Lecture on Storage Systems

File systems under Linux

André Brinkmann

Outline

- File systems under Linux
  - File systems in Unix / Linux
  - Symbolic links
  - Mounting of file systems
- Virtual file system
  - Superblock
  - Inode
  - Dentry object
  - File object
- Example implementation of a file system in Linux based on ext2

Unix file systems

- **Hierarchical**
  - Tree structure
  - File catalogues as internal nodes
  - Files as leaves
  - No restrictions regarding width and depth
- **Consistent**
  - Nearly all system objects are represented as files and can be used via the file interface (files, catalogues, communication objects, devices, ...)
  - Syntactically equal treatment of all types, semantically as far as possible
  - Hence: Applications are independent from the object type
- **Simple**
  - Only a few, but flexible file operations
  - Simple file structure

Files in Unix

- Byte string
- Arbitrarily addressable
- Content has no predefined properties
- Form and content created by user
- Restricted to a single logical medium
- Protected by access rights
  - r (read)
  - w (write)
  - x (execute)
- defined for user, group, others

Inode (Index node)

- Each file is represented by an Inode
- It contains
  - Owner (UID, GID)
  - Access rights
  - Time of last modification / access
  - Size
  - Type (file, directory, device, pipe, ...)
  - Pointers to data blocks that store file's content

Directories (file catalogues)

- Directories are handled as normal files, but are marked in Inode-type as directory
- A directory entry contains
  - Length of the entry
  - Name (variable length up to 255 characters)
  - Inode number
- Multiple directory entries may reference the same Inode number (hard link)
- Users identify files via pathnames ("/path/to/file") that are mapped to Inode numbers by the OS
- If the path starts with "/", it is absolute and is resolved up from the root directory
- Otherwise the path is resolved relative to the current directory

Directories

- Each directory contains an entry "." that represents the Inode of the current directory
- The second entry "..” references parent directory
- The path is resolved from left to right and the respective name is looked up in the directory
- As long as the current name is not the last in the path, it has to be a directory. Otherwise, the lookup terminates with an error

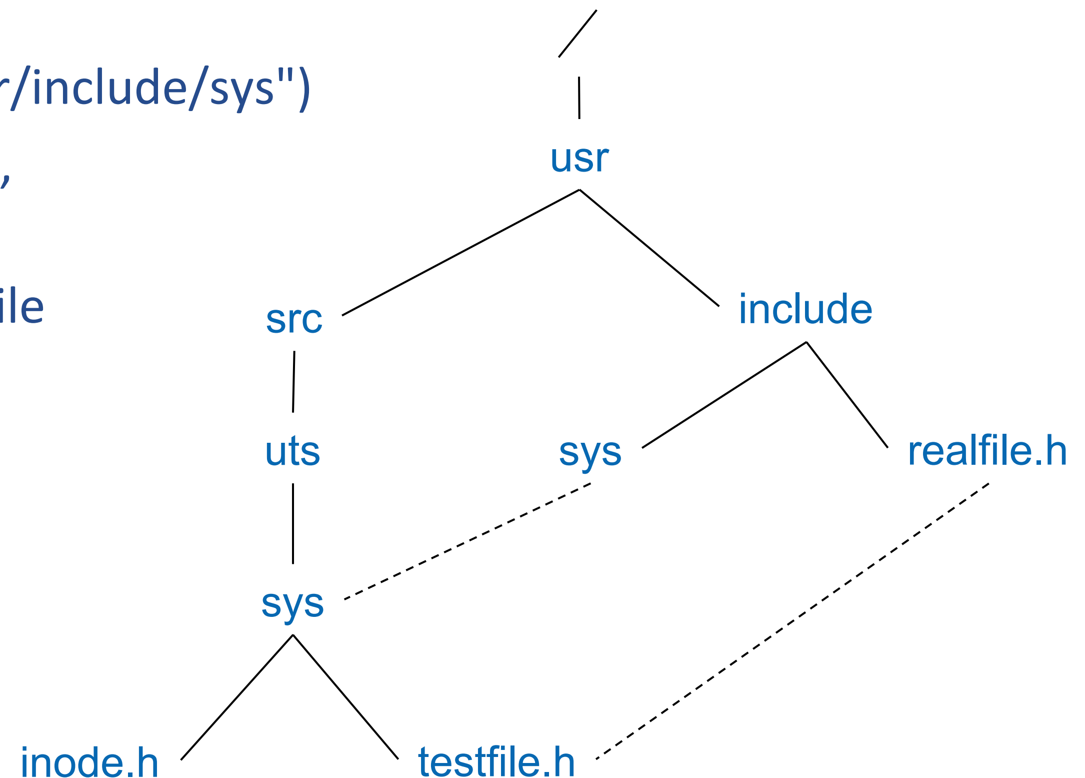
Symbolic Links

- To improve shared access to files, UNIX allows use of symbolic links to reference single files and directories via multiple different paths
- `symlink(bisheriger_name, neuer_name)` creates an additional path to the resource
- Example

  after symlink("/usr/src/uts/sys","/usr/include/sys")

  and symlink("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h")

