

Course  
TÖL401G: Stýrikerfi /  
Operating Systems  
10. File-System Interface

---

# Chapter Objectives

---

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Understand memory-mapped files.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore briefly file-system protection.

# Contents

---

1. Introduction: File Concept
2. Operations for Accessing Files
3. Concept & Structure of Directories
4. File-System Mounting
5. Shared File Access and Protection
6. Summary

Note for users of the Silberschatz et al. book: newer editions have the material ordered differently than on my slides.

# 10.1 Introduction: File Concept

---

- For storing data, an application program or its developer should not have to deal with physical details of a storage device.
  - Chaos would occur if all the applications would write directly to a storage device: application A would not be aware of application B's data and might just overwrite it.
    - Organised access to data on a storage device required.

⇒ Operating systems offer support for **files**.

- Store and access of data using **symbolic names**.
  - Application needs not to know about internal physical organisation of files on storage device – just use the file name.

# File Concept: File Attributes

---

- In addition to the actual file contents, **file attributes** are required for managing files:
  - **Name** (Many Unixes: 255 character maximum length),
  - **Location** where file contents is stored on storage device,
  - **Size** (in bytes)
- Additional (typically used, but not necessarily required) file attributes:
  - Date (e.g. file creation; last write, last read access),
  - Owning user, owning group of users,
  - Protection information.

# 10.2 Operations for Accessing Files

---

- POSIX operating systems:

- Open an existing file:

`int open(const char *pathname, int flags)`

- Create new file/Overwrite an existing file and open it in write mode:

`int creat(const char *pathname, mode_t mode)`

- Return value: File descriptor, -1=error
- pathname: Name of file to open
- flags: e.g.: `O_RDONLY/O_WRONLY/O_RDWR`=for read only/write only/read and write access, `O_APPEND`=for appending at the file end.
- mode: Access rights for the file to be created

- Close opened file:

