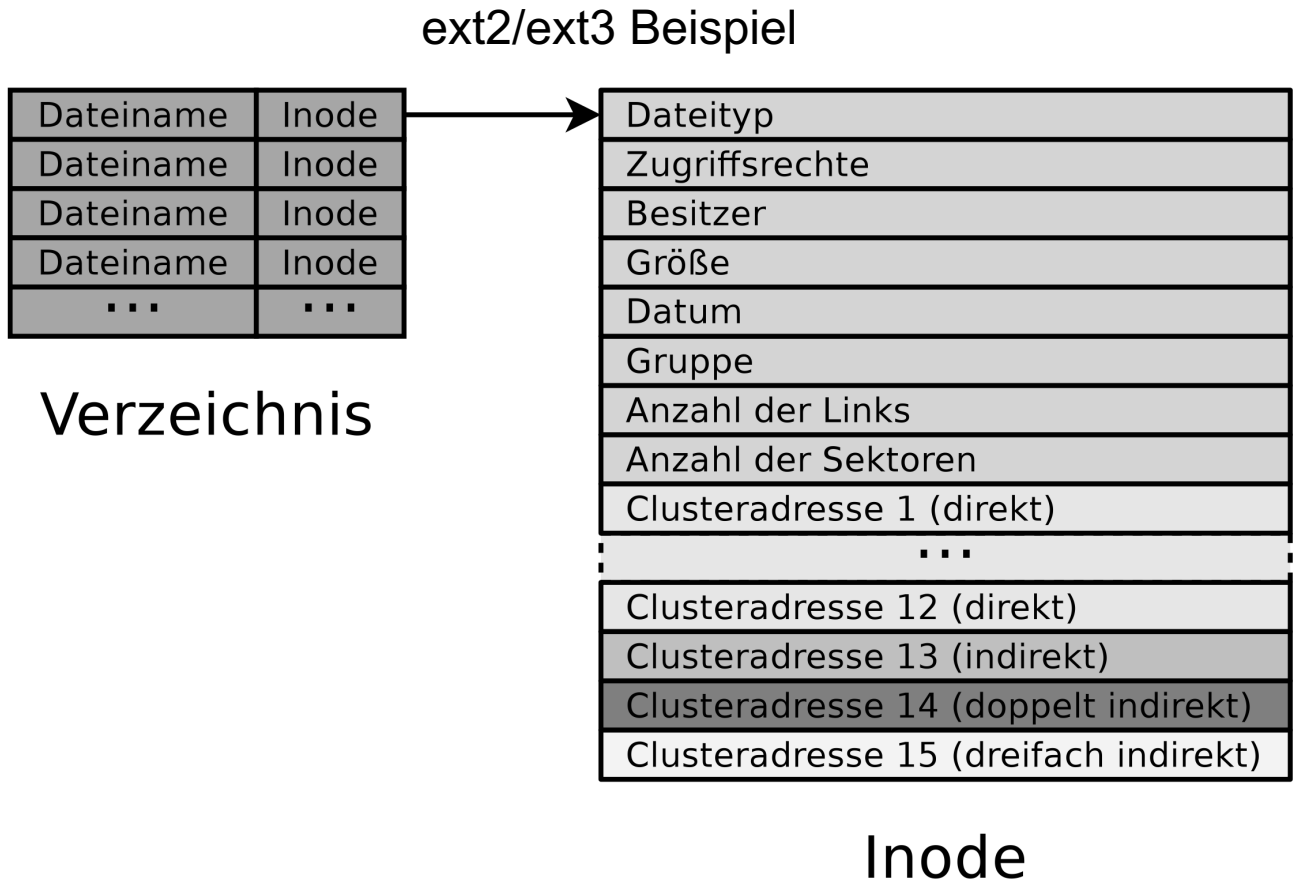
- Verwalten von Dateisystemen (Kommandozeile)
  - `mount [-t fs-type -o options] <device> <mount point>`
  - `umount <mount point>`
  - `mkfs.xxx`
  - `fsck.xxx`
- Beispiel zum Anlegen und Einbinden

Terminal

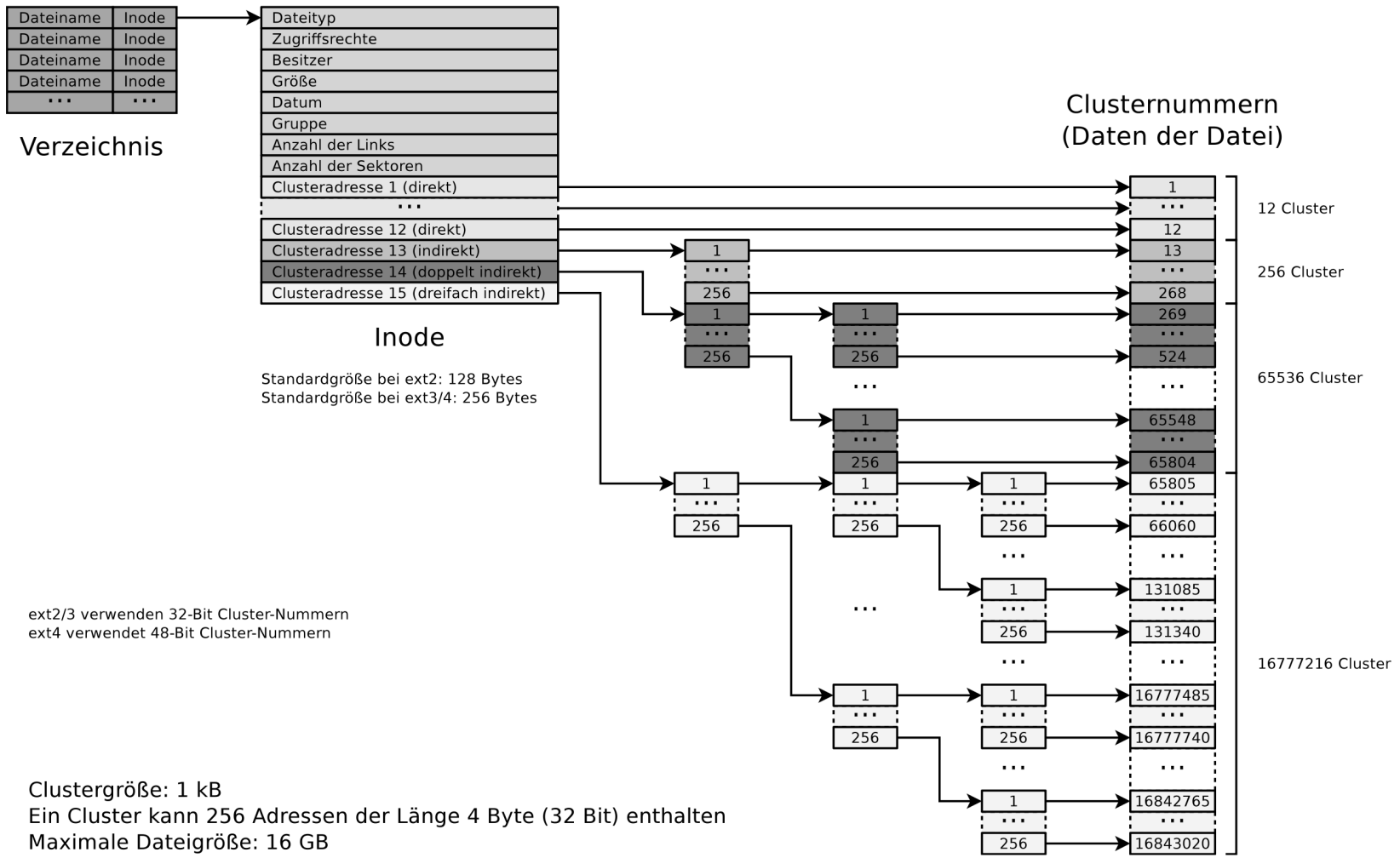
```
> dd if=/dev/zero of=loop_fs.img bs=1M count=100
> mkfs.ext2 loop_fs.img
> mkdir /mnt/my-fs
> mount -o loop loop_fs.img /mnt/my-fs
…
> umount /mnt/my-fs/
```