  there exist three paths to the same file



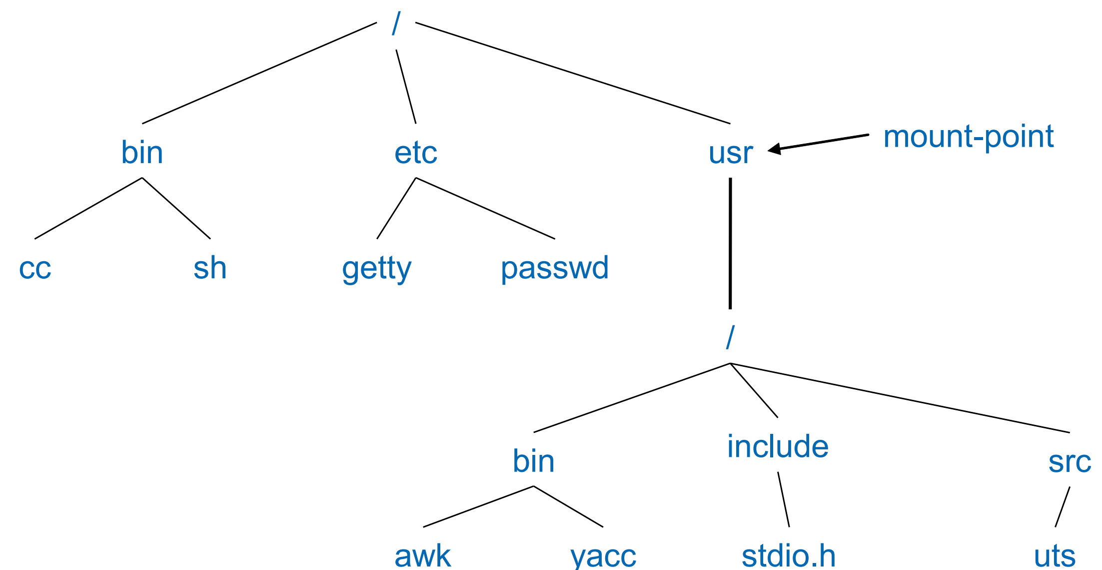
PADERBORN CENTER FOR PARALLEL COMPUTING

Hard and Symbolic Links

- A **hard link** is an additional file name
  - There exists another directory entry that points to the same file
  - All hard links point to the same Inode
  - Each new hard link increments the link counter of the Inode
  - As long as the link counter ≠ 0, the file "survives" a remove() and only the link counter is decremented
  - If the last link is removed, the file is deleted and the Inode can be reused
  - Hard links can only be created for files in the same file logical file system
- A **symbolic link** (soft link) is a file that contains the path of another file or directory
  - Symbolic links are interpreted and resolved on every access
  - If the target of a symbolic link is deleted, the link becomes invalid but remains existent
  - Symbolic links to files and directories can be created for files that do not exist (yet)

Logical and Physical File System

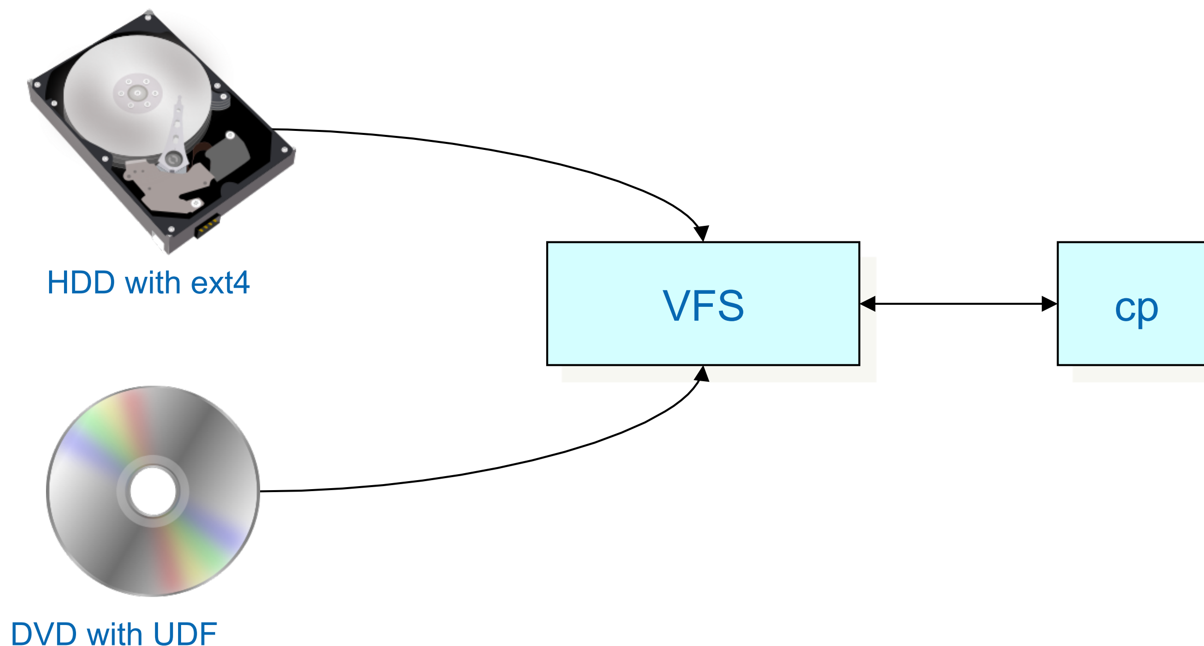
- A logical file system may consist of multiple physical file systems
- A file system can be hooked into any path of the virtual file system tree with the "mount" command
- Mounted file systems are managed by the OS in a "mount table" that connects paths to mount points
- This allows to identify the root Inodes of mounted file systems



Virtual File System

- The Virtual File System (VFS) implements a generic file system interface between the actual file system implementation (in kernel) and accessing applications to provide interoperability

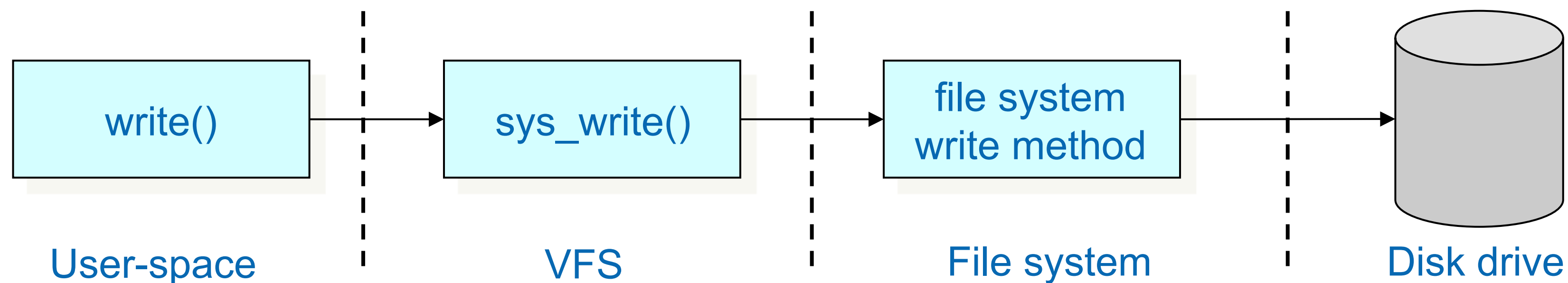
➔ Applications can access different file systems on different media via a homogeneous set of UNIX system calls



Virtual File System

Example: `write(f, &buf, len);`

- Write of `len` Bytes in file with descriptor `f` from Buffer `buf` is translated into system call
- The system call is forwarded to the actual file system implementation
- The file system executes the write command