`int close(int fd)`

- Return value: -1=error
- fd: File descriptor of file to be closed

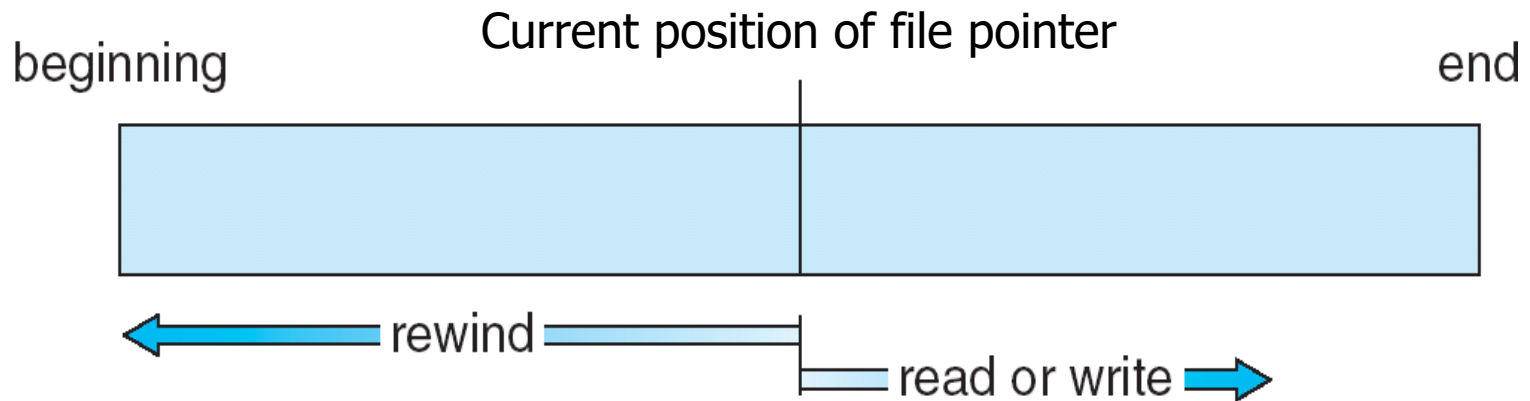
# Excursion: File Descriptor

---

- A **file descriptor** (FD) is a **number that uniquely identifies an opened file**. The FD is generated by the OS when opening/creating a file.
  - When opening/creating file, it is checked whether file exists, where it is stored, who is permitted to access, etc. If everything is OK, FD is generated.
- Most file operations require a valid file descriptor.
  - If no FD would be used, but instead a file name would be passed to each file operation, the OS would have to check at each file operation, whether the file exists, where it is located in the file system, ...
  - Using the FD approach, the OS just has to check this once when opening/creating a file.

# File Pointer

- When reading/writing bytes from/to a file, the bytes are read/written with respect to an offset in the file: the **file pointer**.
- When a file is opened, file pointer is set to 0.
  - (When opened with as `O_APPEND`, file pointer is set to end of file.)
- **Reading/writing advances file pointer (sequential access).**
- In addition, “rewind”/“fast forward” is possible to move file pointer without reading/writing (**direct access/random access**).
- Each process has it's own file pointer for each opened file.





# Operations for Accessing Files (2)

- POSIX operating systems:
  - Read data starting from current file pointer position:  
`ssize_t read(int fd, void *buf, size_t count)`
  - Write data starting from current file pointer position:  
`ssize_t write(int fd, const void *buf, size_t count)`
    - **Return value:** Number of actually transferred bytes, -1=error
    - **fd:** File descriptor
    - **buf:** Address in main memory where bytes should read into/write from
    - **count:** Number of bytes to be transferred
  - Move file pointer: `off_t lseek(int fd, off_t offset, int whence)`
    - **Return value:** New (absolute) file pointer position, -1=error
    - **fd:** File descriptor
    - **offset:** New position for file pointer (relative or absolute according to **whence**)
    - **whence:** `SEEK_SET=absolute`, `SEEK_CUR=relative` with respect to current file pointer, `SEEK_END=relative` with respect to end of file (i.e.using negative offset)
  - Further operations: **truncate, rename, unlink** (delete file or directory).

