Linux File System & Operations

Dr. Vimal Baghel, Assistant Professor
SCSET, BU

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

Q & A

Linux File System

- A file is an ***object that stores data, information, settings or commands*** in a computer system.
- A directory is a structure that is used ***to store files & subdir.***
- A filesystem is ***the methods and data structures that an OS uses to keep track of files on a disk or partition;*** that is, the way the files are organized on the disk.
- The file system is a data structure to store the meta-data of files.
- In Linux, the files are organized in a tree structure.

Linux File System

- A file system is a data structure that stores index of each file.
- The files reside in physical form of data blocks in the hard disk.
- The FS is responsible to map the physical & its logical location in directory structure.

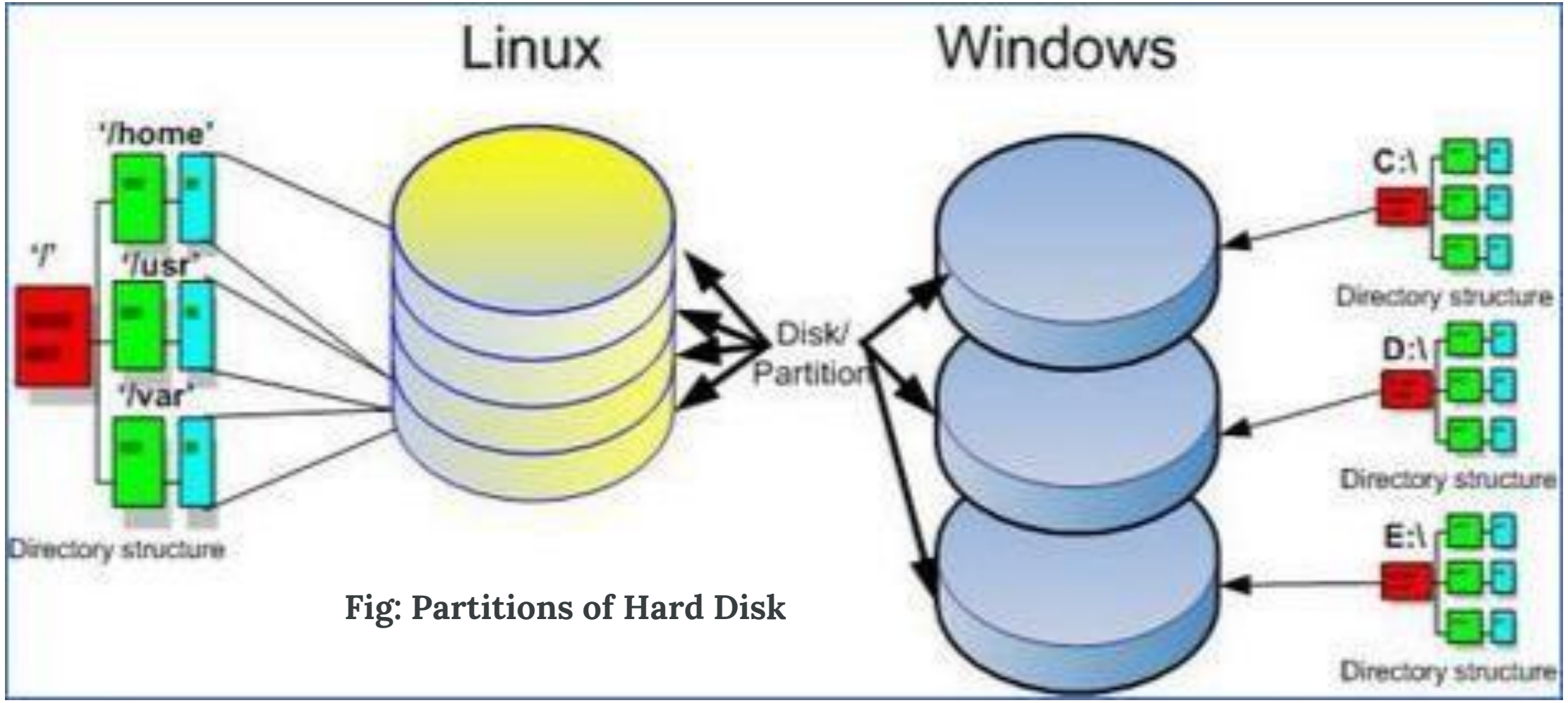


Fig: Partitions of Hard Disk

file systems supported by Linux

BENNETT UNIVERSITY
THE TIMES GROUP

File System	Max File Size	Max Partition Size	Notes
Fat16	2 GiB	2 GiB	Legacy
Fat32	4 GiB	8 TiB	Legacy
NTFS	2 TiB	256 TiB	(For Windows Compatibility)
ext2	2 TiB	32 TiB	Legacy
ext3	2 TiB	32 TiB	Standard linux filesystem for many years. Best choice for super-standard installation.
ext4	16 TiB	1 EiB	Modern iteration of ext3. Best choice for new installations where super-standard isn't necessary.
reiserFS	8 TiB	16 TiB	No longer well-maintained.
JFS	4PiB	32PiB	Created by IBM
XFS	8 EiB	8 EiB	Created by SGI

Linux Directory Structure

BENNETT UNIVERSITY
THE TIMES GROUP

- Linux classifies the files into 3 main categories:
  - User Files – Files created and being accessed by users of the system
  - System Files – Executable files, binary files, configuration files, etc.
  - Device Files – Files corresponding to devices like sound card, graphics card, NIC, etc.