VFS Objects and Data Structures

- VFS is object oriented
- Four base objects
  - Super block: Represents specific properties of a file system
  - Inode: File description
  - Dentry: The directory entry represents a single component of a path
  - File: Representation of an open file that is associated with a process
- VFS handles directories like files
  - Dentry object represents component of a path that may be a file
  - Directories are handled like files as Inodes
- Each object provides a set of operations

Superblock

- Each file system must provide a superblock
  - Contains properties of the file system
  - Is stored on special sectors of disk or is created dynamically (i.e. by sysfs)
  - Structure is created by `alloc_super()` when the file system is mounted

```
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        dev_t                   s_dev;          /* search index; _not_ kdev_t */
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_dirt;
        unsigned long long      s_maxbytes;     /* Max file size */
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        struct quotactl_ops     *s_qcop;
        struct export_operations *s_export_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        struct rw_semaphore     s_umount;
        struct mutex            s_lock;
        int                     s_count;
        int                     s_syncing;
        int                     s_need_sync_fs;
        atomic_t                s_active;
        void                    *s_security;
        struct xattr_handler    **s_xattr;

        struct list_head        s_inodes;       /* all inodes */
        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_io;           /* parked for writeback */
        struct hlist_head       s_anon;         /* anonymous dentries for (nfs) exporting */
        struct list_head        s_files;

        struct block_device     *s_bdev;
        struct list_head        s_instances;
        struct quota_info       s_dquot;        /* Diskquota specific options */

        unsigned int            s_prunes;       /* protected by dcache_lock */
        wait_queue_head_t       s_wait_prunes;

        int                     s_frozen;
        wait_queue_head_t       s_wait_unfrozen;
        char s_id[32];                          /* Informational name */
        void                    *s_fs_info;     /* Filesystem private info */
        /*
         * The next field is for VFS *only*. No filesystems have any business
         * even looking at it. You had been warned.
         */
        struct semaphore s_vfs_rename_sem;      /* Kludge */
        /* Granuality of c/m/atime in ns.
           Cannot be worse than a second */
        u32                s_time_gran;
};
```

PC²
PADERBORN CENTER FOR PARALLEL COMPUTING

Superblock Operations

```
/*
 * NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
 * without the big kernel lock held in all filesystems.
 */
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);

        void (*read_inode) (struct inode *);

        void (*dirty_inode) (struct inode *);
        int (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct kstatfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);

        int (*show_options)(struct seq_file *, struct vfsmount *);

        ssize_t (*quota_read)(struct super_block *, int, char *,
                size_t, loff_t);
        ssize_t (*quota_write)(struct super_block *, int,
                const char *, size_t, loff_t);
};
```

- Each entry contains pointer to a function
- File system provides implementation for the operations
- Example:
  Write superblock `sb`:

  `sb->s_op->write_super(sb)`

Inode Object

- Contains information specific to a file
- For typical Unix file systems, an Inode can directly be read from disk
- Other file systems hold this information as part of a file or in a database

  ➔ Inode has to be created by the file system
- Special Entries for non-data files
  - i.e. `i_pipe`, `i_bdev`, or `i_cdev` are reserved for pipes, block and character devices
- Some entries are not supported by all file systems and may therefore be set to `Null`

```
struct inode {
        struct hlist_node       i_hash;
        struct list_head        i_list;
        struct list_head        i_sb_list;
        struct list_head        i_dentry;
        unsigned long           i_ino;
        atomic_t                i_count;
        umode_t                 i_mode;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        dev_t                   i_rdev;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        unsigned int            i_blkbits;
        unsigned long           i_blksize;
        unsigned long           i_version;
        unsigned long           i_blocks;
        unsigned short          i_bytes;
        spinlock_t              i_lock;
        struct mutex            i_mutex;
        struct rw_semaphore     i_alloc_sem;
        struct inode_operations *i_op;
        struct file_operations  *i_fop;
        struct super_block      *i_sb;
        struct file_lock        *i_flock;
        struct address_space    *i_mapping;
        struct address_space    i_data;
        struct dquot            *i_dquot[MAXQUOTAS];
        struct list_head        i_devices;
        struct pipe_inode_info  *i_pipe;
        struct block_device     *i_bdev;
        struct cdev             *i_cdev;
        int                     i_cindex;
        __u32                   i_generation;
        unsigned long           i_dnotify_mask;
        struct dnotify_struct   *i_dnotify;
        struct list_head        inotify_watches;
        struct semaphore        inotify_sem;
        unsigned long           i_state;
        unsigned long           dirtied_when;
        unsigned int            i_flags;
        atomic_t                i_writecount;
        void                    *i_security;
        union {
                void            *generic_ip;
        } u;
        seqcount_t              i_size_seqcount;
};
```

Inode Operations

PC²
PADERBORN CENTER FOR PARALLEL COMPUTING

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        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 *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata *);
        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 *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
};
```

- Inode Operations describe the set of operations that are implemented by the file system and are accessed via VFS

Dentry Objects

- Unix directories are handled like files
- The path `/bin/vi` contains the directories `/` and `bin` as well as the file `vi`
- Resolution of paths requires introduction of `dentry` objects
- Each part of a path is `dentry` object
- VFS creates `dentry` objects on the fly
- No equivalent on disk drive
- Are stored in dentry cache (handled by OS)
  - Frontend of Inode cache

```
struct dentry {
        atomic_t d_count;
        unsigned int d_flags;           /* protected by d_lock */
        spinlock_t d_lock;              /* per dentry lock */
        struct inode *d_inode;          /* Where the name belongs to - NULL is
                                         * negative */
        /*
         * The next three fields are touched by __d_lookup.  Place them here
         * so they all fit in a cache line.
         */
        struct hlist_node d_hash;       /* lookup hash list */
        struct dentry *d_parent;        /* parent directory */
        struct qstr d_name;

        struct list_head d_lru;         /* LRU list */
        /*
         * d_child and d_rcu can share memory
         */
        union {
                struct list_head d_child;       /* child of parent list */
                struct rcu_head d_rcu;
        } d_u;
        struct list_head d_subdirs;     /* our children */
        struct list_head d_alias;       /* inode alias list */
        unsigned long d_time;           /* used by d_revalidate */
        struct dentry_operations *d_op;
        struct super_block *d_sb;       /* The root of the dentry tree */
        void *d_fsdata;                 /* fs-specific data */
#ifdef CONFIG_PROFILING
        struct dcookie_struct *d_cookie; /* cookie, if any */
#endif
        int d_mounted;
        unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* small names */
};
```

File Object

- File object represents open file
- Interface to applications
- Is created as reply to `open()` system call
- Is removed on `close()`
- Different processes can open a file multiple times ➔ different file objects
- The file object is an in-memory data structure of the OS