# Memory-mapped Files

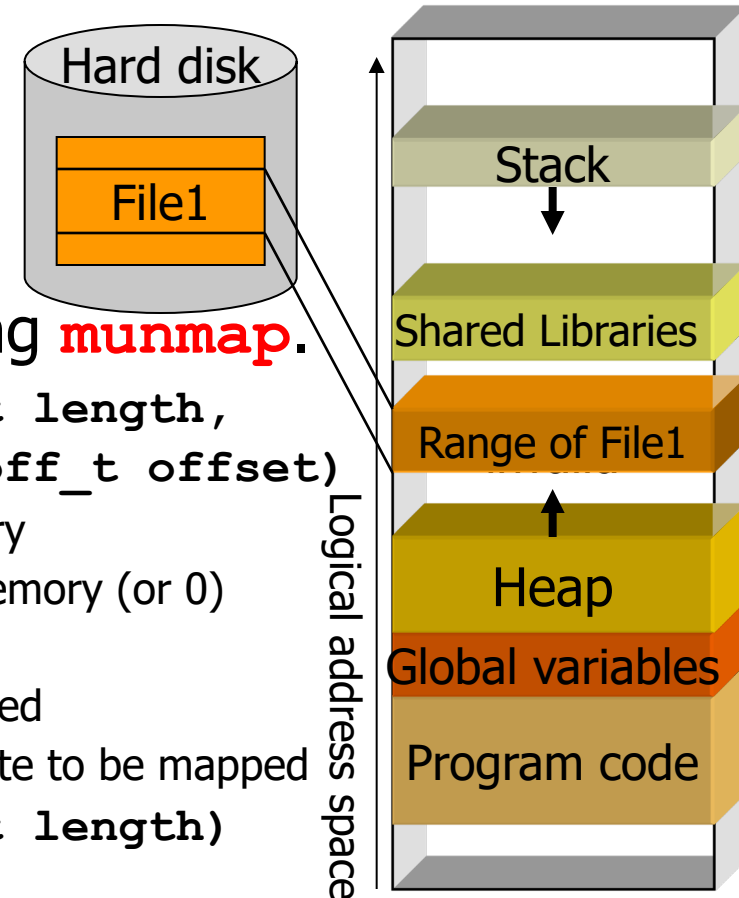
---

- If a huge file shall be accessed, it may be more convenient and faster to map the file contents into memory:
  - File contents can be accessed via ordinary memory accesses (e.g. using pointers).
    - No need to use `read()/write()` system calls (system call=slow).
  - Demand paging/lazy loading is internally used: only if a block of a file (corresponding to a page in memory) is actually accessed, it is loaded from the file system into memory.
    - Initially, a page corresponding to a file block is marked as invalid,
    - An access to that page will trigger a page fault interrupt,
    - Interrupt service routine will load block into frame,
    - Page table is updated: page marked as valid and points to frame,
    - Now, access to page is possible.

# Memory-mapped Files (POSIX)

1. Map a range of file into the logical address space (**mmap**).
2. Modify memory (or just read).
3. Write modified memory back using **munmap**.

- **void\* mmap**(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset)
  - **return value**: Address of mapped memory
  - **start**: Desired start address in main memory (or 0)
  - **length**: Size of range to be mapped
  - **fd**: File descriptor of file to be mapped
  - **offset**: Position in file containing first byte to be mapped
- **int munmap**(void \*start, size\_t length)
  - **return value**: 0=OK, -1=error
  - **start**: Start address of range to be written back
  - **length**: Size of range to be written back

