# Operating Systems Notes - Chapter 13

July 27, 2025

## Contents

# 1 File concept

## Things to learn

- Explain file system function.
- Describe file system interfaces.
- Discuss file system design tradeoffs: access methods, sharing, locking, directory structures.
- Explore file system protection.

## Introduction

- **File**: collection of related information defined by its creator.
- OS maps files onto physical mass-storage devices.
- File system: describes how files map to physical devices, how accessed/manipulated.
- File systems designed for efficient access (physical storage can be slow).
- Other requirements: file sharing support, remote access.
- File system: most visible OS aspect for users.
- Provides mechanism for on-line storage/access to OS data/programs and user data.
- Consists of:
  - Collection of files (storing related data).
  - Directory structure (organizes, provides info about files).
- Most file systems live on storage devices (nonvolatile, persistent).

## File Concept

- Computers store info on various media (NVM, HDDs, tapes, optical disks).
- OS provides uniform logical view of stored info.
- OS abstracts physical properties to define logical storage unit: the **file**.
- Files mapped by OS onto physical devices.
- Storage devices usually nonvolatile, so contents persistent.
- File: named collection of related information recorded on secondary storage.
- User perspective: smallest allotment of logical secondary storage (data written only within a file).
- Commonly: files represent programs (source/object) and data.
- Data files: numeric, alphabetic, alphanumeric, binary.
- Files can be free form (text) or rigidly formatted.
- Generally: file is sequence of bits, bytes, lines, or records; meaning defined by creator/user.
- File concept is extremely general.
- Use stretched beyond original confines (e.g., UNIX 'proc' file system uses file-system interfaces for system info access).
- Info in file defined by creator.
- File has defined structure, depends on type.
- **Text file**: sequence of characters organized into lines.
- **Source file**: sequence of functions (declarations, executable statements).
- **Executable file**: series of code sections loader can bring into memory and execute.

## File Attributes

- File named for human users, referred by name (e.g., 'example.c').
- Some systems differentiate case, others not.
- File independent of creator process, user, system.
- Example: 'example.c' created by one user, edited by another, copied to USB/email/network, still 'example.c'.
- Second copy independent if no sharing/synchronization.
- File attributes vary by OS, typically include:
  - **Name**: symbolic, human-readable.
  - **Identifier**: unique tag (number), non-human-readable, identifies file within file system.
  - **Type**: for systems supporting different file types.
  - **Location**: pointer to device and file location on device.
  - **Size**: current size (bytes, words, blocks), possibly max allowed.

  - **Protection**: access-control info (read, write, execute).
  - **Timestamps and user identification**: creation, last modification, last use (useful for protection, security, monitoring).
- Newer file systems support **extended file attributes**: character encoding, file checksum.
- **File info window**: GUI view of file metadata (e.g., macOS).
- Info about all files kept in directory structure, on same device as files.
- Directory entry: file name, unique identifier (locates other attributes).
- Directory size can be large (MBs/GBs).
- Directories stored on device, brought into memory piecemeal.

## File Operations

- File is an abstract data type.
- OS provides system calls: create, write, read, reposition, delete, truncate files.
- **Creating a file**:
  1. Find space in file system.
  2. Make entry for new file in directory.
- **Opening a file**:
  - All operations except create/delete require ‘open()‘ first.
  - Returns file handle used as argument in other calls.
- **Writing a file**:
  - System call: open file handle, info to write.
  - System searches directory for file location.
  - System keeps **write pointer** to next write location (sequential).
  - Write pointer updated after each write.
- **Reading a file**:
  - System call: file handle, memory location for next block.
  - Directory searched for entry.
  - System keeps **read pointer** to next read location (sequential).
  - Read pointer updated after each read.
  - Current operation location: per-process **current-file-position pointer** (shared by read/write).
- **Repositioning within a file**:
  - Current-file-position pointer repositioned to given value.
  - No actual I/O involved.
  - Also known as file **seek**.
- **Deleting a file**:
  - Search directory for named file.
  - Release all file space for reuse.
  - Erase/mark directory entry as free.
  - **Hard links**: multiple names for same file; actual content deleted only when last link deleted.
- **Truncating a file**:
  - Erase contents but keep attributes.
  - Reset file length to zero, release file space.
- These 7 are minimal set. Other common: appending, renaming.
- Primitive operations combine for others (e.g., copy file).
- Operations to get/set file attributes (e.g., length, owner).
- To avoid constant directory searching: ‘open()‘ system call before first use.
- OS keeps **open-file table**: info about all open files.
- File specified by index into table (no searching).
- File closed: OS removes entry from open-file table, releases locks.
- ‘create()‘ and ‘delete()‘ work with closed files.
- Some systems implicitly open/close files (job termination).
- Most systems require explicit ‘open()‘/‘close()‘.
- ‘open()‘: takes file name, searches directory, copies entry to open-file table.