```
struct file {
        union {
                struct list_head        fu_list;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        unsigned int            f_uid, f_gid;
        struct file_ra_state    f_ra;
        unsigned long           f_version;
        void                    *f_security;
        void                    *private_data;
        struct list_head        f_ep_links;
        spinlock_t              f_ep_lock;
        struct address_space    *f_mapping;
};
```

PC²
PADERBORN CENTER FOR PARALLEL COMPUTING

File operations

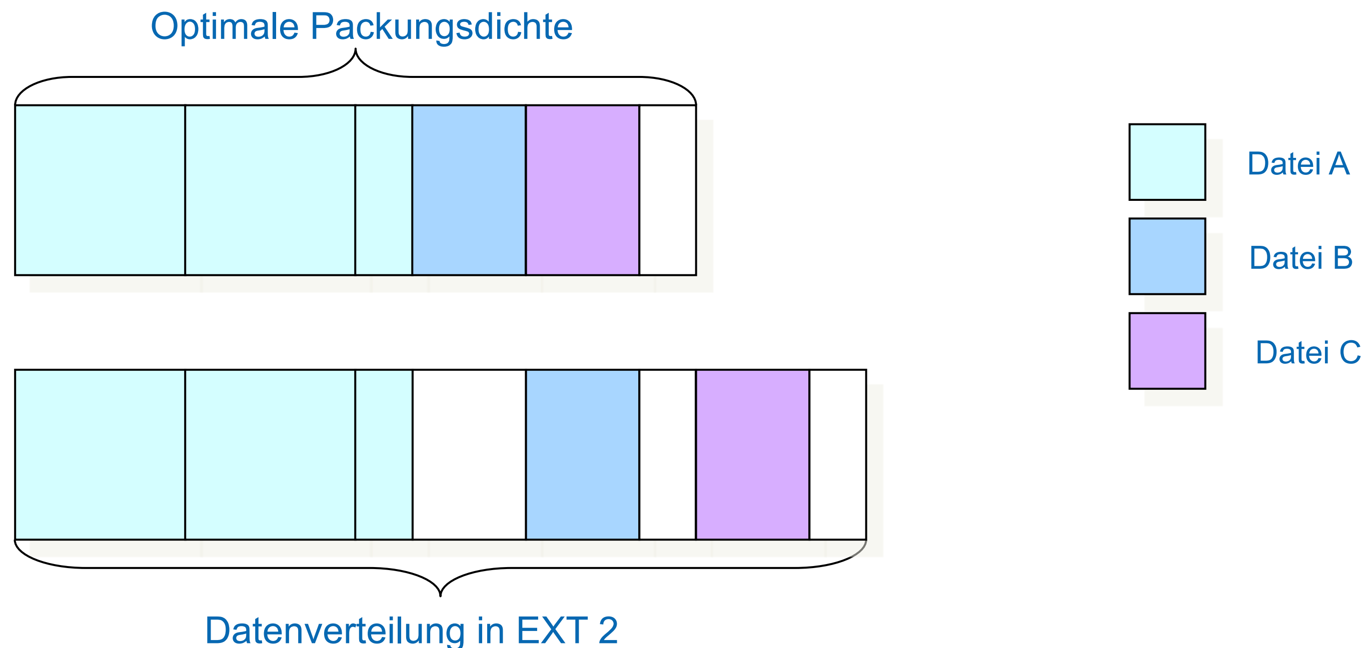
```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev, unlocked_ioctl and compat_ioctl
 * can be called without the big kernel lock held in all filesystems.
 */
struct file_operations {
	struct module *owner;
	loff_t (*llseek) (struct file *, loff_t, int);
	ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
	ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
	ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
	ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
	int (*readdir) (struct file *, void *, filldir_t);
	unsigned int (*poll) (struct file *, struct poll_table_struct *);
	int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
	long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
	long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
	int (*mmap) (struct file *, struct vm_area_struct *);
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, struct dentry *, int datasync);
	int (*aio_fsync) (struct kiocb *, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
	ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
	ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
	ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
	unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
	int (*check_flags)(int);
	int (*dir_notify)(struct file *filp, unsigned long arg);
	int (*flock) (struct file *, int, struct file_lock *);
#define HAVE_FOP_OPEN_EXEC
	int (*open_exec) (struct inode *);
};
```

File systems in Linux - EXT 2

- First linux file system has been derived from Minix
- Limitations of Minix-FS couldn't be tolerated for long
  - Only 14 characters for file names
  - Paritions had to be smaller 64 Mbyte
  - No symbolic links
- Minix-FS has been the file system under linux until the introduction of EXT
- EXT has been influenced by the Fast File System from the BSD world
- Aims of EXT2
  - Variable block sizes to better support big AND small files
  - Fast symbolic links
  - Extending the file system without reformatting
  - Decrease the harm of crashes (`fsck`)
  - Introduction of unchangable files

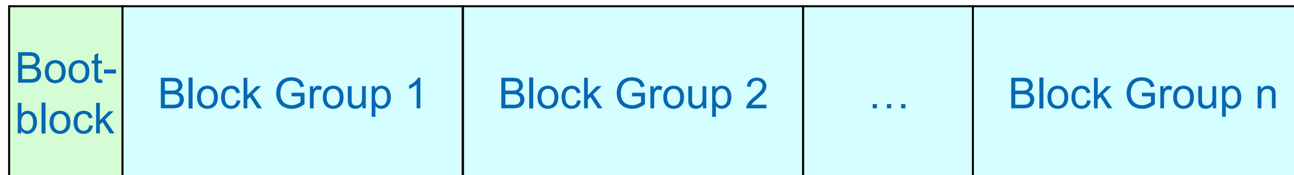
Physical Architecture

- Block based devices have sectors as smallest addressable unit
- EXT2 is block based file system that partitions the hard disk into blocks (clusters) of the same size
- Blocks are used for metadata and data
- Blocks lead to internal fragmentation

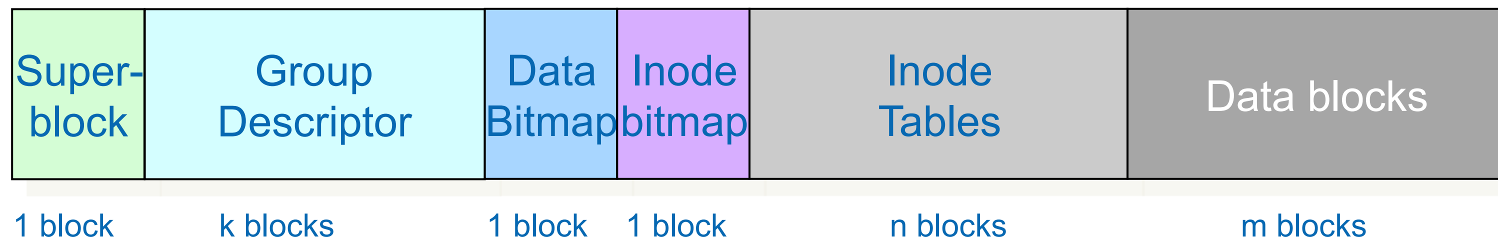


Structural Architecture of EXT 2

- EXT2 divides storage system into block groups



- Boot block is equivalent to first sector on hard disk
- Block group is basic component, which contains further file system components



Metadata

- Superblock: Central structure, which contains number of free and allocated blocks, state of the file system, used block size, …
- Group descriptor contains the state, number of free blocks and inodes in each block group. Each block group contains group descriptor!
- Data bitmap: 1/0 allocation representation for data blocks
- Inode bitmap: 1/0 allocation representation for inode blocks
- Inode table stores all inodes for this block group

- Data blocks store user data

Data Structures

- EXT2 stores metadata in each block group
- Basic idea:
  - If a system crash corrupts the superblock, then there are enough redundant copies of it
  - Distance between metadata and data is small ➔ fewer head movements

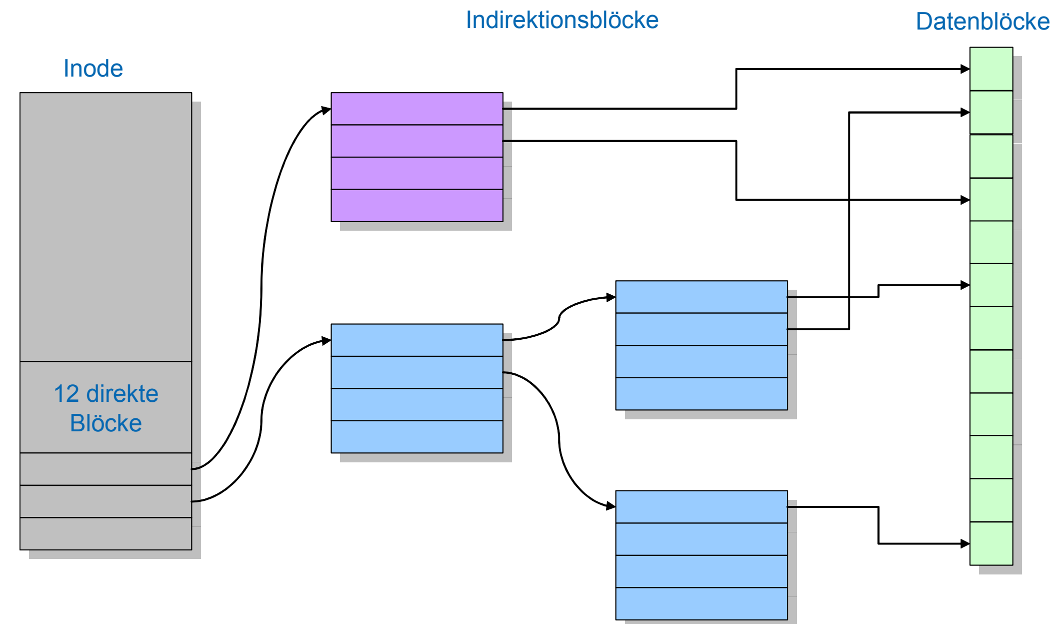
- Implementations work differently:
  - Kernel only works with in RAM copy of the first superblock, which is written back to redundant super blocks during file system checks
  - Later versions of EXT2 include *Sparse Superblock* option, where superblocks are only stored in group 0, 1 as well in groups, which are a power of 3, 5, or 7

Limitations

- Size of a file is restricted by the number of block entries in an inode
- Assumption:
  - 700 Mbyte file size and 4 Kbyte block size
  - ➔ 179.200 block entries are necessary and each entry needs 32 / 64 Bit
  - ➔ 700 KByte storage space within one Inode are necessary

➔ If the inode size is fixed, the you also need 700 KByte inodes for 4 Kbyte files

- EXT2 supports direct and indirect blocks
- There is one pointer each for one-time, tweo-time, and three-time indirect blocks



Calculation of maximum file site

Blockgröße / Kbyte	1	2	4	8
Blöcke erste Stufe	12	12	12	12
Blöcke über erste Indirektion	256	512	1.024	2.048
Blöcke über zweite Indirektion	65.536	262.144	1.048.576	4.194.304
Blöcke über dritte Indirektion	16.777.216	134.217.728	1.073.741.824	8.589.934.592
Maximal darstellbare Zahl	4.294.967.296	4.294.967.296	4.294.967.296	4.294.967.296
max. Dateigröße / Gbyte	16	257	4.100	32.768

- Increasing the block size increases maximum file size quadratically
- High file capacities are mostly ensured by third level of indirection
- Most applications only work with files up to 2 Tbyte
- 64 bit addressing only works with half as much pointers per block, but helps to overcome 2 Tbyte limit

EXT 2-Superblock

- Read in using `ext2_read_super()`
- Meaning is typically part of the name ...
- `s_log_block_size` stores size of a block in Kbyte as its logarithm
- `s_blocks_per_group` defines number of blocks per group
- `s_magic` stores magic value for EXT2
- `s_feature_compat`, ... define compatibility requirements