Linux-Dateisysteme

- Dateien und Verzeichnisse werden als ***Inode*** (Index Node) abgelegt
- Inode …
  - … enthält Metadaten
    - Dateigröße
    - UID/GID
    - Zugriffsrechte
    - Zugriffsdatum
  - … hat eine eindeutige Inode-Nummer
- Datei-Inode verweist auf Cluster mit Dateiinhalt
- Verzeichnis-Inode enthält Namen und Inode-Nummern für Verzeichnisinhalt



Beispiel ext2/ext3



Clustergröße: 1 kB
Ein Cluster kann 256 Adressen der Länge 4 Byte (32 Bit) enthalten
Maximale Dateigröße: 16 GB

Universität Bremen

Themenübersicht

- FS als API
- Terminologie
- Management

Grundlagen

- Grundidee
- API-Typen
- Laufzeitsicht
- Dateisysteme registrieren
- Strukturen
  - Superblock
  - Inode
  - Dentry
  - File

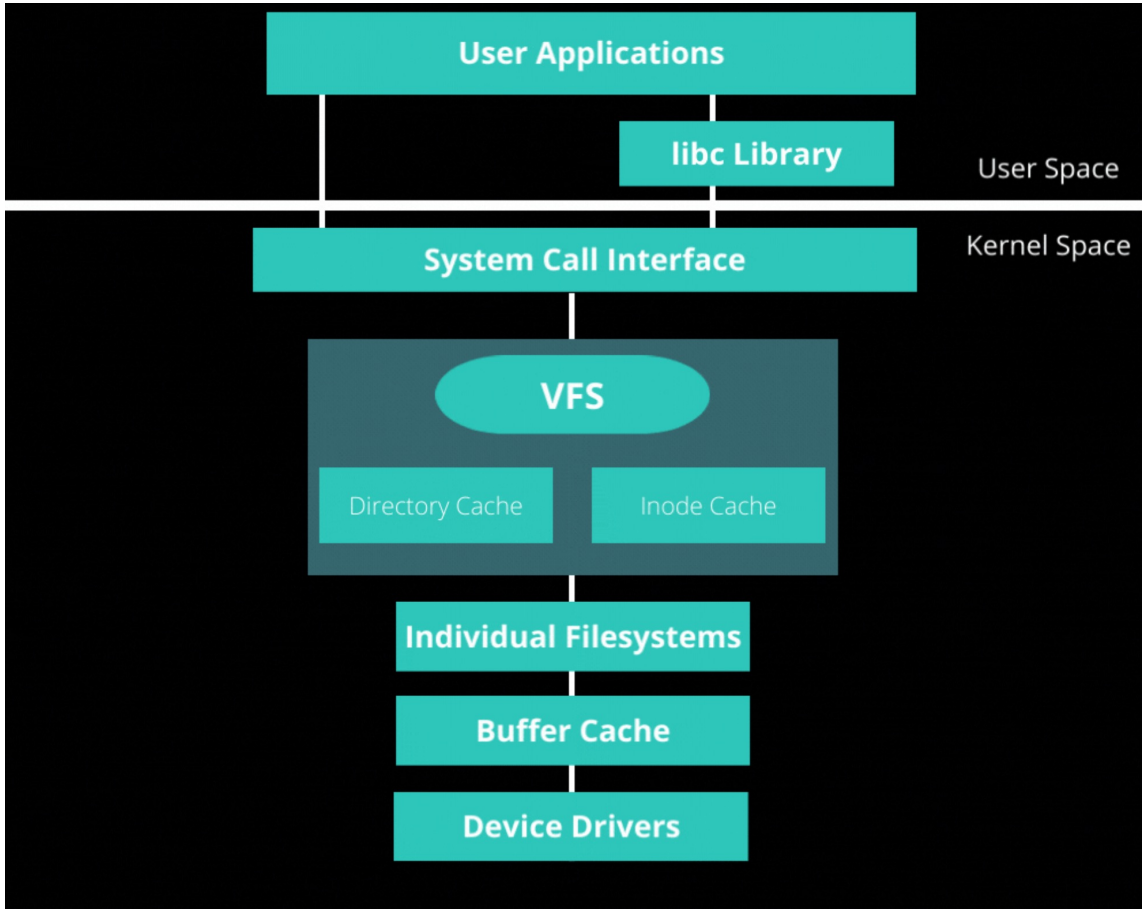
VFS

- Minix
- FAT-Dateisysteme
- ext-Dateisysteme
- Journaling
- Extents
- Copy-On-Write

Dateisysteme

Grundidee

- Linux VFS verwendet Objektorientierung (in C)
- Betriebssystem bietet Systemaufrufe an
  - open, close
  - read, write
  - lseek, unlink, mkdir, rmdir, readdir
- Systemaufrufe sind generisch umgesetzt
  - Operieren auf generischen Datenstrukturen
  - Führen grundlegende Buchführung durch
  - Verwendendet dateisystemspezifische Funktionen
- Linux bietet eine einfache Möglichkeit neue Dateisysteme zu implementieren.
  - Kernelmodul erstellen
  - Funktionen implementieren
  - Dateisystem registrieren
- Details
  https://git.fh-aachen.de/tb3838s/linux/-/blob/master/Documentation/filesystems/vfs.rst



https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour

API-Typen

VFS-Elemente: Insgesamt gibt es vier Objekte, die das VFS zur Verfügung stellt:

1. **Superblock-Objekt** Ein eingebundenes Dateisystem. Erlaubt das Einbinden und Aushängen.

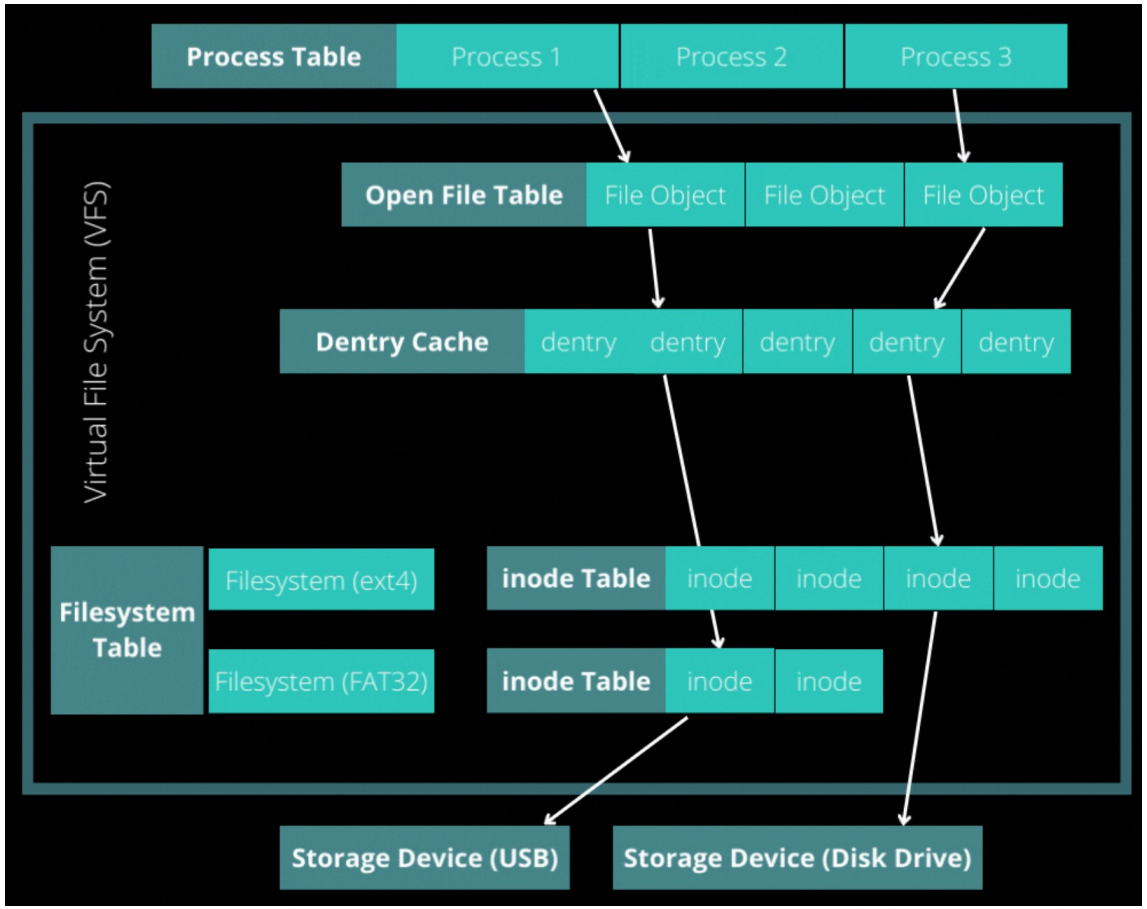
2. **Inode-Objekt** Eine konkrete Datei im Dateisystem. Stellt Funktionen zum Anlegen, Löschen, Umbenennen usw. zur Verfügung.