- ‘open()‘ accepts access-mode info (create, read-only, read-write, append-only).
- Mode checked against file permissions. If allowed, file opened.
- ‘open()‘ returns pointer to open-file table entry; used in all I/O ops.
- ‘open()‘/‘close()‘ complicated with simultaneous opens by multiple processes.
- OS uses two levels of internal tables:
  - Per-process table: tracks files process has open, current file pointer, access rights, accounting.
  - System-wide open-file table: process-independent info (disk location, access dates, size).
- File opened by another process: new entry in process's table points to system-wide entry.
- Open-file table has **open count**: number of processes with file open.
- ‘close()‘ decreases count; when zero, entry removed.
- File locks: allow one process to lock file/sections, prevent others.
- Useful for shared files (e.g., system log).
- **Shared lock**: multiple processes acquire concurrently (like reader lock).
- **Exclusive lock**: only one process at a time (like writer lock).
- Not all OS provide both types.
- **Mandatory** vs. **advisory** file-locking mechanisms.
- Mandatory: OS prevents other processes from accessing locked file (e.g., Windows).
- Advisory: OS does not prevent access; text editor must manually acquire lock (e.g., UNIX).
- Mandatory: OS ensures locking integrity. Advisory: developers ensure locks.
- File locks require same precautions as process synchronization (e.g., hold exclusive locks only during access, avoid deadlock).

## File Types

- OS should recognize/support file types for reasonable operations.
- Example: prevent outputting binary-object program as garbage.
- Common technique: include type as part of file name (name.extension).
- User/OS can tell type from name (e.g., ‘resume.docx‘, ‘server.c‘).
- OS uses extension to indicate type and allowed operations (e.g., ‘.com‘, ‘.exe‘, ‘.sh‘ for execution).
- ‘.sh‘ is a **shell script** (ASCII commands).
- Application programs use extensions (e.g., Java compilers expect ‘.java‘).
- Extensions not always required; user can omit, app looks for expected extension.
- Extensions are "hints" to applications, not OS-enforced.
- macOS: each file has type (‘.app‘), creator attribute (program that created it).
- Creator attribute set by OS during ‘create()‘ call, enforced.
- Example: word processor file has word processor as creator; double-click opens app, loads file.
- UNIX: **magic number** at beginning of some binary files indicates data type (e.g., image format).
- Text magic number for text files (e.g., shell language).
- Not all files have magic numbers; system features not solely based on this.
- UNIX does not record creating program.
- UNIX allows file-name-extension hints, but not enforced/depended on by OS; aid users.
- File types indicate internal structure.
- Source/object files: structures match expectations of programs reading them.
- Certain files must conform to OS-understood structure (e.g., executable file structure for loading/running).
- Some OS extend this: system-supported file structures with special operations.
- Disadvantage of OS supporting multiple file structures: large, cumbersome OS.
- OS needs code for each supported structure.
- Every file may need to be defined as one of supported types.
- New applications with unsupported structures: problems.
- Example: encrypted file not ASCII text, not executable binary. May need to circumvent/misuse OS file-type mechanism.
- Some OS impose minimal file structures (UNIX, Windows).
- UNIX: each file is sequence of 8-bit bytes; no OS interpretation.
- Provides maximum flexibility but little support; application must interpret structure.

- All OS must support at least one structure: executable file.