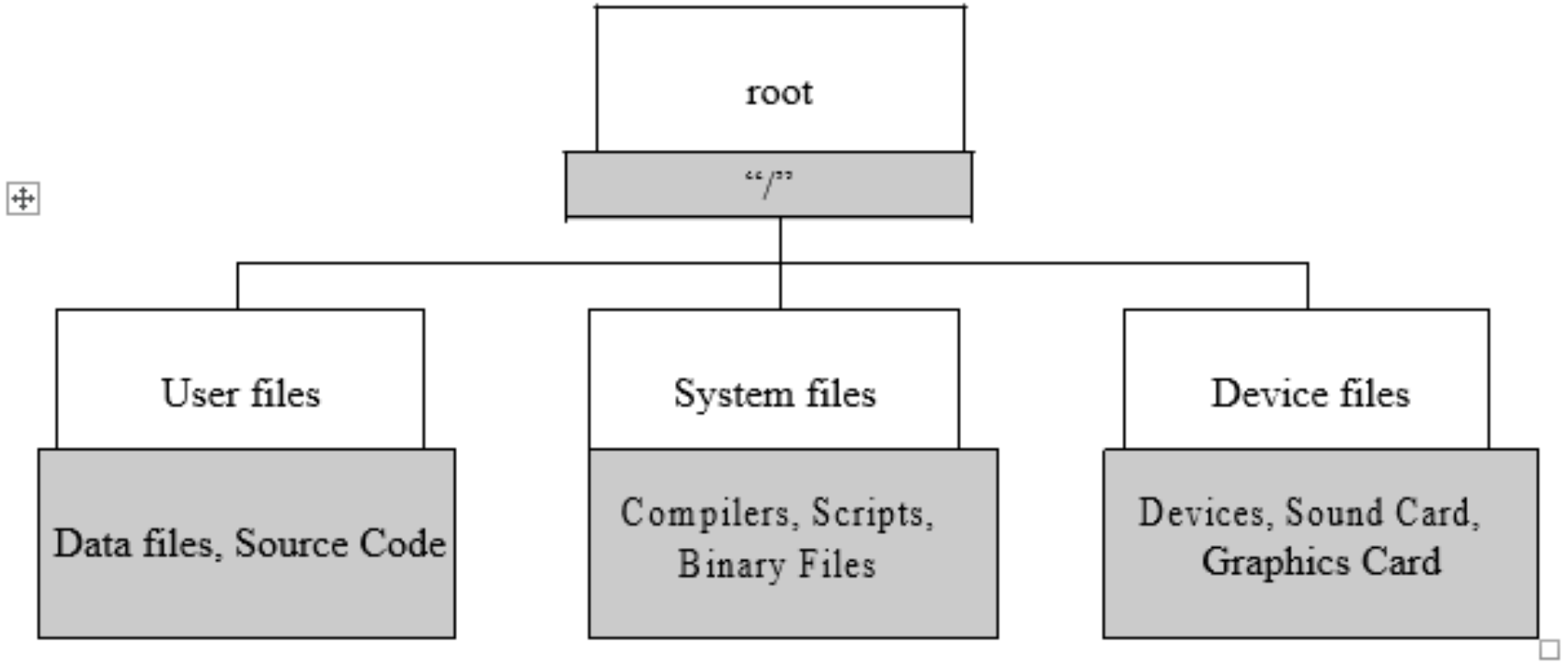


Figure 2: Linux directories & files

Mount Point

BENNETT UNIVERSITY
THE TIMES GROUP

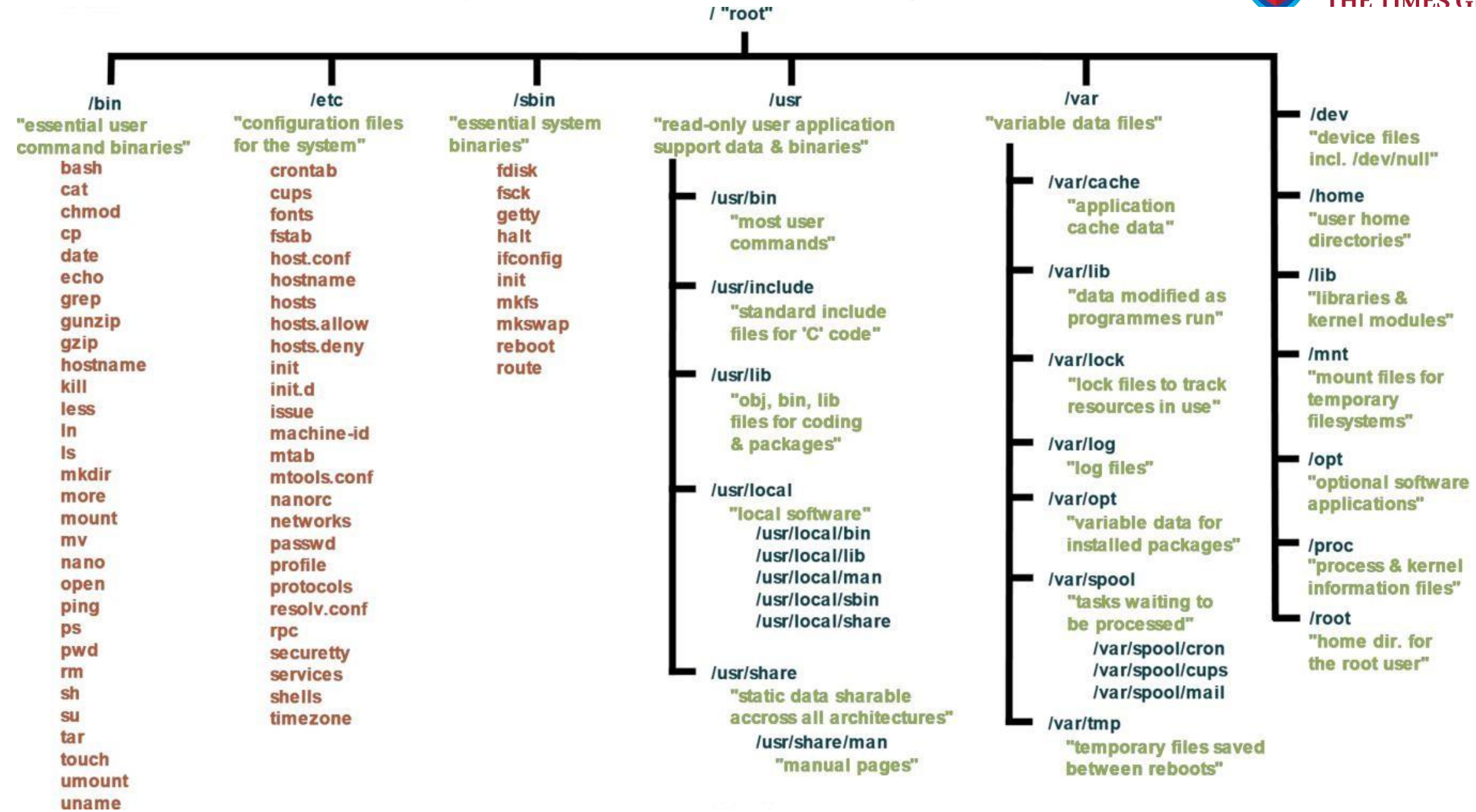
- A mount point is a directory in a file system where additional information is logically connected from a storage location outside the operating system's root drive and partition.

```
$ mount /dev/sda6 /home
```

```
$ umount /home
```

Linux Directory and Files Organization

BENNETT UNIVERSITY
THE TIMES GROUP



Linux Files

- Linux files are classified into the following three general categories depending on the content and usage of file:
  - Regular files (-): randomly addressable sequence of bytes
  - Directory files (d): Contains files & other dir
  - Device files or special files: a point of interface to one of the computer's hardware devices.
    - Block files (b)
    - Character device files (c)
    - Named Pipe file (p)
    - Symbolic Link (l)
    - Socket files (s)
- ***a device file acts as a communication channel between two or more cooperating programs.***

Types of Linux Commands

BENNETT UNIVERSITY
THE TIMES GROUP

External Commands:

- Commands with an independent existence in the form of a separate file.
- For example, programs for the commands such as **cat and Is,** exist independently in a directory called the /bin.
- When such commands are given, the shell reaches these command files with the **help of a system variable called the PATH variable** and executes them.
- Mostly Linux commands are external.