3. **Dentry-Objekt** Ein Verzeichniseintrag (Datei oder Verzeichnis) und zugehörige Verzeichnisoperationen, wie Dateinamenvergleich, aushängen aus dem Dateisystembaum

4. **File-Objekt** Stellt eine offene Datei dar und stellt alle dateibezogenen Operationen wie Lesen und Schreiben zur Verfügung

Laufzeitsicht

- Liste offener Dateien in Prozesstabelleneintrag
- Geöffnete Datei verweist auf dentry-Eintrag
- Dentry-Eintrag referenziert inodes der Datei
- Indode ordnet auf echtes Hardwaregerät zu
- Alle Strukturen sind VFS-Elemente



https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour

Dateisystem registrieren – Kernelmodul

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include <linux/fs.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Bernhard J. Berger <berber@tzi.de>");
MODULE_DESCRIPTION("Simple filesystem for the lecture operating systems");

// Forward declaration section
struct dentry *bsfs_mount (struct file_system_type *type, int flags,
                           const char *name, void *ptr);

const struct file_system_type bsfs = {
        .owner = THIS_MODULE,
        .name = "bsfs",
        .mount = bsfs_mount,
        .kill_sb = kill_litter_super
};
```

Dateisystem registrieren – Kernelmodul

```
static int __init module_start(void) {
        printk(KERN_INFO "Registering filesystem BSFS!\n");
        return register_filesystem(&bsfs);
}

static void __exit module_end(void) {
        printk(KERN_INFO "Unregistering filesystem BSFS!\n");
        unregister_filesystem(&bsfs);
}

struct dentry *bsfs_mount (struct file_system_type *type, int flags,
                           const char *name, void *ptr) {
        return NULL; // Implementierung ergänzen
}

module_init(module_start);
module_exit(module_end);
```

Der Superblock – include/linux/fs.h

- Datenstruktur `struct super_block`
  - Definiert in `include/linux/fs.h`
  - Linux/Unix-spezifische Sicht auf das physische Dateisystem
  - Implementierung muss zwischen Linux und physischer Sicht *vermitteln*

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

Liste der eingebundenen Dateisysteme (werden durch ihren Superblock repräsentiert)

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

Verweis auf das unterliegende Gerät

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

Informationen über die Größe des physikalischen Dateisystems.

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

Zeiger auf die Superblock-Operationen (siehe Registrierung des Dateisystems).

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

Operationen für den Superblock

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Der Superblock – include/linux/fs.h

```
struct super_block {
  struct list_head s_list;
  dev_t s_dev;
  unsigned char s_blocksize_bits;
  unsigned long s_blocksize;
  loff_t s_maxbytes;
  struct file_system_type *s_type;
  const struct super_operations *s_op;
  const struct dquot_operations *dq_op;
  const struct quotactl_ops *s_qcop;
  const struct export_operations
                              *s_export_op;
  unsigned long s_flags;
  unsigned long s_magic;
  struct dentry *s_root;
  struct rw_semaphore s_umount;
  ...
```

Zeiger auf das Wurzelverzeichnis des Dateisystems

Der Superblock – include/linux/fs.h

Liste aller geladenen inode-Strukturen

```
struct super_block {
  ...
  struct list_head s_inodes;
  struct hlist_bl_head s_anon;
  struct list_head s_files;
  struct list_head s_mounts;
  ...
  struct block_device *s_bdev;
  struct backing_dev_info *s_bdi;
  struct mtd_info *s_mtd;
  struct hlist_node s_instances;
```

Der Superblock – include/linux/fs.h

Liste aller offenen Dateieinträge

```
struct super_block {
  ...
  struct list_head s_inodes;
  struct hlist_bl_head s_anon;
  struct list_head s_files;
  struct list_head s_mounts;
  ...
  struct block_device *s_bdev;
  struct backing_dev_info *s_bdi;
  struct mtd_info *s_mtd;
  struct hlist_node s_instances;
```

Der Superblock – include/linux/fs.h

Zugehöriges Block-Gerät

```
struct super_block {
  ...
  struct list_head s_inodes;
  struct hlist_bl_head s_anon;
  struct list_head s_files;
  struct list_head s_mounts;
  ...
  struct block_device *s_bdev;
  struct backing_dev_info *s_bdi;
  struct mtd_info *s_mtd;
  struct hlist_node s_instances;
```

Die Superblock-Operationen – include/linux/fs.h

Anlegen und Zerstören einer inode

```
struct super_operations {
   struct inode *(*alloc_inode)(struct super_block *sb);
   void (*destroy_inode)(struct inode *);
   void (*dirty_inode) (struct inode *, int flags);
   int (*write_inode) (struct inode *,
                       struct writeback_control *wbc);
   int (*drop_inode) (struct inode *);
   void (*evict_inode) (struct inode *);
   void (*put_super) (struct super_block *);
   int (*sync_fs)(struct super_block *sb, int wait);
   int (*freeze_fs) (struct super_block *);
   int (*unfreeze_fs) (struct super_block *);
   int (*statfs) (struct dentry *, struct kstatfs *);
   int (*remount_fs) (struct super_block *, int *, char *);
   void (*umount_begin) (struct super_block *);
   …
```

Die Superblock-Operationen – include/linux/fs.h

Inode als geändert markieren. Wird für spezielle Operationen benötigt.

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *, int flags);
  int (*write_inode) (struct inode *,
                      struct writeback_control *wbc);
  int (*drop_inode) (struct inode *);
  void (*evict_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*umount_begin) (struct super_block *);
  …
```

Die Superblock-Operationen – include/linux/fs.h

Schreiben und Löschen einer inode auf/von physischem Gerät

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *, int flags);
  int (*write_inode) (struct inode *,
                      struct writeback_control *wbc);
  int (*drop_inode) (struct inode *);
  void (*evict_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*umount_begin) (struct super_block *);
  …
```

Die Superblock-Operationen – include/linux/fs.h

Superblock freigeben (bei unmount)

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *, int flags);
  int (*write_inode) (struct inode *,
                      struct writeback_control *wbc);
  int (*drop_inode) (struct inode *);
  void (*evict_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*umount_begin) (struct super_block *);
  …
```

Die Superblock-Operationen – include/linux/fs.h

Metadaten des Dateisystems auf das physische Gerät schreiben.

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *, int flags);
  int (*write_inode) (struct inode *,
                      struct writeback_control *wbc);
  int (*drop_inode) (struct inode *);
  void (*evict_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);
  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*umount_begin) (struct super_block *);
  …
```

Die inode – include/linux/fs.h

Berechtigungsinformationen

```
struct inode {
    umode_t i_mode;
    unsigned short i_opflags;
    kuid_t i_uid;
    kgid_t i_gid;
    unsigned int i_flags;
    struct posix_acl *i_acl;
    struct posix_acl *i_default_acl;
    const struct inode_operations *i_op;
    struct super_block *i_sb;
    struct address_space *i_mapping;
    dev_t i_rdev;
    loff_t i_size;
    …
```

Die inode – include/linux/fs.h

Zugriffskontrollinformationen

```
struct inode {
  umode_t i_mode;
  unsigned short i_opflags;
  kuid_t i_uid;
  kgid_t i_gid;
  unsigned int i_flags;
  struct posix_acl *i_acl;
  struct posix_acl *i_default_acl;
  const struct inode_operations *i_op;
  struct super_block *i_sb;
  struct address_space *i_mapping;
  dev_t i_rdev;
  loff_t i_size;
  …
```

Die inode – include/linux/fs.h

Inode bezogene Operationen