## Internal File Structure

- Internally, locating offset within file can be complicated for OS.
- Disk systems: well-defined block size (sector size).
- All disk I/O in units of one block (physical record); all blocks same size.
- Physical record size unlikely to match desired logical record length.
- Logical records may vary in length.
- Solution: packing logical records into physical blocks.
- UNIX: all files are streams of bytes. Each byte individually addressable by offset.
- File system automatically packs/unpacks bytes into physical disk blocks (e.g., 512 bytes/block).
- Logical record size, physical block size, packing technique determine records per block.
- Packing by user app or OS. File considered sequence of blocks.
- Basic I/O functions operate in terms of blocks.
- Conversion from logical records to physical blocks: simple software problem.
- Disk space allocated in blocks: some portion of last block wasted (**internal fragmentation**).
- Example: 512-byte blocks, 1,949-byte file → 4 blocks (2,048 bytes); 99 bytes wasted.
- All file systems suffer internal fragmentation; larger block size → greater fragmentation.

## Section glossary

| Term | Definition |
|---|---|
| **file** | Smallest logical storage unit; collection of related information. |
| **text file** | File containing text (alphanumeric characters). |
| **source file** | File containing program source code. |
| **executable file** | File containing program ready for loading/execution. |
| **extended file attributes** | Extended metadata (character encoding, checksums). |
| **file info window** | GUI view of file metadata. |
| **write pointer** | Location in file for next write. |
| **read pointer** | Location in file for next read. |
| **current-file-position pointer** | Per-process pointer to next read/write location. |
| **seek** | Operation of changing current file-position pointer. |
| **hard links** | File-system links where file has two+ names to same inode. |
| **open-file table** | OS data structure with details of every open file. |
| **open count** | Number of processes with an open file. |
| **shared lock** | File lock allowing concurrent acquisition by multiple processes. |
| **exclusive lock** | File lock allowing only one process to acquire at a time. |
| **advisory file-lock mechanism** | File-locking system where OS does not enforce locking. |
| **shell script** | File containing set series of commands specific to shell. |
| **magic number** | Number at start of file indicating data type. |
| **internal fragmentation** | Wasted disk space in last block of file due to block allocation. |

# 2 Access methods

## Accessing File Information

- Files store information; must be accessed and read into memory.
- Information accessed in several ways.
- Some systems provide only one access method; others many.
- Choosing right method: major design problem.

## Sequential Access

- Simplest access method: **sequential access**.
- Information processed in order, one record after another.
- Most common (editors, compilers).
- `read_next()`: reads next portion, automatically advances file pointer.
- `write_next()`: appends to end, advances to end of newly written material.
- File can be reset to beginning.
- Some systems: skip forward/backward `n` records.
- Based on tape model of file.
- Works on sequential-access devices and random-access ones.

## Direct Access

- Another method: **direct access** (or **relative access**).
- File: fixed-length **logical records**.
- Programs read/write records rapidly in no particular order.
- Based on disk model of file (disks allow random access).
- File viewed as numbered sequence of blocks/records.
- No restrictions on read/write order.
- Great use for immediate access to large info amounts (e.g., databases).
- Example: airline reservation system, flight info in block identified by flight number.
- Direct-access file operations: include block number as parameter.
- `read(n)`, `write(n)` instead of `read_next()`, `write_next()`.
- Alternative: retain `read_next()`, `write_next()`, add `position_file(n)`.
- Block number provided by user: **relative block number**.
- Relative block number: index relative to beginning of file (first is 0, next 1, etc.).
- OS decides file placement (**allocation problem**).
- Prevents user from accessing non-file portions of file system.
- Some systems start relative block numbers at 0, others at 1.
- Satisfying request for record `N`: turned into I/O request for `N` bytes starting at `N` * (logical record length).
- Logical records fixed size: easy to read, write, delete a record.
- Not all OS support both sequential and direct access.
- Some require file defined as sequential/direct at creation.
- Simulate sequential access on direct-access file: keep `cp` variable for current position.
- Simulating direct-access on sequential-access: extremely inefficient and clumsy.

## Other Access Methods

- Built on top of direct-access method.
- Involve constructing an **index** for the file.
- Index: contains pointers to various blocks (like book index).
- Find record: search index, use pointer to access file directly.
- Example: retail-price file (UPCs, prices). Sorted by UPC.
- Index: first UPC in each block. Can be kept in memory.
- Binary search index → find block → access block.
- Large files: index file too large for memory.
- Solution: index for the index file (primary index → secondary index → data).