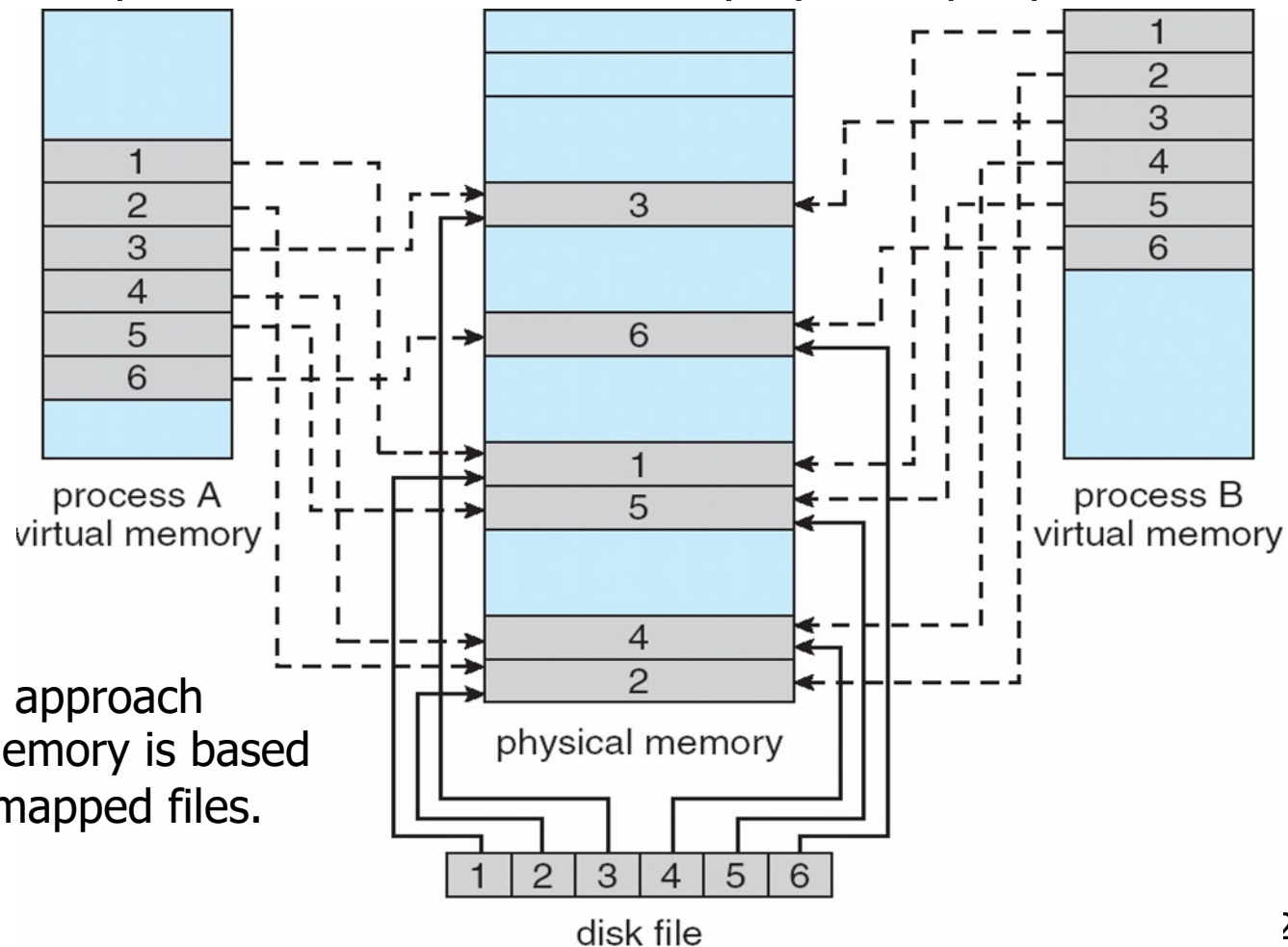


Memory-mapped files also supported by Java API!

# Memory-mapped Files and Shared Memory

- Even possible to map the same file concurrently by multiple processes

⇒ shared memory between processes.



- MS Windows approach for shared memory is based on memory-mapped files.

# Memory-mapped Files: Discussion

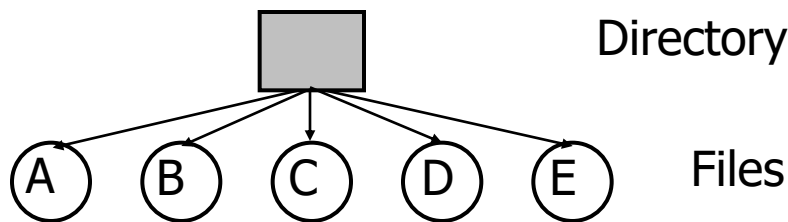
---

- You can of course as well load a huge file into memory using `read()`, modify it, and write it back using `write()`.
- However, memory-mapped files allow the OS to **load only those pages from a file that are actually accessed**.
  - ⇒ Suitable for processing huge files where you do not know in advance which locations you actually access.
  - Only the actually accessed locations need to be loaded from the (slow) mass-storage device,
  - **Only the modified (using PMMU modified bit) locations need to be written to the (slow) mass-storage device.**
- In fact, an operating system typically does demand loading of executable binary files (→9-36) by using internally the memory-mapped file concept.

# 10.3 Concept & Structure of Directories

---

- Files are managed in directories:
  - Flat (=single level) directory structure (outdated):  
Directory contains **only files**.

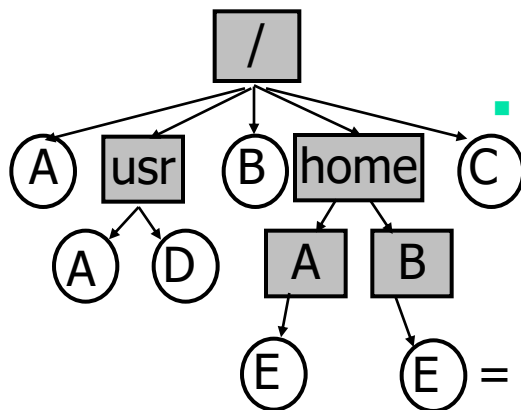


- Disadvantage: Directory gets cluttered in presence of many files  
⇒ **Hierarchal tree-like directory structure!**

# Hierarchical Directory Structure

- Directory contains files and sub-directories.
  - Directory tree starting from root directory:
    - Sub-directories can be used for structuring.
    - Files in different directories are independent from each other and may even have the same name.
    - Path navigates to file:
      - "/" as separator of directory levels.