```
struct inode {
  umode_t i_mode;
  unsigned short i_opflags;
  kuid_t i_uid;
  kgid_t i_gid;
  unsigned int i_flags;
  struct posix_acl *i_acl;
  struct posix_acl *i_default_acl;
  const struct inode_operations *i_op;
  struct super_block *i_sb;
  struct address_space *i_mapping;
  dev_t i_rdev;
  loff_t i_size;
  …
```

Die inode – include/linux/fs.h

```
struct inode {
  umode_t i_mode;
  unsigned short i_opflags;
  kuid_t i_uid;
  kgid_t i_gid;
  unsigned int i_flags;
  struct posix_acl *i_acl;
  struct posix_acl *i_default_acl;
  const struct inode_operations *i_op;
  struct super_block *i_sb;
  struct address_space *i_mapping;
  dev_t i_rdev;
  loff_t i_size;
  …
```

Aktuelle Dateigröße

Die inode – include/linux/fs.h

Anzahl an Bytes im letzten Block

```
struct inode {
  unsigned short i_bytes;
  unsigned int i_blkbits;
  blkcnt_t i_blocks;
  struct hlist_node i_hash;
  struct list_head i_wb_list;
  struct list_head i_lru;
  struct list_head i_sb_list;
  union {
    struct hlist_head i_dentry;
    struct rcu_head i_rcu;
  };
  u64 i_version;
  atomic_t i_count;
  atomic_t i_dio_count;
  atomic_t i_writecount;
  const struct file_operations *i_fop;
  ...
  struct list_head i_devices;
  union {
    struct pipe_inode_info *i_pipe;
    struct block_device *i_bdev;
    struct cdev *i_cdev;
  };
  …
```

Die inode – include/linux/fs.h

Anzahl an Blöcken

```
struct inode {
  unsigned short i_bytes;
  unsigned int i_blkbits;
  blkcnt_t i_blocks;
  struct hlist_node i_hash;
  struct list_head i_wb_list;
  struct list_head i_lru;
  struct list_head i_sb_list;
  union {
    struct hlist_head i_dentry;
    struct rcu_head i_rcu;
  };
  u64 i_version;
  atomic_t i_count;
  atomic_t i_dio_count;
  atomic_t i_writecount;
  const struct file_operations *i_fop;
  ...
  struct list_head i_devices;
  union {
    struct pipe_inode_info *i_pipe;
    struct block_device *i_bdev;
    struct cdev *i_cdev;
  };
  …
```

Die inode – include/linux/fs.h

Liste von LRU-Seiten, die Blöcke der Datei cachen

```
struct inode {
  unsigned short i_bytes;
  unsigned int i_blkbits;
  blkcnt_t i_blocks;
  struct hlist_node i_hash;
  struct list_head i_wb_list;
  struct list_head i_lru;
  struct list_head i_sb_list;
  union {
    struct hlist_head i_dentry;
    struct rcu_head i_rcu;
  };
  u64 i_version;
  atomic_t i_count;
  atomic_t i_dio_count;
  atomic_t i_writecount;
  const struct file_operations *i_fop;
  ...
  struct list_head i_devices;
  union {
    struct pipe_inode_info *i_pipe;
    struct block_device *i_bdev;
    struct cdev *i_cdev;
  };
  …
```

Die inode – include/linux/fs.h

Dateioperationen

```
struct inode {
  unsigned short i_bytes;
  unsigned int i_blkbits;
  blkcnt_t i_blocks;
  struct hlist_node i_hash;
  struct list_head i_wb_list;
  struct list_head i_lru;
  struct list_head i_sb_list;
  union {
    struct hlist_head i_dentry;
    struct rcu_head i_rcu;
  };
  u64 i_version;
  atomic_t i_count;
  atomic_t i_dio_count;
  atomic_t i_writecount;
  const struct file_operations *i_fop;
  ...
  struct list_head i_devices;
  union {
    struct pipe_inode_info *i_pipe;
    struct block_device *i_bdev;
    struct cdev *i_cdev;
  };
  …
```

Die inode Operationen – include/linux/fs.h

Sucht file-Eintrag im angegebenen Verzeichnis

```
struct inode_operations {
  struct dentry * (*lookup) (struct inode *,
                             struct dentry *,
                             unsigned int);
  void * (*follow_link) (struct dentry *, struct nameidata *);
  int (*permission) (struct inode *, int);
  struct posix_acl * (*get_acl)(struct inode *, int);
  int (*readlink) (struct dentry *, char __user *,int);
  void (*put_link) (struct dentry *, struct nameidata *, void *);
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,umode_t);
  int (*rmdir) (struct inode *,struct dentry *);
  …
```

Die inode Operationen – include/linux/fs.h

Gibt ACL-Informationen der Inode zurück.

```
struct inode_operations {
  struct dentry * (*lookup) (struct inode *,
                             struct dentry *,
                             unsigned int);
  void * (*follow_link) (struct dentry *, struct nameidata *);
  int (*permission) (struct inode *, int);
  struct posix_acl * (*get_acl)(struct inode *, int);
  int (*readlink) (struct dentry *, char __user *,int);
  void (*put_link) (struct dentry *, struct nameidata *, void *);
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,umode_t);
  int (*rmdir) (struct inode *,struct dentry *);
  …
```

Die inode Operationen – include/linux/fs.h

Erzeugt eine neue Inode in dem angegebenen Ordner. Die übergebene Inode wird gefüllt.

```
struct inode_operations {
  struct dentry * (*lookup) (struct inode *,
                             struct dentry *,
                             unsigned int);
  void * (*follow_link) (struct dentry *, struct nameidata *);
  int (*permission) (struct inode *, int);
  struct posix_acl * (*get_acl)(struct inode *, int);
  int (*readlink) (struct dentry *, char __user *,int);
  void (*put_link) (struct dentry *, struct nameidata *, void *);
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,umode_t);
  int (*rmdir) (struct inode *,struct dentry *);
  …
```

Die inode Operationen – include/linux/fs.h

Hängt die übergebene Inode aus.

```
struct inode_operations {
  struct dentry * (*lookup) (struct inode *,
                             struct dentry *,
                             unsigned int);
  void * (*follow_link) (struct dentry *, struct nameidata *);
  int (*permission) (struct inode *, int);
  struct posix_acl * (*get_acl)(struct inode *, int);
  int (*readlink) (struct dentry *, char __user *,int);
  void (*put_link) (struct dentry *, struct nameidata *, void *);
  int (*create) (struct inode *,struct dentry *, umode_t, bool);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,umode_t);
  int (*rmdir) (struct inode *,struct dentry *);
  …
```

Die inode Operationen – include/linux/fs.h

```
struct inode_operations {
  …
  int (*rename) (struct inode *, struct dentry *,
                 struct inode *, struct dentry *);
  int (*setattr) (struct dentry *, struct iattr *);
  int (*getattr) (struct vfsmount *mnt, struct dentry *,
                  struct kstat *);
  int (*setxattr) (struct dentry *, const char *,
                   const void *,size_t,int);
  ssize_t (*getxattr) (struct dentry *, const char *,
                       void *, size_t);
  ssize_t (*listxattr) (struct dentry *, char *, size_t);
  int (*removexattr) (struct dentry *, const char *);
  int (*fiemap)(struct inode *, struct fiemap_extent_info *,
                u64 start, u64 len);
  int (*update_time)(struct inode *, struct timespec *, int);
  int (*atomic_open)(struct inode *, struct dentry *,
                     struct file *, unsigned open_flag,
                     umode_t create_mode, int *opened);
```

Erzeugt und öffnet eine Datei

Das dentry Objekt – include/linux/dcache.h

Die Inode, die diesen dentry darstellt

```
struct dentry {
  unsigned int d_flags;
  seqcount_t d_seq;
  struct hlist_bl_node d_hash;
  struct dentry *d_parent;
  struct qstr d_name;
  struct inode *d_inode;
  unsigned char d_iname[DNAME_INLINE_LEN];
  unsigned int d_count;
  spinlock_t d_lock;
  const struct dentry_operations *d_op;
  struct super_block *d_sb;
  …
```

Das dentry Objekt – include/linux/dcache.h

Operationen für den dentry

```
struct dentry {
  unsigned int d_flags;
  seqcount_t d_seq;
  struct hlist_bl_node d_hash;
  struct dentry *d_parent;
  struct qstr d_name;
  struct inode *d_inode;
  unsigned char d_iname[DNAME_INLINE_LEN];
  unsigned int d_count;
  spinlock_t d_lock;
  const struct dentry_operations *d_op;
  struct super_block *d_sb;
  …
```

Das dentry Operationen – include/linux/dcache.h

Vergleiche zwei Verzeichniseinträge anhand ihres Namens.