- Example: IBM ISAM (indexed sequential-access method).
- Small main index points to disk blocks of secondary index.
- Secondary index blocks point to actual file blocks.
- File sorted on key.
- Find item: binary search main index → get secondary index block → binary search secondary index → find block with record → sequential search block.
- Any record located by at most two direct-access reads.

## Section glossary

| Term | Definition |
|---|---|
| **sequential access** | File-access method: contents read in order, beginning to end. |
| **direct access** | File-access method: contents read in random order. |
| **relative access** | File-access method: contents read in random order. |
| **logical records** | File contents logically designated as fixed-length structured data. |
| **relative block number** | Index relative to beginning of file (first is block 0). |
| **allocation problem** | OS determination of where to store file blocks. |
| **index** | Access method built on direct access; file contains index with pointers to contents. |

# 3 Directory structure

Directory: symbol table translating file names to file control blocks. Organization must allow:

- Insert entries
- Delete entries
- Search for named entry
- List all entries

Operations on a directory:

- **Search for a file**: Find entry for particular file; find files matching pattern.
- **Create a file**: Add new files to directory.
- **Delete a file**: Remove file from directory; may leave hole, defragmentation needed.
- **List a directory**: List files and their entry contents.
- **Rename a file**: Change file name when contents/use changes; may change position.
- **Traverse the file system**: Access every directory/file; for backup or space release.

## 3.1 Single-level directory

- Simplest structure: all files in same directory.
- Easy to support and understand.
- Limitations:
  - Files must have unique names (name collision problem for multiple users).
  - Difficult for single user to remember many file names.

## 3.2 Two-level directory

- Separate directory for each user.
- Each user has own **user file directory (UFD)**.
- System's **main file directory (MFD)** indexed by user name/account, points to UFD.
- When user refers to file, only their UFD searched.
- Different users can have same file names (unique within each UFD).
- OS searches only user's UFD for create/delete.
- UFDs created/deleted by special system program (restricted to administrators).
- Disadvantages:
  - Isolates users; disadvantage for cooperation.
  - To access another user's file, must specify user name and file name.
  - Two-level directory as a tree (MFD root, UFDs descendants, files leaves).
  - User name + file name = **path name**.
  - Example: `/userb/test.txt` or `C:\userb\test`.
- System files:
  - Copying system files to each UFD wastes space.
  - Solution: special user directory for system files (e.g., user 0).
  - OS first searches local UFD, then special system directory.
  - Sequence of directories searched: **search path**.
  - Search path can be extended; users can have own search paths.

## 3.3 Tree-structured directories

- Generalization of two-level directory to arbitrary height.
- Most common directory structure.
- Root directory; every file has unique path name.
- Directory (or subdirectory) contains files or subdirectories.
- Directory often treated as special file; one bit defines entry as file (0) or subdirectory (1).
- Special system calls to create/delete directories.
- Each process has a **current directory**.
- Reference to file: current directory searched.
- If not in current directory: specify path name or change current directory.

- Initial current directory from accounting file.
- Path names:
  - **Absolute path name**: begins at root (e.g., "/"), follows path down.
  - **Relative path name**: defines path from current directory.
  - Example: if current is `/spell/mail`, `prt/first` is same as `/spell/mail/prt/first`.
- User defines subdirectories for organization (e.g., by topic, info type).
- Deletion of a directory:
  - If empty: entry simply deleted.
  - If not empty:
    * Some systems: only delete if empty (user must delete contents recursively first).
    * Others (e.g., UNIX `rm -r`): delete directory and all its files/subdirectories recursively. More convenient, but dangerous.
- Users can access other users' files by specifying path name or changing current directory.