```
struct ext2_super_block {
        __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
        __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_frag_size;        /* Fragment size */
        __le32  s_blocks_per_group;     /* # Blocks per group */
        __le32  s_frags_per_group;      /* # Fragments per group */
        __le32  s_inodes_per_group;     /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
        __le32  s_wtime;                /* Write time */
        __le16  s_mnt_count;            /* Mount count */
        __le16  s_max_mnt_count;        /* Maximal mount count */
        __le16  s_magic;                /* Magic signature */
        __le16  s_state;                /* File system state */
        __le16  s_errors;               /* Behaviour when detecting errors */
        __le16  s_minor_rev_level;      /* minor revision level */
        __le32  s_lastcheck;            /* time of last check */
        __le32  s_checkinterval;        /* max. time between checks */
        __le32  s_creator_os;           /* OS */
        __le32  s_rev_level;            /* Revision level */
        __le16  s_def_resuid;           /* Default uid for reserved blocks */
        __le16  s_def_resgid;           /* Default gid for reserved blocks */
        __le32  s_first_ino;            /* First non-reserved inode */
        __le16   s_inode_size;          /* size of inode structure */
        __le16  s_block_group_nr;       /* block group # of this superblock */
        __le32  s_feature_compat;       /* compatible feature set */
        __le32  s_feature_incompat;     /* incompatible feature set */
        __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
        __u8    s_uuid[16];             /* 128-bit uuid for volume */
        char    s_volume_name[16];      /* volume name */
        char    s_last_mounted[64];     /* directory where last mounted */
        __le32  s_algorithm_usage_bitmap; /* For compression */
        __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/
        __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
        __u16   s_padding1;
        __u32   s_journal_inum;         /* inode number of journal file */
        __u32   s_journal_dev;          /* device number of journal file */
        __u32   s_last_orphan;          /* start of list of inodes to delete */
        __u32   s_hash_seed[4];         /* HTREE hash seed */
        __u8    s_def_hash_version;     /* Default hash version to use */
        __u8    s_reserved_char_pad;
        __u16   s_reserved_word_pad;
        __le32  s_default_mount_opts;
        __le32  s_first_meta_bg;        /* First metablock block group */
        __u32   s_reserved[190];        /* Padding to the end of the block */
};
```

EXT 2-Superblock-Info

- VFS-superblock gets pointer to file system specific elements via `*s_fs_info`
- Data is read from superblock and/or created from the file system
- Most important:
  - `s_mount_opt`: Mount options
  - `s_mount_state`: Current state
  - `s_dir_count`: Number of directories