```
struct dentry_operations {
  int (*d_revalidate)(struct dentry *, unsigned int);
  int (*d_weak_revalidate)(struct dentry *, unsigned int);
  int (*d_hash)(const struct dentry *,
                const struct inode *,
                struct qstr *);
  int (*d_compare)(const struct dentry *,
                   const struct inode *,
                   const struct dentry *,
                   const struct inode *,
                   unsigned int,
                   const char *, const struct qstr *);
  int (*d_delete)(const struct dentry *);
  void (*d_release)(struct dentry *);
  void (*d_prune)(struct dentry *);
  void (*d_iput)(struct dentry *, struct inode *);
  char *(*d_dname)(struct dentry *, char *, int);
  struct vfsmount *(*d_automount)(struct path *);
  int (*d_manage)(struct dentry *, bool);
}
```

Das File Objekt

Liste aller File-Objekte

```
struct file {
  union {
    struct list_head fu_list;
    struct rcu_head fu_rcuhead;
  } f_u;
  struct path f_path;
  struct inode *f_inode;
  const struct file_operations *f_op;
  spinlock_t f_lock;
  atomic_long_t f_count;
  unsigned int f_flags;
  fmode_t f_mode;
  loff_t f_pos;
  struct fown_struct f_owner;
  const struct cred *f_cred;
  struct file_ra_state f_ra;
```

Das File Objekt

Flags zum genaueren Spezifizieren des Eintrags

```
struct file {
  union {
    struct list_head fu_list;
    struct rcu_head fu_rcuhead;
  } f_u;
  struct path f_path;
  struct inode *f_inode;
  const struct file_operations *f_op;
  spinlock_t f_lock;
  atomic_long_t f_count;
  unsigned int f_flags;
```

```
enum path_flags {
  PATH_IS_DIR = 0x1,                /* path is a directory */
  PATH_CONNECT_PATH = 0x4,          /* connect disconnected paths to / */
  PATH_CHROOT_REL = 0x8,            /* do path lookup relative to chroot */
  PATH_CHROOT_NSCONNECT = 0x10,     /* connect paths that are at ns root */

  PATH_DELEGATE_DELETED = 0x08000, /* delegate deleted files */
  PATH_MEDIATE_DELETED = 0x10000,  /* mediate deleted paths */
};
```

Das File Objekt

Dateioperationen

```
struct file {
  union {
    struct list_head fu_list;
    struct rcu_head fu_rcuhead;
  } f_u;
  struct path f_path;
  struct inode *f_inode;
  const struct file_operations *f_op;
  spinlock_t f_lock;
  atomic_long_t f_count;
  unsigned int f_flags;
  fmode_t f_mode;
  loff_t f_pos;
  struct fown_struct f_owner;
  const struct cred *f_cred;
  struct file_ra_state f_ra;
```

Das File Objekt

```
struct file {
  union {
    struct list_head fu_list;
    struct rcu_head fu_rcuhead;
  } f_u;
  struct path f_path;
  struct inode *f_inode;
  const struct file_operations *f_op;
  spinlock_t f_lock;
  atomic_long_t f_count;
  unsigned int f_flags;
  fmode_t f_mode;
  loff_t f_pos;
  struct fown_struct f_owner;
  const struct cred *f_cred;
  struct file_ra_state f_ra;
```

Position des Lese-/Schreibindex

Das File Objekt

Setzen des Lese-/Schreibindex

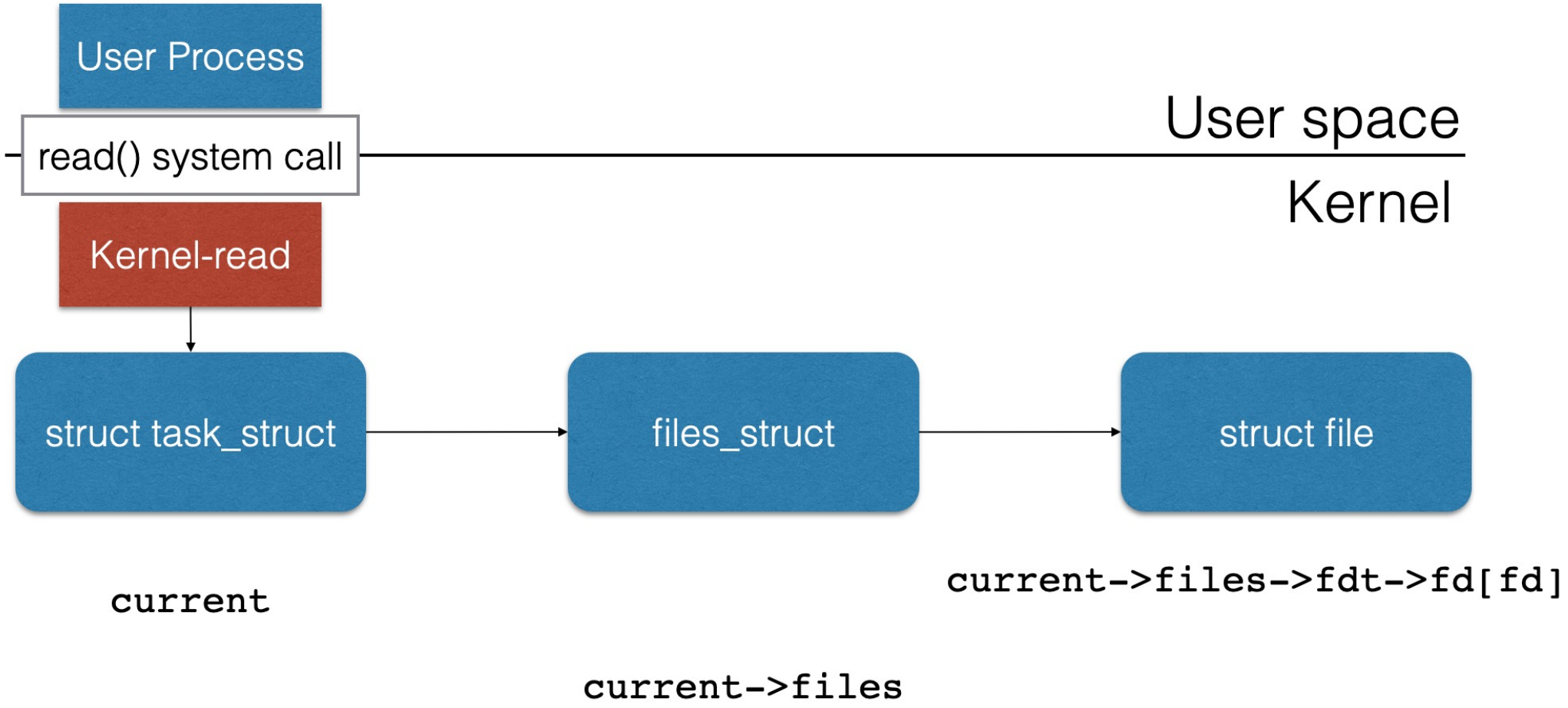
```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *,
                   size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *,
                    size_t, loff_t *);
  ...
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *,
                        struct poll_table_struct *);
  ..
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t,
                int datasync);
  …
```

Das File Objekt

Lese und Schreiben von Daten

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *,
                   size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *,
                    size_t, loff_t *);
  ...
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *,
                        struct poll_table_struct *);
  ..
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t,
                int datasync);
  …
```

Beispiel eines Systemaufrufs



Beispiel eines Systemaufrufs - fs/read_write.c

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

Beispiel eines Systemaufrufs - fs/read_write.c

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

```
struct fd {
  struct file *file;
  int need_put;
};
```