Internal/built-in Commands:

- Commands that are built into the shell.
- For example, the echo command is an internal command as its routine will be a part of the shell's routine.
- **cd and mkdir**, are two examples of internal commands.

File Operations

Performing basic file operations

cp *file1 file2*	copy file1 and call it file2
mv *file1 file2*	move or rename file1 to file2
rm *file*	remove a file
rmdir *directory*	remove a directory
cat *file*	display a file
more *file*	display a file a page at a time
head *file*	display the first few lines of a file
tail *file*	display the last few lines of a file
less *file1*	Display/search a keyword in the file1
grep *'keyword' file*	search a file for keywords
wc *file*	count number of lines/words/characters in file

Advanced File Operations

BENNETT UNIVERSITY
THE TIMES GROUP

- **Copying files remotely:** **`scp [option] user1@host1:source user2@host2:destination`**
- **Comparing files:** **`diff [option] file1 file2`**
- **Finding files:** `find search_path [option]`

- **Searching files according to use case:**

```
#!/bin/bash
# Filename: finding_files.sh
echo -n "Number of C/C++ header files in system: "
find / -name "*.h" 2>/dev/null | wc -l
echo -n "Number of shell script files in system: "
find / -name "*.sh" 2>/dev/null | wc -l
echo "Files owned by user who is running the script ..."
echo -n "Number of files owned by user $USER :"
find / -user $USER 2>/dev/null | wc -l
echo -n "Number of executable files in system: "
find / -executable 2>/dev/null | wc -l
```

- **Finding and deleting a file based on inode number:**

```
$ find ~/ -inum 8159146 -exec rm -i {} \;
```

Links to a file:

- A soft link or a symbolic link
- A hard link

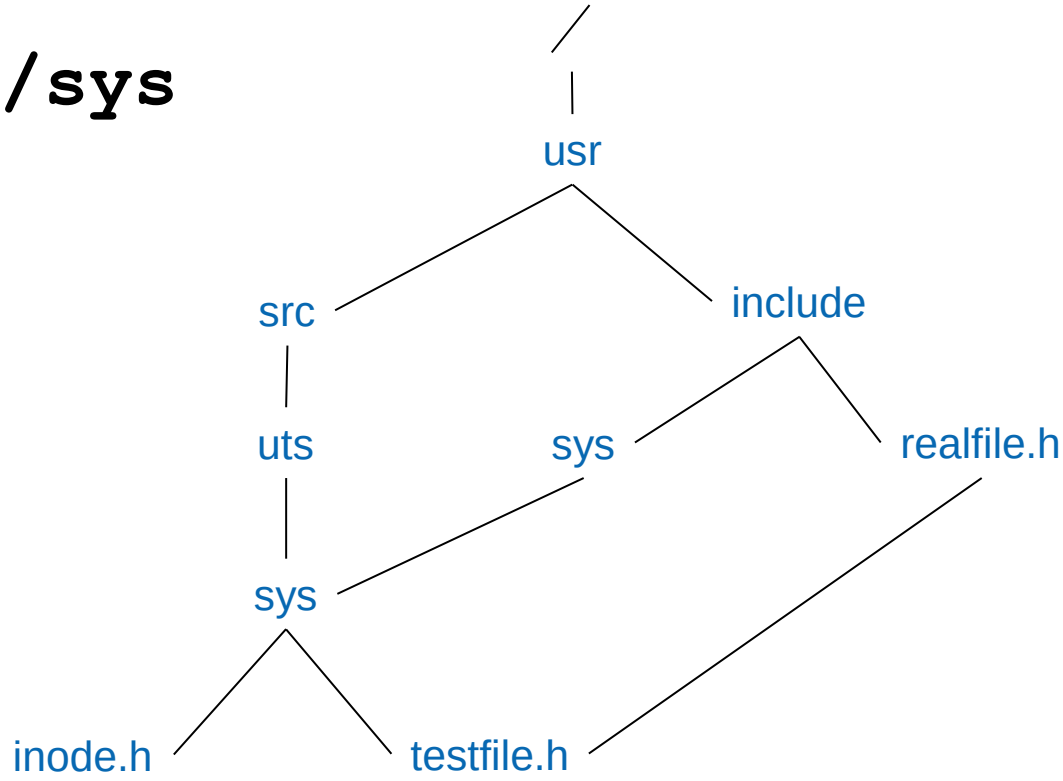
```
ln [option] target link_name
```

Soft or Symbolic Links

To know whether a file is a symbolic link or not, run `ls -l` on a file:

```
$ ls -l ~/tmp
lrwxrwxrwx. 1 foo foo 5 Aug 23 23:31 /home/foo/tmp -> /tmp/
```

```
$ ln -s /usr/src/uts/sys ~/sys
```



Hard Links

- A hard link is a way to refer a file with different names. All such files will have the same inode number.

- To create a hard link of a file, use the `ln` command without any option.

```
$ touch file.txt
$ ls -l file.txt
-rw-rw-r--. 1 foo foo 0 Aug 24 00:13 file.txt
```

- to create a hard link of `file.txt`

```
$ ln file.txt hard_link_file.txt
```

- To check whether a hard link is created for `file.txt`

```
$ ls -l file.txt
-rw-rw-r--. 2 foo foo 0 Aug 24 00:13 file.txt
```

```
$ ls -i file.txt hard_link_file.txt
96844 file.txt
96844 hard_link_file.txt
```

Difference between hard link and soft link

Soft link	Hard link
The inode number of the actual file and the soft link file are different.	The inode number of the actual file and the hard link file are the same.
A soft link can be created across different filesystems.	A hard link can only be created in the same filesystem.
A soft link can link to both regular files and directories.	A hard link doesn't link to directories.
Soft links are not updated if the actual file is deleted. It keeps pointing to a nonexistent file.	Hard links are always updated if the actual file is moved or deleted.

Special files

- The block device file
- The character device file
- The named pipe file
- The socket file

The block device file

- A block device file is a file that reads and writes data in block.
- Such files are useful when data needs to be written in bulk.
- Devices such as hard disk drive, USB drive, and CD-ROM are considered as block device files.
- Data is written asynchronously and, hence, other users are not blocked to perform the write operation at the same time.

The block device file

- To create a block device file, `mknod` is used with the option `b` along with providing a major and minor number.
- A major number selects which device driver is being called to perform the input and output operation.
- A minor number is used to identify subdevices:

```
$ sudo mknod block_device b 0X7 0X6
```

minor number

major number

it is a block device file

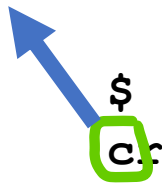
```
$ ls -l block_device
brw-r--r--. 1 root root 7, 6 Aug 24 12:21 block_device
```

Character Device File

- A character device file is a file that reads and writes data in character-by-character fashion.
- Such devices are synchronous and only one user can do the write operation at a time.
- Devices such as keyboard, printer, and mouse are known as character device files.
- Following command will create a character special file:

```
$ sudo mknod character_device c 0X78 0X60
```

it is a character device file



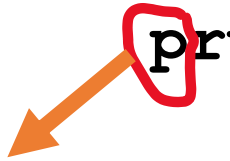
```
$ ls -l character_device # viewing attribute of character_device file
crw-r--r--. 1 root root 120, 96 Aug 24 12:21 character_device
```

Named pipe file

- Named pipe files are used by different system processes to communicate with each other.
- Such communication is also known as inter-process communication(IPC).

- To create such a file, we use the mkfifo command:

```
$ mkfifo pipe_file # Pipe file created
$ ls pipe_file # Viewing file content
prw-rw-r--. 1 foo foo 0 Aug 24 01:41 pipe_file
```



it is a pipe file

- We can also create a named pipe using the mknod command with the p option:

```
$ mknod named_pipe_file p
$ ls -l named_pipe_file
prw-rw-r--. 1 foo foo 0 Aug 24 12:33 named_pipe_file
```

Script to send and receive a message over/from pipe file


```
#!/bin/bash
#Filename: send.sh
#Script which sends message over pipe
pipe=/tmp/named_pipe
if [[ ! -p $pipe ]]
then
mkfifo $pipe
fi
echo "Hello message from Sender">$pipe
```

```
#!/bin/bash
#Filename: receive.sh
#Script receiving message from sender from
pipe file
pipe=/tmp/named_pipe
if [[ ! -p $pipe ]]
then
echo "Reader is not running"
fi
while read line
do
echo "Message from Sender:"
echo $line
done < $pipe
```