```
struct ext2_sb_info {
        unsigned long s_frag_size;      /* Size of a fragment in bytes */
        unsigned long s_frags_per_block;/* Number of fragments per block */
        unsigned long s_inodes_per_block;/* Number of inodes per block */
        unsigned long s_frags_per_group;/* Number of fragments in a group */
        unsigned long s_blocks_per_group;/* Number of blocks in a group */
        unsigned long s_inodes_per_group;/* Number of inodes in a group */
        unsigned long s_itb_per_group;  /* Number of inode table blocks per group */
        unsigned long s_gdb_count;      /* Number of group descriptor blocks */
        unsigned long s_desc_per_block; /* Number of group descriptors per block */
        unsigned long s_groups_count;   /* Number of groups in the fs */
        struct buffer_head * s_sbh;     /* Buffer containing the super block */
        struct ext2_super_block * s_es; /* Pointer to the super block in the buffer */
        struct buffer_head ** s_group_desc;
        unsigned long  s_mount_opt;
        uid_t s_resuid;
        gid_t s_resgid;
        unsigned short s_mount_state;
        unsigned short s_pad;
        int s_addr_per_block_bits;
        int s_desc_per_block_bits;
        int s_inode_size;
        int s_first_ino;
        spinlock_t s_next_gen_lock;
        u32 s_next_generation;
        unsigned long s_dir_count;
        u8 *s_debts;
        struct percpu_counter s_freeblocks_counter;
        struct percpu_counter s_freeinodes_counter;
        struct percpu_counter s_dirs_counter;
        struct blockgroup_lock s_blockgroup_lock;
};
```

EXT 2 Mount Options

```
/*
 * Mount flags
 */
#define EXT2_MOUNT_CHECK             0x000001  /* Do mount-time checks */
#define EXT2_MOUNT_OLDALLOC          0x000002  /* Don't use the new Orlov allocator */
#define EXT2_MOUNT_GRPID             0x000004  /* Create files with directory's group */
#define EXT2_MOUNT_DEBUG             0x000008  /* Some debugging messages */
#define EXT2_MOUNT_ERRORS_CONT       0x000010  /* Continue on errors */
#define EXT2_MOUNT_ERRORS_RO         0x000020  /* Remount fs ro on errors */
#define EXT2_MOUNT_ERRORS_PANIC      0x000040  /* Panic on errors */
#define EXT2_MOUNT_MINIX_DF          0x000080  /* Mimics the Minix statfs */
#define EXT2_MOUNT_NOBH              0x000100  /* No buffer_heads */
#define EXT2_MOUNT_NO_UID32          0x000200  /* Disable 32-bit UIDs */
#define EXT2_MOUNT_XATTR_USER        0x004000  /* Extended user attributes */
#define EXT2_MOUNT_POSIX_ACL         0x008000  /* POSIX Access Control Lists */
#define EXT2_MOUNT_XIP               0x010000  /* Execute in place */
#define EXT2_MOUNT_USRQUOTA          0x020000 /* user quota */
#define EXT2_MOUNT_GRPQUOTA          0x040000 /* group quota */
```

Group descriptor

- One copy of the descriptor for each block group in the kernel
- Block descriptor for each block group in each block group
  - ➔ Bitmaps can be accessed from everywhere
- Pointer to bitmaps with allocation information of blocks and inodes
  - ➔ Number of blocks in each block group restricted by block size
- Position of free blocks can be directly calculated from position in bitmap
- Counter for free structures

```
struct ext2_group_desc
{
        __le32  bg_block_bitmap;
        __le32  bg_inode_bitmap;
        __le32  bg_inode_table;
        __le16  bg_free_blocks_count;
        __le16  bg_free_inodes_count;
        __le16  bg_used_dirs_count;
        __le16  bg_pad;
        __le32  bg_reserved[3];
};
```

EXT2 Inodes

- `i_mode` stores access permissions for and type of file
- Several time stamps
- `i_size` and `i_blocks` store size in bytes, resp. blocks
- `i_block` contains pointer to direct and indirect block links
- `i_links_count` counts hard links

```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_reserved1;
                } linux1;
                struct {
                        __le32  h_i_translator;
                } hurd1;
                struct {
                        __le32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
        union {
                struct {
                        __u8    l_i_frag;       /* Fragment number */
                        __u8    l_i_fsize;      /* Fragment size */
                        __u16   i_pad1;
                        __le16  l_i_uid_high;   /* these 2 fields    */
                        __le16  l_i_gid_high;   /* were reserved2[0] */
                        __u32   l_i_reserved2;
                } linux2;
                struct {
                        .....
                } hurd2;
                struct {
                       ....
                } masix2;
        } osd2;                         /* OS dependent 2 */
};
```

How does OS find an Inode?