Beispiel eines Systemaufrufs - fs/read_write.c

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

Aufruf von `fget_light`

Beispiel eines Systemaufrufs - `fs/read_write.c`

```
struct file *fget_light(unsigned int fd, int *fput_needed)
{
  struct file *file;
  struct files_struct *files = current->files;
  *fput_needed = 0;
  if (atomic_read(&files->count) == 1) {
    file = fcheck_files(files, fd);
    if (file && (file->f_mode & FMODE_PATH))
      file = NULL;
    } else {
      rcu_read_lock();
      file = fcheck_files(files, fd);
      if (file) {
        if (!(file->f_mode & FMODE_PATH) &&
     atomic_long_inc_not_zero(&file->f_count))
          *fput_needed = 1;
        else
         /* Didn't get the reference, someone's freed */
         file = NULL;
    }
  rcu_read_unlock();
  }
  return file;
}
```

Beispiel eines Systemaufrufs - `fs/read_write.c`

```
struct file *fget_light(unsigned int fd, int *fput_needed)
{
  struct file *file;
  struct files_struct *files = current->files;
  *fput_needed = 0;
  if (atomic_read(&files->count) == 1) {
    file = fcheck_files(files, fd);
    if (file && (file->f_mode & FMODE_PATH))
      file = NULL;
    } else {
      rcu_read_lock();
      file = fcheck_files(files, fd);
      if (file) {
        if (!(file->f_mode & FMODE_PATH) &&
     atomic_long_inc_not_zero(&file->f_count))
          *fput_needed = 1;
        else
         /* Didn't get the reference, someone's freed */
         file = NULL;
    }
  rcu_read_unlock();
  }
  return file;
}
```

Holt alle offenen Dateien des aktiven Prozesses

Beispiel eines Systemaufrufs - `fs/read_write.c`

```
struct file *fget_light(unsigned int fd, int
*fput_needed)
{
  struct file *file;
  struct files_struct *files = current->files;
  *fput_needed = 0;
  if (atomic_read(&files->count) == 1) {
    file = fcheck_files(files, fd);
    if (file && (file->f_mode & FMODE_PATH))
      file = NULL;
    } else {
      rcu_read_lock();
      file = fcheck_files(files, fd);
      if (file) {
        if (!(file->f_mode & FMODE_PATH) &&
     atomic_long_inc_not_zero(&file->f_count))
          *fput_needed = 1;
        else
         /* Didn't get the reference, someone's freed */
         file = NULL;
    }
  rcu_read_unlock();
  }
  return file;
}
```

Liefert `files->fdt->fd[fd]`

Beispiel eines Systemaufrufs - `fs/read_write.c`

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

Hole Lese-/Schreibindex aus `f.file`

Beispiel eines Systemaufrufs - fs/read_write.c

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

Rufe die eigentliche Lesefunktion auf

Beispiel eines Systemaufrufs

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ))
    return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
    return -EFAULT;

  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Beispiel eines Systemaufrufs

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ))
    return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
    return -EFAULT;

  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Dateisystem mit Lesen beauftragen

Beispiel eines Systemaufrufs - fs/read_write.c

```
SYSCALL_DEFINE3(read,
                unsigned int, fd,
                char __user *, buf,
                size_t, count)
{
  struct fd f = fdget(fd);
  ssize_t ret = -EBADF;
  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    file_pos_write(f.file, pos);
    fdput(f);
  }
  return ret;
}
```

Aktualisiere Position im File-Objekt

Universität Bremen

Themenübersicht

- FS als API
- Terminologie
- Management

Grundlagen

- Grundidee
- API-Typen
- Laufzeitsicht
- Dateisysteme registrieren
- Strukturen
  - Superblock
  - Inode
  - Dentry
  - File

VFS

- Minix
- FAT-Dateisysteme
- ext-Dateisysteme
- Journaling
- Extents
- Copy-On-Write

Dateisysteme

Minix-Dateisystem

Bereich 1	Bereich 2	Bereich 3	Bereich 4	Bereich 5	Bereich 6
Bootblock (1 Cluster)	Superblock (1 Cluster)	Inodes-Bitmap (1 Cluster)	Block-Bitmap (1 Cluster)	Inodes (15 Cluster)	Daten (??? Cluster)

- **Bootblock** Boot-Loader für das Betriebssystem, Informationen über das Dateisystem,
  - z.B. Anzahl der Inodes und Cluster
- **Inodes-Bitmap** “Bitmap” die speichert, ob Inodes belegt (Wert: 1) oder frei (Wert: 0) sind
- **Cluster-Bitmap** “Bitmap” die speichert, ob Cluster belegt (Wert: 1) oder frei (Wert: 0) sind
- **Inodes** Enthält Inodes mit den Metadaten
  - Jede Datei und jedes Verzeichnis hat einen Inode mit Metadaten (Dateityp, UID/GID, Zugriffsrechte, Größe)
- **Daten** Inhalt der Dateien und Verzeichnisse

FAT - Dateizuordnungstabelle

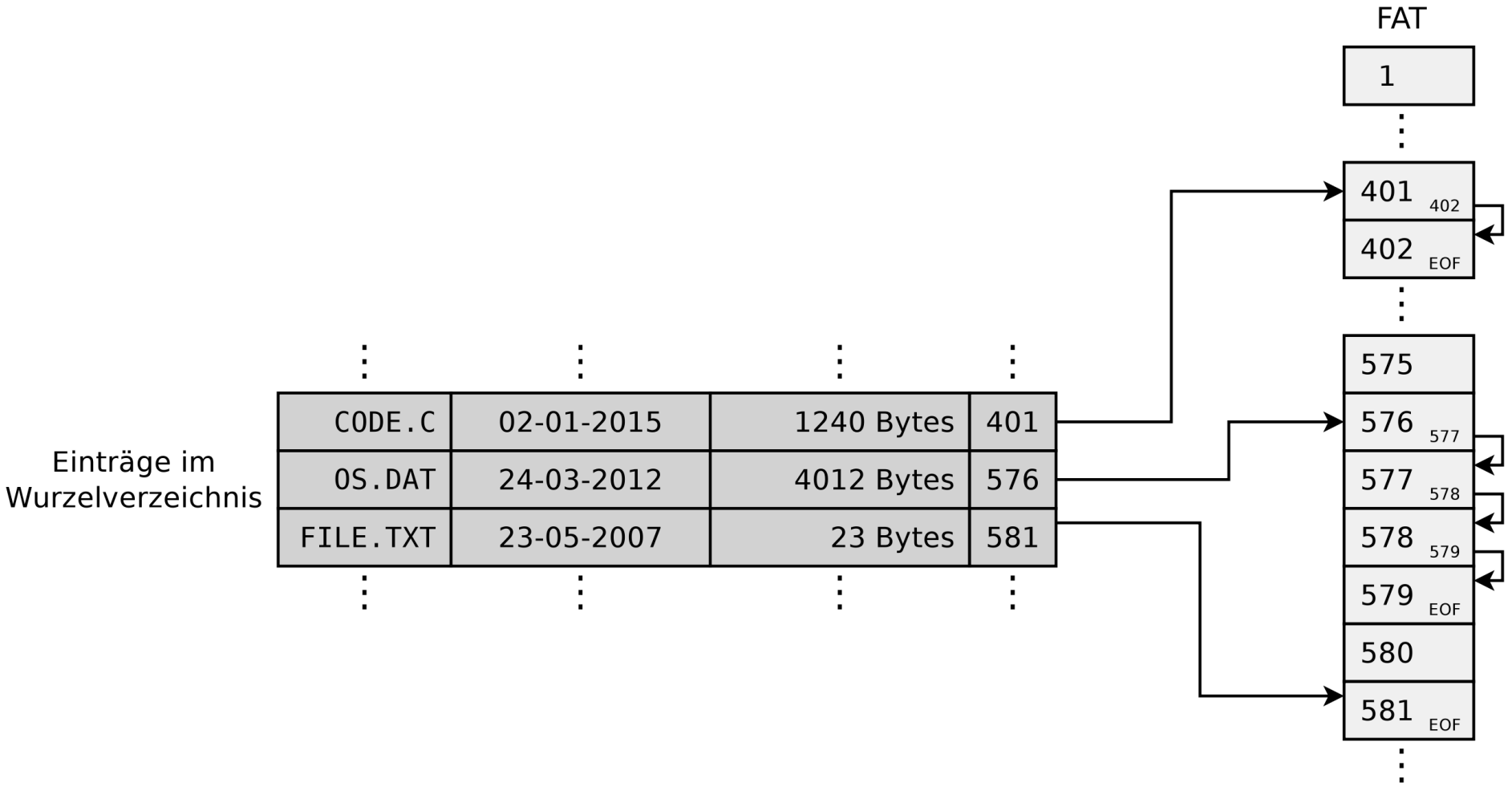
- Namensgeber ist Datenstruktur **File Allocation Table** (FAT, Dateizuordnungstabelle)
- Größe der FAT ist fest
- FAT speichert für jeden Cluster folgenden Informationen:
  - Frei
  - Beschädigt
  - Belegt
- Cluster speichert Verweis auf den nächsten Cluster einer Datei
- FAT implementiert das Konzept einer einfach verketteten Liste