Root directory



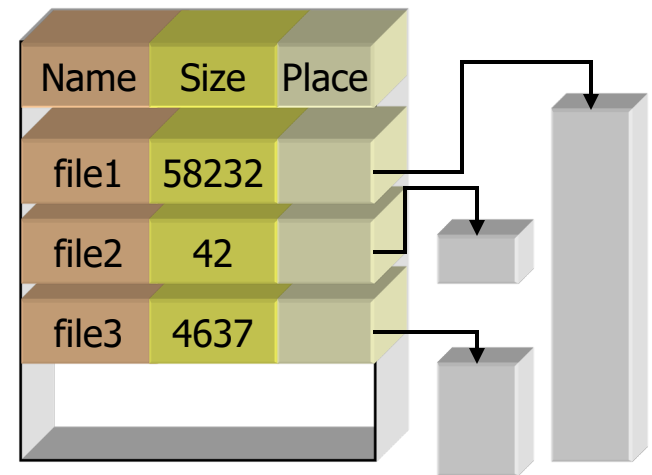
- "/" at the beginning of path refers to the root directory, i.e. a path beginning with "/" is an absolute path.
- If no "/" is at the beginning of path, this is a relative path, that refers to the current directory of a process. (Each process has a current directory as part of its context, i.e. this information is stored in the PCB.)

# Directories are just Files

- Using hierarchical directory structures, a (sub-)directory is just another file that is contained in a directory.
- However, what is the contents of such a directory file?

→ File attributes of all files stored in that directory!

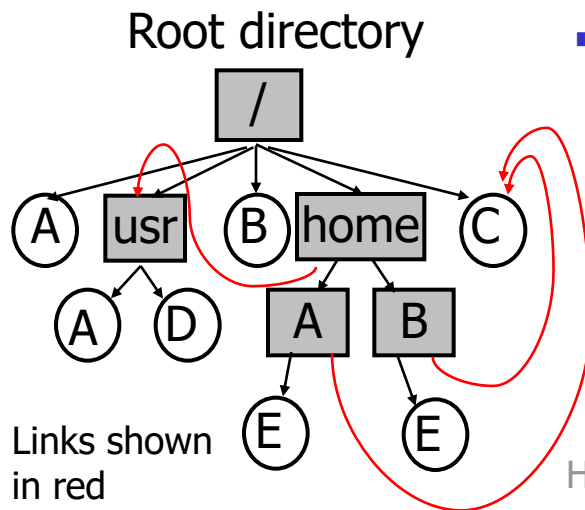
- At least stored in directory entry:
  - Name of file,
  - Location of file in file system (e.g. sector or block number on storage device).
- Further attributes (e.g. file size or information that a file is a directory):
  - Either stored as well in directory file
  - or stored at location of file in file system (enables hard links → 10-17).





# Directory with Links

- Adding **links** within directory tree:  
acyclic tree → acyclic graph (or even cyclic graph):
  - One file may be referenced in different directories or even using different file names.
  - **Special directory entries** that are in fact links:
    - "." references the directory itself, i.e.  
"." is a link to the file containing the respective directory.
    - ".." references the parent directory, i.e.  
".." is a link to the file containing the parent directory.



The contents of file `/C` can also be accessed as `/home/A/C` and `/home/B/C`; the directory `/usr` also as `/home/usr`.


Example usage of links: keep different versions of a file and switch between them by letting the link point to the file that shall be used; see example on slide 10-20.