To execute it, run `send.sh` in a terminal and `receive.sh` in another terminal:

```
$ sh send.sh # In first terminal
$ sh receive.sh # In second terminal
Message from Sender:
Hello message from Sender
```

Socket file

- A socket file is used to pass information from one application to another.
- For example, if **Common UNIX Printing System** (**CUPS**) daemon is running and my printing application wants to communicate with it, then my printing application will write a request to a socket file where CUPS daemon is listening for upcoming requests.
- Once a request is written to a socket file, the daemon will serve the request:

```
$ ls -l /run/cups/cups.sock # Viewing socket file attributes
srw-rw-rw-. 1 root root 0 Aug 23 15:39 /run/cups/cups.sock
```

it is a socket file

Temporary files

- Such files are being used to keep intermediate results of running a program and they are no longer needed after the program execution is complete.

Creating a temporary file using mktemp:

```
$ mktmp
/tmp/tmp.xEXXxYeRcF
```

✓The `mktemp` command creates a temporary file and prints its name on `stdout`.

✓Temporary files are created by default in the `/tmp` directory.

inode (Index node)

BENNETT UNIVERSITY
THE TIMES GROUP

- 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

BENNETT UNIVERSITY
THE TIMES GROUP

Each directory contains an entry "." that represents the Inode of the current directory

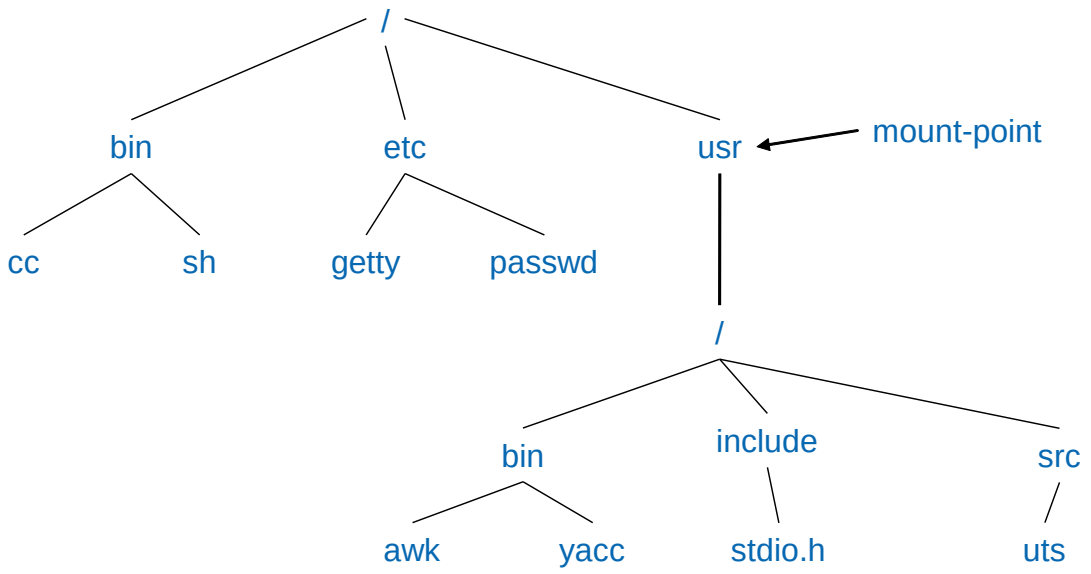
The second entry ".." references parent directory