FAT - Dateizuordnungstabelle

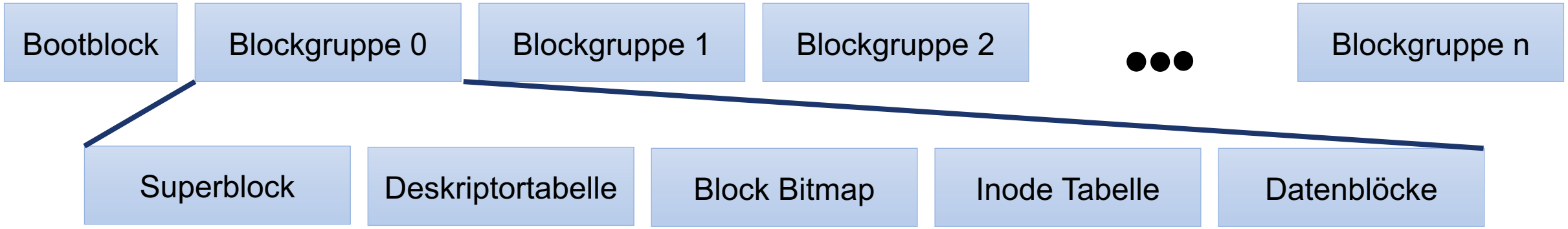
Bereich 1	Bereich 2	Bereich 3	Bereich 4	Bereich 5	Bereich 6
Bootblock (1 Cluster)	Reservierte Cluster	FAT 1	FAT 2	Stammverzeichnis	Daten

- **Bootblock** Boot-Loader für das Betriebssystem, Informationen über das Dateisystem
  - z.B. Blöcke pro Cluster, Anzahl an Blöcken
- **Reservierte Cluster** Zusätlicher Platz für den Bootmanager (wenn notwendig)
- **FAT** Informationen über belegte Blöcke (Bereich ist dupliziert um Ausfallwahrscheinlichkeit zu reduzieren)
- **Stammverzeichnis** Informationen zum Wurzelverzeichnis
  - z.B. Dateinamen, Größe, Datum, Dateityp (Verzeichnis, Datei), FAT-Eintrag
- **Daten** Inhalt der Dateien und Verzeichnisse

FAT - Dateizuordnungstabelle

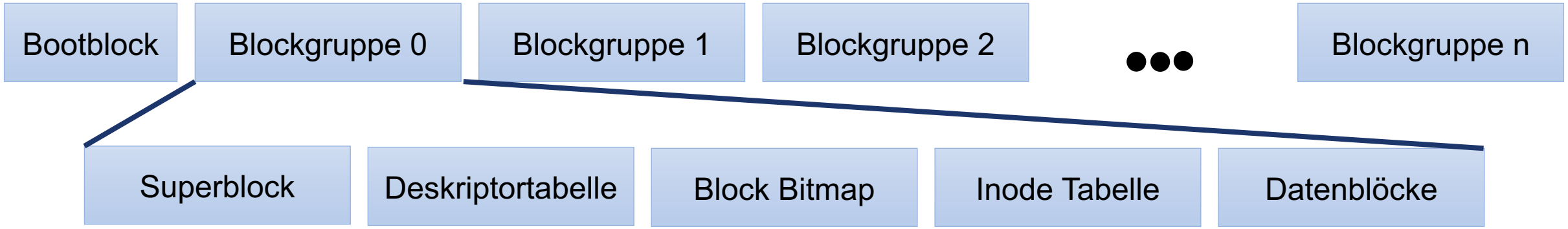


ext-Dateisysteme



- Der Datenträger wird in gleich große Blockgruppen unterteilt
- Jede Blockgruppe enthält alle Informationen für diese Blockgruppe
- Vorteile
  - Lokalitätsprinzip der Informationen
  - Schnellere Zugriffszeiten bei Festplatten
  - Replikation des Superblocks (Datensicherheit)

ext-Dateisysteme



- **Bootblock** in Cluster 1 (1 kB)
- Die **Deskriptor-Tabelle** enthält
  - Clusternummer des Block-Bitmaps
  - Clusternummer des Inode-Bitmaps
  - Anzahl der freien Cluster und Inodes in der Blockgruppe
- **Block-** und **Inode-Bitmap** sind jeweils einen Cluster groß
  - Speichern Informationen über die belegten Cluster und Inodes der Blockgruppe
- Die **Inode-Tabelle** enthält die Inodes der Blockgruppe
- Die restlichen Cluster der Blockgruppe sind die **Datenblöcke**

Journaling

- Beim Modifizieren von Verzeichnisstrukturen oder Metadaten sind Schreibzugriffe notwendig
  - Die Änderungen an dem Dateisystem sind Transaktionen (atomar, konsistent, isoliert, dauerhaft)
- Bei Ausfall während des Schreibens ist anschließende Konsistenzprüfung notwendig
  - In großen Dateisystemen kann die Prüfung sehr lange (Stunden bis Tage) dauern
- Die Konsistenzprüfung zu überspringen, kann zum Datenverlust führen
- „Änderungsjournal“ über die Schreibzugriffe kann Prüfaufwand einschränken

Journaling

- Dateisystem führt ein Journal und sammelt Schreibzugriffe
- Nach festem Zeitabstand wird das Journal geschlossen und die Operationen durchgeführt
- **Vorteil**:
  Nach einem Absturz müssen nur Dateien und Metadaten aus dem Journal überprüft werden
- **Nachteil**
  Durch Journaling mehr Schreiboperationen, da Änderungen erst ins Journal geschrieben und danach durchgeführt werden
- Varianten:
  - Metadaten-Journaling
  - Vollständiges Journaling