# Hard Links

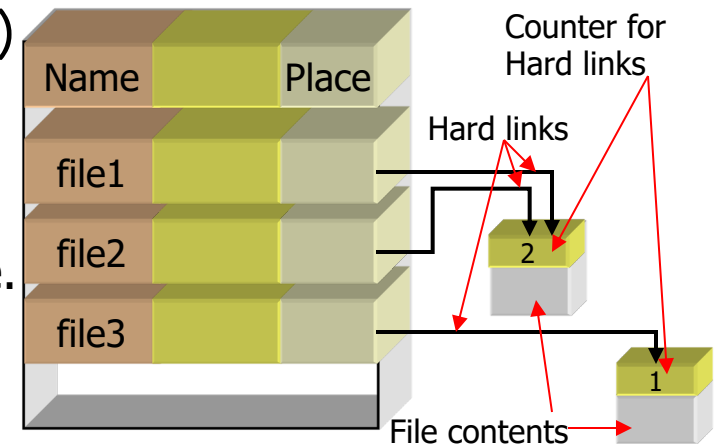
- **Hard link:** Link target is original file, i.e. a pointer to the original file's location (=block number).  $\Rightarrow$  It cannot be distinguished which of the directory entries contains the primary entry and which one the hard link. (In fact, both entries are hard links.)

- Calling a delete command on a hard link would delete the original file (& the hard link entry in the directory)  $\Rightarrow$  all further hard links would point to non-existing file.

■ **Solution:**

- Each file has a counter: how many hard links are currently pointing to that file?
  - When deleting, decrement hard link counter and remove directory entry, but do not remove file itself.
  - Only after last reference to a file has been deleted, the file itself is deleted.
- 
- The diagram illustrates a file system structure. On the left, a yellow triangle points to the first bullet point. To the right, a 3D box represents a directory entry. A red arrow points from the text 'File contents' to a specific block within the directory entry, which is labeled with the number '1'.

- Disadvantage: file location pointer only possible within same file system; file attributes other than name & place must be stored at file location.

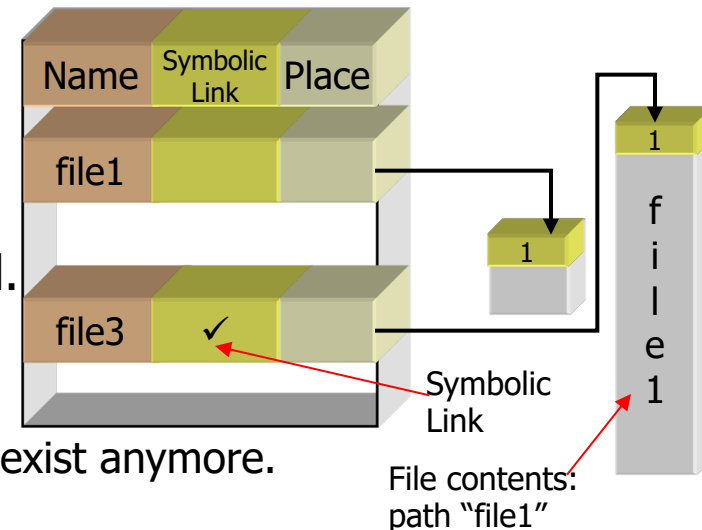


This is the reason, why the POSIX system call for deleting a file has as name **unlink**

Also known as  
"Shortcut" on  
MS Windows.

# Symbolic Links

- **Symbolic link:** Link target is a special file that contains a path name pointing to the original file. (The fact that the file is a special file, i.e. containing a path must be flagged by a special file attribute.)  $\Rightarrow$  It can be distinguished which directory entry is the primary entry for the file, and which directory entry is just the link.
  - Calling a delete command on a symbolic link just deletes the special file containing the path name (and the symbolic link entry in the directory). Original file is not affected.
  - Disadvantage:
    - Deleting the original file, leaves a dangling symbolic link pointing to a file that does not exist anymore.
  - Advantage:
    - Paths possible that link to files that are located on a different file systems.
    - File attributes (e.g. size) may be stored in directory entry (instead of storing file attributes at file location like the hard link counter).