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

Logical and Physical File System

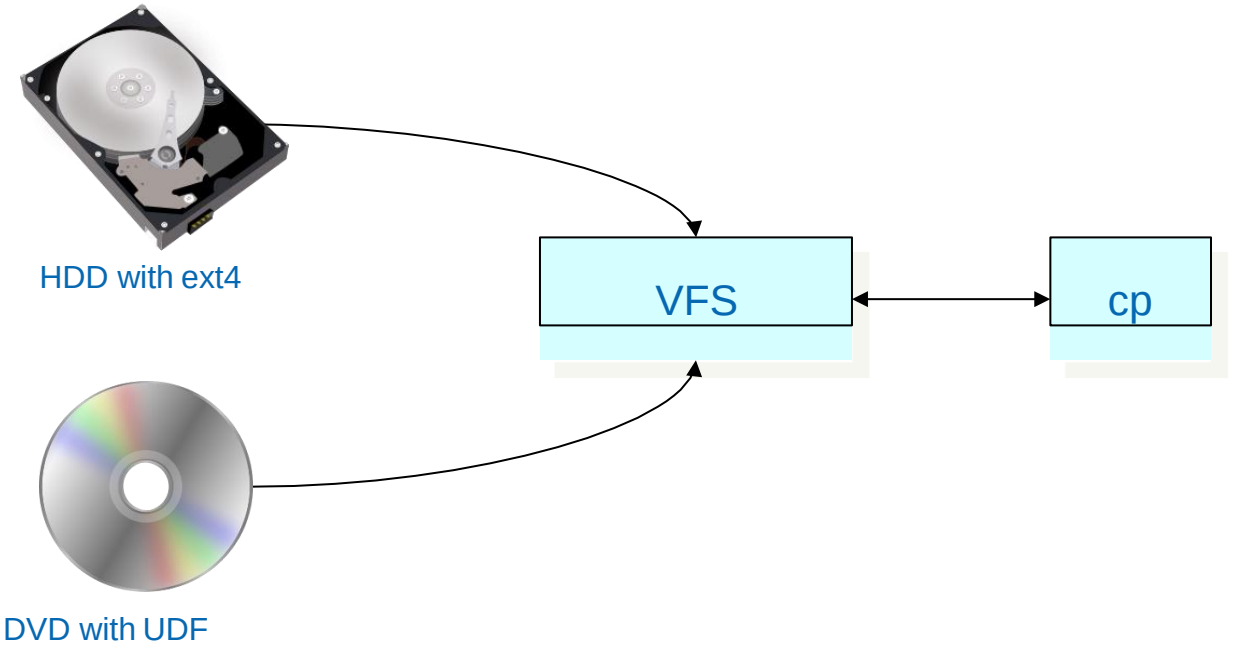
- A logical file system may consist of multiple physical file systems
- A file system can be hooked into any path of the virtual file system tree with the "mount" command
- Mounted file systems are managed by the OS in a "mount table" that connects paths to mount points
- This allows to identify the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

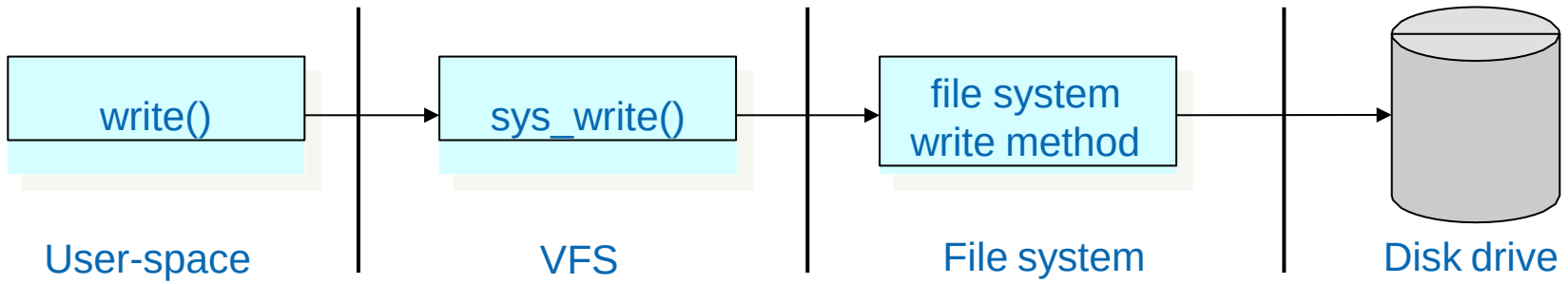
➔➔Applications can access different file systems on different media via a homogeneous set of UNIX system calls



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;
};
```

Superblock Operations

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

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

```
/*
 * 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);
};
```

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 or in a database

  ➔➔Inode has to be created by the file system
- Special Entries for non-datafiles
  - i.e. `i_pipe`, `i_bdev`, or `i_cdev` are reserve 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