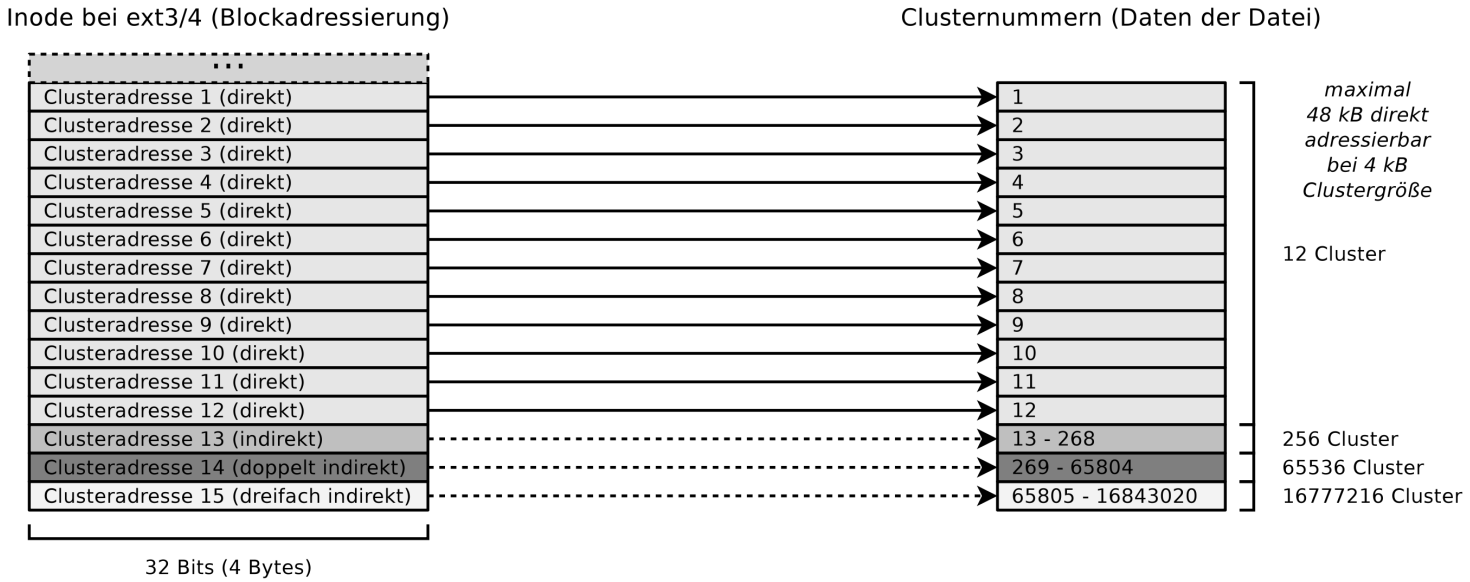
Journaling

- **Metadaten-Journaling** (Write-Back)
  - Nur Änderungen (Inodes) an Metadaten im Journal
  - Nur Metadaten können auf Korrektheit geprüft werden
  - Konsistenzprüfungen nach wenigen Sekunden abgeschlossen
  - Datenverlust noch immer möglich
  - ext3/4, NTFS und XFS unterstützen diese Version

- **Vollständiges Journaling**
  - Änderungen an Metadaten und Dateien werden protokolliert
  - Konsistenz der Dateien ist gewährleistet
  - Alle Schreiboperation werden doppelt durchgeführt
  - Diese Version ist bei ext3/4 optional
  - NTFS und XFS unterstützen diese Option nicht

Extents

- Inodes verwenden
  - Direkte Blockadressierung
  - Indirekte Blockadressierung
  - Doppelt indirekte Blockadressierung

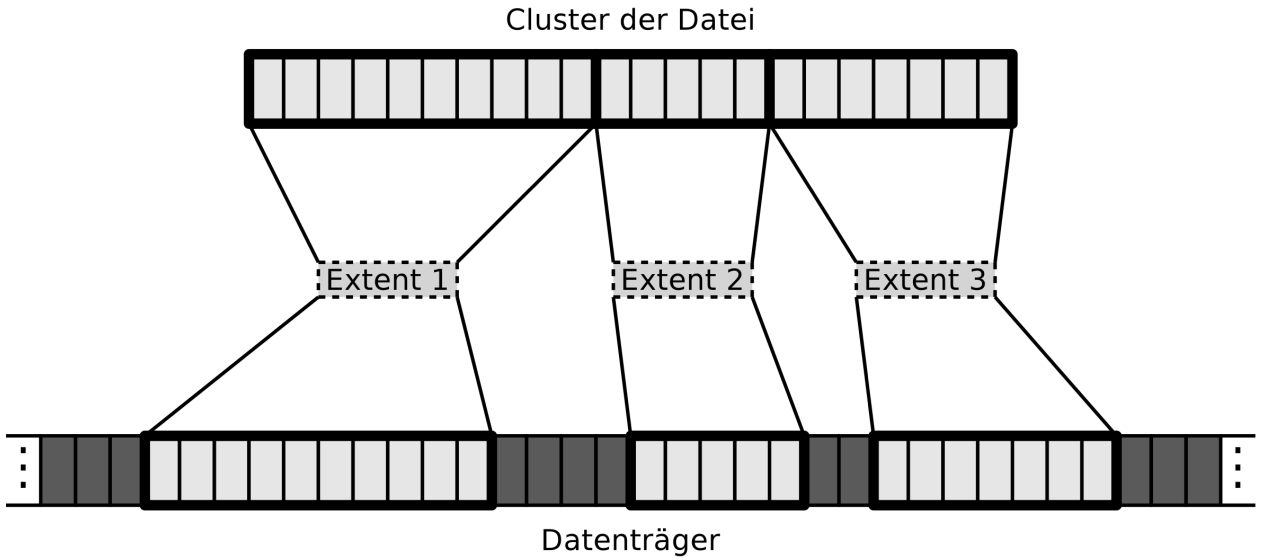


- Hierdurch steigt der Overhead für Verwaltungsinformationen
- Lösung: **Extents**

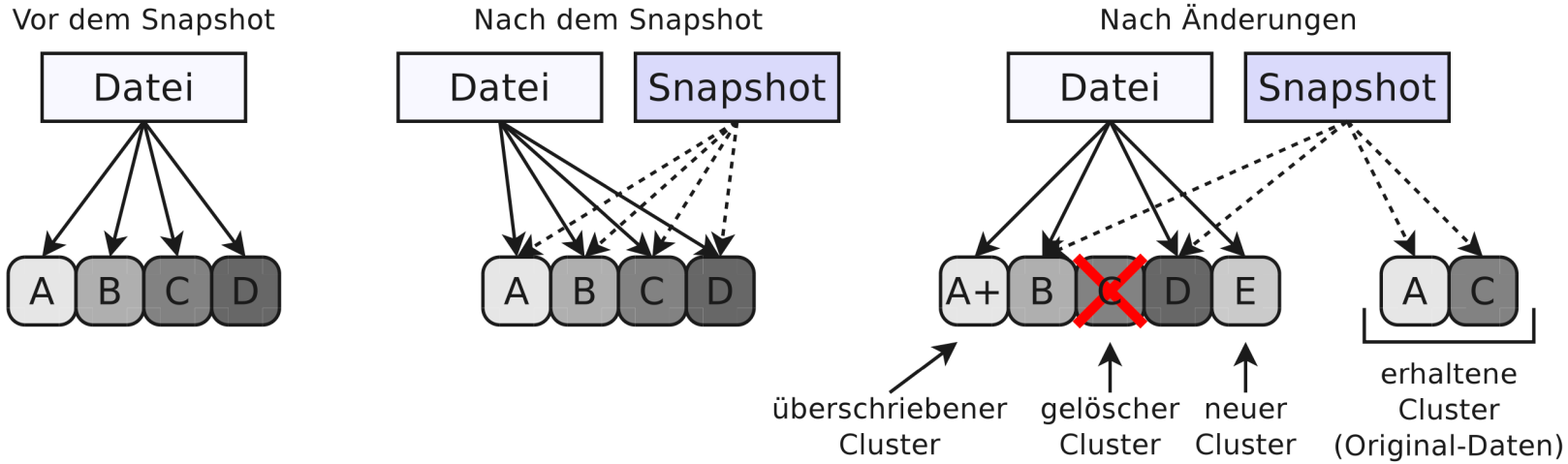
Extents

- **Grundlegende Idee:** Cluster sollen nicht einzeln zugeordnet werden. Stattdessen versucht man zusammenhängende Bereiche zuzuordnen.
- Hierfür müssen drei Werte gespeichert werden:
  - Nummer des Start-Clusters des Bereichs (Extents)
  - Größe des Bereichs in Clustern
  - Nummer des ersten Clusters auf dem Speichergerät

- Ergebnis: Weniger Verwaltungsaufwand
- Beispiele: JFS, XFS, btrfs, NTFS, ext4



Snapshots – Copy on Write



- Schreibzugriffe im Dateisystem ändern/löschen keine Dateiinhalte
  - Veränderte Inhalte werden in freie Cluster geschrieben
  - Anschließend werden die Metadaten auf die neue Datei angepasst
- Ältere Dateiversionen bleiben erhalten und können wiederhergestellt werden
  - Die Datensicherheit ist besser als bei Dateisystemen mit Journal
  - Snapshots können sehr schnell erzeugt werden (nur Metadaten-Änderung)
- Beispiele: ZFS, btrfs und ReFS (Resilient File System)

Universität Bremen

Themenübersicht

- FS als API
- Terminologie
- Management

Grundlagen

- Grundidee
- API-Typen
- Laufzeitsicht
- Dateisysteme registrieren
- Strukturen
  - Superblock
  - Inode
  - Dentry
  - File

VFS

- Minix
- FAT-Dateisysteme
- ext-Dateisysteme
- Journaling
- Extents
- Copy-On-Write

Dateisysteme