# Hard links/Symlinks on POSIX

## ■ Example: Output of `ls -laF /boot`

"File contains directory entries" flag

Hard link pointing to current directory

Hard-Link pointing to parent directory

Target of symlink

<code>drwxr-xr-x</code>	3	root	root	4096	Nov	21	16:05	<code>./</code>	
<code>drwxr-xr-x</code>	24	root	root	4096	Nov	13	10:10	<code>../</code>	
<code>lrwxrwxrwx</code>	1	root	root		21	Sep	8	18:09	<code>System.map -&gt; System.map-3.4.22</code>
<code>-rw-r--r--</code>	1	root	root	516858	Mar	26	2018		<code>System.map-3.4.20</code>
<code>-rw-r--r--</code>	1	root	root	544356	Jun	25	2018		<code>System.map-3.4.21</code>
<code>-rw-r--r--</code>	1	root	root	591812	Nov	21	16:05		<code>System.map-3.4.22</code>
<code>-rw-r--r--</code>	1	root	root	512	Jun	12	2018		<code>boot.0300</code>
<code>-rw-r--r--</code>	1	root	root	36044	Mar	26	2018		<code>config-3.4.20</code>
<code>-rw-r--r--</code>	1	root	root	34896	Jun	25	2018		<code>config-3.4.21</code>
<code>-rw-r--r--</code>	1	root	root	27784	Nov	21	16:04		<code>config-3.4.22</code>
<code>drwxr-xr-x</code>	2	root	root	4096	Nov	21	16:46		<code>grub/</code>
<code>-rw-r--r--</code>	1	root	root	147086	Jun	12	2018		<code>initrd.gz</code>
<code>-rw-----</code>	1	root	root	54272	Sep	8	19:23		<code>map</code>
<code>lrwxrwxrwx</code>	1	root	root		18	Nov	21	16:05	<code>vmlinuz -&gt; vmlinuz-3.4.22</code>
<code>-rw-r--r--</code>	1	root	root	925720	Mar	26	2018		<code>vmlinuz-3.4.20</code>
<code>-rw-r--r--</code>	1	root	root	941965	Jun	25	2018		<code>vmlinuz-3.4.21</code>
<code>-rw-r--r--</code>	1	root	root	1220912	Nov	21	16:05		<code>vmlinuz-3.4.22</code>

"File contains symlink path" flag      Hard link counter      Size of directory file always a multiple of the block size (typically 4096)

Helmut Neukirchen: Operating Systems/updated

I.Hörleifsson email:ingolfuh@hi.is st.TG225

# Directory Operations

- Unix-based operating systems:
  - Change current working directory of process (used by relative pathnames):
    - `int chdir(const char *pathname)`
  - Create/remove directory:
    - `int mkdir(const char *pathname, mode_t mode)`
    - `int rmdir(const char *pathname)`
  - Create hard link/symbolic link, dereference symbolic link:
    - `int link(const char *oldpath, const char *newpath)`
    - `int symlink(const char *oldpath, const char *newpath)`
    - `int readlink(const char *path, char *buf, size_t bufsiz)`
  - Read directory entries (DIR=directory descriptor incl. pointer to current dir. entry):
    - `DIR *opendir(const char *pathname)` Open directory
    - `int closedir(DIR *dir)` Close directory
    - `struct dirent *readdir(DIR *dir)` Retrieve current directory entry and advance pointer to next entry
    - `void rewinddir(DIR *dir)` Reset pointer to current dir. entry
  - ...