```
static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
                                         struct buffer_head **p)
{
        struct buffer_head * bh;
        unsigned long block_group;
        unsigned long block;
        unsigned long offset;
        struct ext2_group_desc * gdp;

        *p = NULL;
        if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
            ino > le32_to_cpu(EXT2_SB(sb)->s_es->s_inodes_count))
                goto Einval;

        block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
        gdp = ext2_get_group_desc(sb, block_group, &bh);
        if (!gdp)
                goto Egdp;
        /*
         * Figure out the offset within the block group inode table
         */
        offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) * EXT2_INODE_SIZE(sb);
        block = le32_to_cpu(gdp->bg_inode_table) +
                (offset >> EXT2_BLOCK_SIZE_BITS(sb));
        if (!(bh = sb_bread(sb, block)))
                goto Eio;

        *p = bh;
        offset &= (EXT2_BLOCK_SIZE(sb) - 1);
        return (struct ext2_inode *) (bh->b_data + offset);

Einval:
        ext2_error(sb, "ext2_get_inode", "bad inode number: %lu",
                   (unsigned long) ino);
        return ERR_PTR(-EINVAL);
Eio:
        ext2_error(sb, "ext2_get_inode",
                   "unable to read inode block - inode=%lu, block=%lu",
                   (unsigned long) ino, block);
Egdp:
        return ERR_PTR(-EIO);
}
```

Is it a valid Inode address?

In which group resides Inode

Information about the group

Offset within the group

Read data from disk / from Cache

Directory entries in EXT2

- Directories are handled as standard inodes
- `ext2_dir_entry` marks directory entry
- `Inode` contains associated inode number
- `name_len` stores length of directory name
  - Has to be multiple of four
  - Can be filled with `/0`
- `rec_len` points to next entry

```
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

Directory entries in EXT2

inode	rec_len	name_len	file_type	name							
	12	1	2	.	\0	\0	\0				
	12	2	2	.	.	\0	\0				
	16	8	4	h	a	r	d	d	i	s	k
	32	5	7	l	i	n	u	x	\0	\0	\0
	16	6	2	d	e	l	d	i	r	\0	\0
	16	6	1	s	a	m	p	l	e	\0	\0
	16	7	2	s	o	u	r	c	e	\0	\0

Corresponds to the following directory:

```
drwxr—xr-x    3 brinkman users     4096  Dec 10 19:44 .
drwxrwxrwx   13 brinkman users     8192  Dec 10 19:44 ..
brw-r—r--     1 brinkman users    3,  0  Dec 10 19:44 harddisk
lrwxrwxrwx    1 brinkman users       14  Dec 10 19:44 linux->/usr/src/linux
-rw-r--r--    1 brinkman users       13  Dec 10 19:44 sample
drwxr-xr-x    2 brinkman users     4096  Dec 10 19:44 source
```

How does the os find a file?

Example: Opening the file `/home/user/.profile`:

- `/` is always stored in Inode 2 of the root file system
  - (Exception: Process was `chroot'ed`)
- Open Inode 2, read data of Inode, lookup entry `home` and read its inode number
- Open Inode for `home`, read its data, lookup entry for `user` and read its inode number
- Open Inode for `user`, read its data, lookup entry for `.profile` and read its inode number
- Open Inode for `.profile`, read its data, create a `struct file`
- A pointer to the file is added to the file pointer table of the OS

➔ The file descriptor table of the calling process is updated with the new pointer

Allocation of data blocks

- Allocation of data blocks always necessary if the file becomes bigger
- Aim: Map successive addresses sequentially to the storage system
- Approach of `ex2_get_block()`
  - If there is a logical block directly before address of current block ➔ take next physical block
  - Else take physical block number of the block with the logical block number directly before the logical block number of the current block
  - Else take block number of first block in block group, where inode is stored
- Target block can be already occupied
  - Task of `ext2_alloc_branch()`: Allocate nearby block based on `goal`-block
- `ext2_alloc_block()` includes options for the preallocation of blocks
- Orlov-Allokator: typically no relationship between subdirectories in root directory ➔ if there a new subdirectory is created in the root directory, just place it somewhere

Journaling File Systems

- "A journaling file system is a file system that logs changes to a journal (usually a circular log in a specially-allocated area) before actually writing them to the main file system"
- Problem description without Journaling:
  - A crashed computer or file system might lead to inconsistent data on a file system
  - Full file system needs to be checked and repaired
  - ➔ This process might take multiple hours!

➔ Idea:
  - Write all data to a journal first, then to its final destination on disk
  - On a crash, only the journal has to be checked for unfinished transactions
  - Operations can be executed atomically

Slide based on Wikipedia

Journaling File Systems

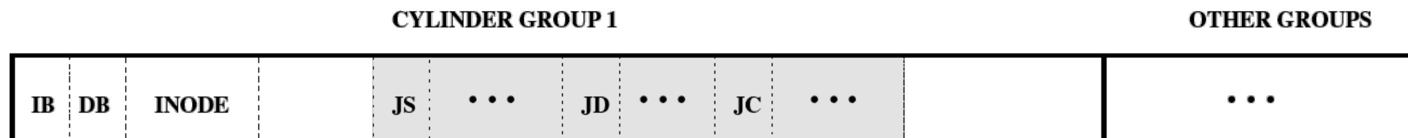
- „Full Journaling“ writes all data twice
  ➔ degraded performance
- Idea of „Metadata Journaling“:
  – Only write metadata of a file to the journal, actual file data is directly written to disk
- File data should be written before the metadata is committed to the journal to prevent file inconsistencies
- Example
  1. Resize file in Inode
  2. Allocate space for file extension in the free space map
  3. Write data to the newly allocated area

> What happens if the computer crashes after step 2?