```
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

Directories 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;
};
```

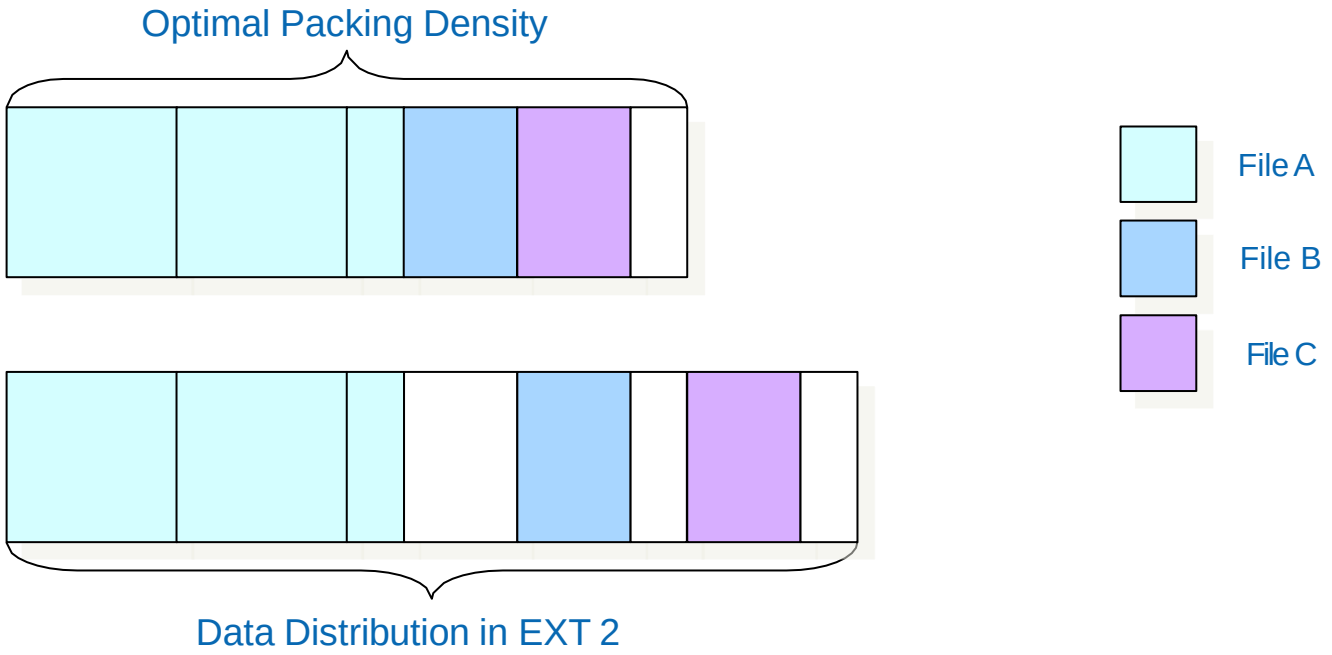
File operations

BENNETT UNIVERSITY
THE TIMES GROUP

```
/*
 * 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 *);
};
```

Physical Architecture

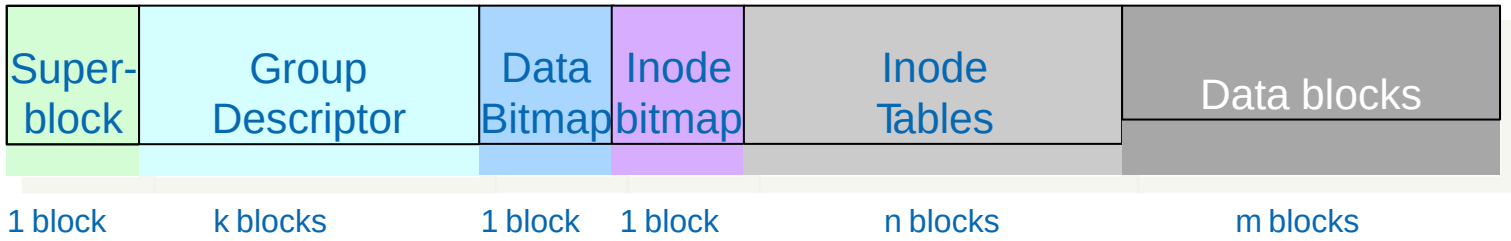
- Block based devices have sectors as smallest addressable unit
- EXT2 is block based file system that partitions the hard disk into blocks (clusters) of the same size
- Blocks are used for metadata and data
- Blocks lead to internal fragmentation



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

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

BENNETT UNIVERSITY
THE TIMES GROUP

- `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?

BENNETT UNIVERSITY
THE TIMES GROUP

```
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

BENNETT UNIVERSITY
THE TIMES GROUP

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

BENNETT UNIVERSITY
TIMES OF INDIA GROUP

Thanks

Q & A