# 10.5 Shared File Access and Protection

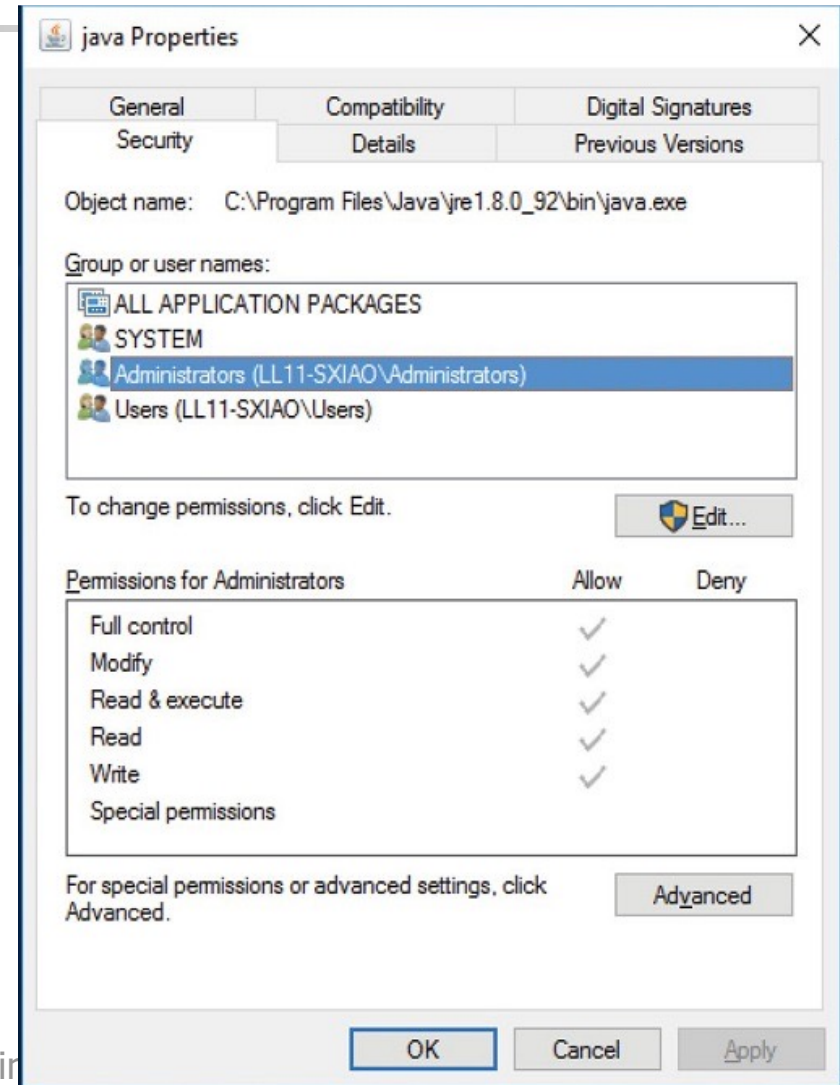
- An OS has to support users to **share files**, e.g.
  - System/program files (to avoid duplicating them for each and every user),
  - Application data (to allow users to collaborate).
- but also to **protect files** from being accessed by unauthorised users.
- To manage sharing/protection on POSIX systems, for each file, the access rights
  - **read/write/execute** can be granted to
    - **owner/user** (user that created the file),
    - **group** (each file has also a group associated: by default, the group to which the owner belongs. Groups are defined by the system administrator),
    - **all** (any user on that system).
- **chown** and **chgrp** commands can be used to change owner/group.
- **chmod** command can be used to change access rights, e.g.
  - **chmod g+w myfile.txt** to give write permission to the group.

`-rw-r--r-- helmut hi myfile.txt`

All may read.  
Members of group **hi** may read.  
Owner **helmut** may read/write.

# Access Control Lists

- MS Windows NTFS file system and some advanced Unix file systems use **Access Control Lists (ACL)** to allow to specify access rights individually to different users (or groups).
  - No need to have the system administrator define fixed groups for you.
  - Instead, the owner of a file can just add new users to the ACL and set permissions accordingly.
  - Incompatible with POSIX tools that assume the **rwX** user/group/all approach.



# 10.6 Summary

---

- Operating system provides file concept.
  - Applications just use file name and access operations to read/write data.
    - Applications do not need to deal with file system details.
    - Contents of file depends on application.
- Directories contain a mapping from file name to location in file system.
  - Nowadays used: hierarchical directories with at least symbolic links.
- In this chapter, only the interface for accessing files and directories have been presented.
  - Possible implementations of file systems are discussed in the next chapter.