EXT 3 Journaling File System

PADERBORN CENTER FOR PARALLEL COMPUTING

- EXT 3 extends EXT 2 by journaling
- Journal is stored as a file on the file system but may also be stored on a separate partition
- Journal is implemented as a ring buffer. If the operations are committed to disk, the journal is reused



IB = Inode Bitmap, DB = Data Bitmap, JS = Journal Superblock, JD = Journal Descriptor Block, JC = Journal Commit Block

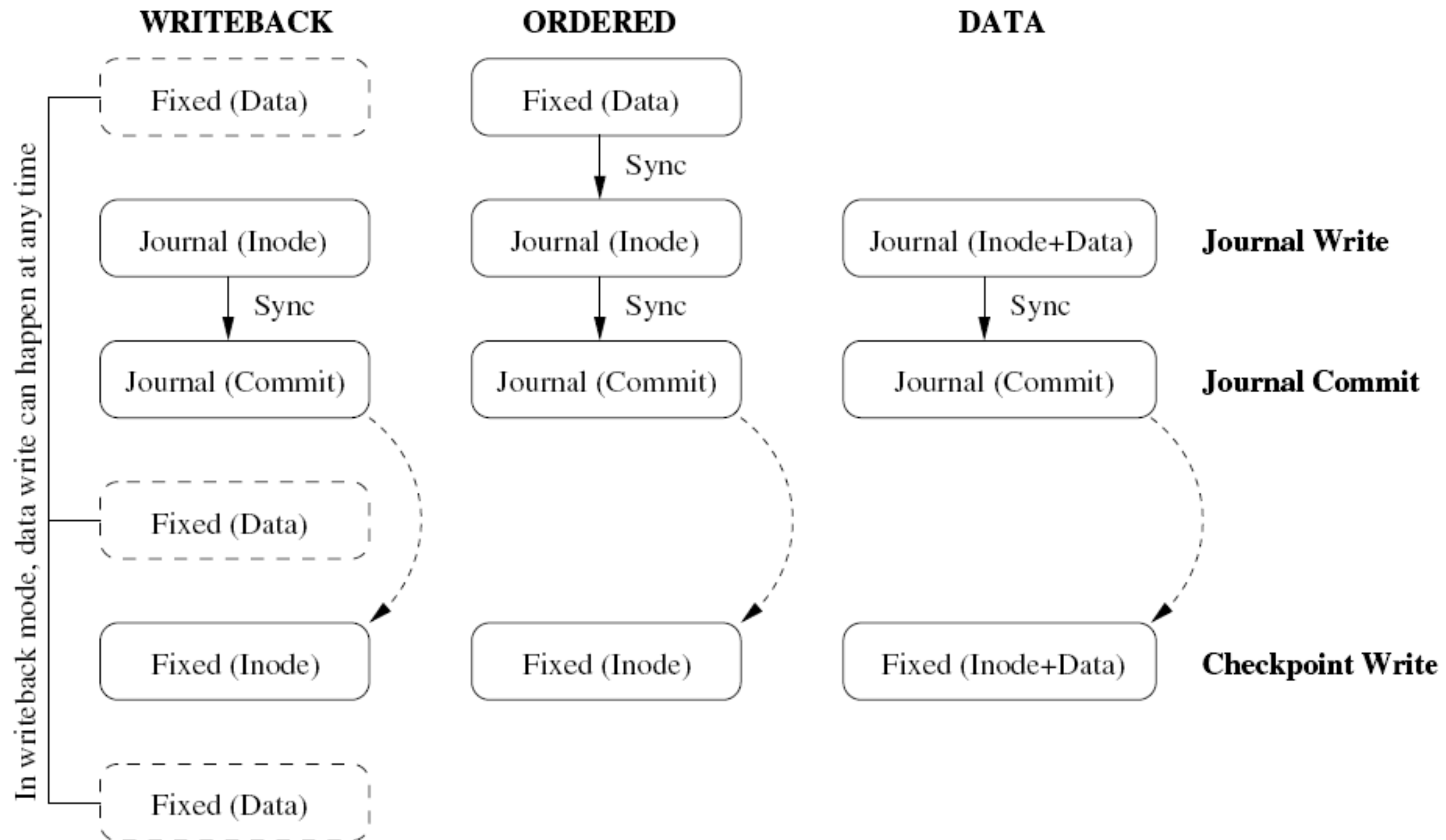
- Journal superblock stores information like block size and pointers to the beginning and the end of the journal
- Journal descriptor block marks the beginning of a transaction and contains information about following blocks, i.e. their storage location
- Journal commit block is written to the end of a transaction. If the JCB was written, the transaction can be recovered without data loss

EXT 3 Journaling Modes

EXT 3 provides three different journaling modes

- Write-back
  - Only the metadata is written to the journal
  - Data blocks are directly written to disk
  - No clear ordering of writes of data blocks or the journal
  - A crash may lead to an inconsistent state
- Ordered
  - Only metadata is written to the journal
  - Data blocks are written to disk, before metadata is written to the journal
  - If metadata write commits, then the data is consistent after crash
- Data
  - Data and metadata are written to the journal

EXT 3 Journaling Modes



EXT 3 Transactions

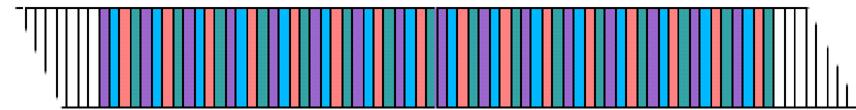
- Transactions
  - EXT 3 groups multiple file system updates into a single transaction
  - Goal: Performance improvement if a structure is updated multiple times
  - Example: Free space bitmap is updated regularly
- Checkpointing
  - Flushes journaling information to be written to their destination blocks
  - Triggered by a timer, if file system buffers become to small, or the journal reaches its maximum size

Reservation of Blocks in EXT3

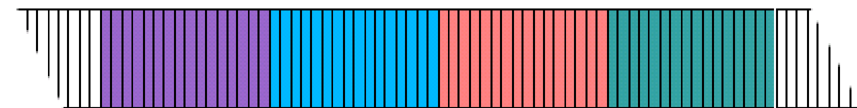
- Pre-allocation of blocks helps to reduce fragmentation of files
- EXT3 supports reservation of areas in the main memory
- Each Inode has its own reservation window
  - Windows do not overlap
  - Windows can grow and shrink dynamically
  - Windows are removed if the file is closed

➔ Improved throughput

Ext3 (before)



Ext3 (After)



file 1   file 2   file 3   file 4

PC²
PADERBORN CENTER FOR PARALLEL COMPUTING

Reservation of Blocks in EXT3