### 3.4 Acyclic-graph directories

- Allows directories to share subdirectories and files.
- No cycles (loops) in the graph.
- Shared file: one actual file exists, changes visible to all.
- Shared subdirectory: new files appear in all shared subdirectories.
- Implementation:
  - **Link**: pointer to another file/subdirectory (e.g., absolute/relative path name).
  - **Resolve**: use path name in link to locate real file.
  - OS ignores links during directory traversal to preserve acyclic structure.
  - Duplicate all info in both sharing directories: entries identical, but consistency issues on modification.
- Problems:
  - Multiple absolute path names for same file (aliasing).
  - Traversing entire file system: avoid traversing shared structures more than once.
  - Deletion: when can space be deallocated?
    * Deleting file leaves dangling pointers if other links exist.
    * Symbolic links: deletion of link doesn't affect original file. If original deleted, links dangle.
    * Preserve file until all references deleted: use **reference count**.
    * Increment count on new link/entry, decrement on deletion. Delete when count is 0.
    * UNIX uses for **hard links**.

### 3.5 General graph directory

- Allows cycles in the directory structure.
- Primary advantage of acyclic graph: simpler traversal and deletion algorithms.
- Problems with cycles:
  - Infinite loops during search/traversal.
  - Reference count may not be 0 even if file/directory is inaccessible.
  - Requires **garbage collection** to determine when space can be reallocated (time consuming for disk-based systems).
- Avoiding cycles: computationally expensive to detect.
- Simpler: bypass links during directory traversal.

Table 1: Section glossary

| Term | Definition |
|---|---|
| **user file directory (UFD)** | Per-user directory of files in two-level directory implementation. |
| **main file directory (MFD)** | Index pointing to each UFD in two-level directory implementation. |
| **path name** | File-system name for a file, containing mount-point and directory-entry info to locate it (e.g., "C:/foo/bar.txt"). |
| **search path** | Sequence of directories searched for an executable file when a command is executed. |
| **absolute path name** | Path name starting at the top of the file system hierarchy. |
| **relative path name** | Path name starting at a relative location (e.g., current directory). |
| **acyclic graph** | Directory structure implementation that contains no cycles (loops). |
| **link** | File that has no contents but points to another file. |
| **resolve** | To follow a link and find the target file. |
| **hard links** | File-system links where a file has two or more names pointing to the same inode. |
| **garbage collection** | Recovery of space containing no-longer-valid data. |

# 4 Protection

Information safety:

- Physical damage: **reliability** (duplicate copies, backups).
- Improper access: **protection**.

Protection mechanisms:

- User name/password authentication.
- Encrypting secondary storage.
- Firewalling network access.
- Multiuser systems: advanced mechanisms for valid data access.

## 4.1 Types of access

- Need for controlled access.
- Protection limits types of file access.
- Operations controlled:
  - **Read**: Read from file.
  - **Write**: Write or rewrite file.
  - **Execute**: Load and execute file.
  - **Append**: Write new info at end of file.
  - **Delete**: Delete file, free space.
  - **List**: List name and attributes.
  - **Attribute change**: Change file attributes.
- Higher-level functions (rename, copy, edit) often implemented by system programs using lower-level calls. Protection at lower level.

## 4.2 Access control

- Access dependent on user identity.
- Most general scheme: **access-control list (ACL)**.
- ACL specifies user names and allowed access types.
- OS checks ACL; allows if listed, denies otherwise.
- Advantages: complex access methodologies.
- Disadvantages:
  - Lengthy lists (tedious to construct, especially if users unknown).
  - Variable-size directory entries (complicated space management).
- Condensed ACL: three user classifications:
  - **Owner**: User who created file.
  - **Group**: Set of users sharing file, needing similar access.
  - **Other**: All other users.
- Common approach: combine ACLs with owner, group, universe scheme (e.g., Solaris).
- UNIX permissions:
  - Three fields: owner, group, universe.
  - Each field: three bits `rwx` (read, write, execute).

- `r` for read, `w` for write, `x` for execution.
- Example: `rwx` for owner, `rw-` for group, `r--` for others.
- `d` as first character indicates subdirectory.
- Sample listing shows links, owner, group, size, date, name.

- Combining ACLs and permissions:
  - User interface challenge: how to show optional ACLs.
  - Solaris: "+" appended to regular permissions (e.g., `-rw-r--r--+`).
  - Commands like `setfacl` and `getfacl` manage ACLs.
  - Windows: GUI for ACL management.
  - Precedence: ACLs typically take precedence over group permissions (specificity priority).

## 4.3 Other protection approaches

- Password with each file:
  - Effective if passwords random and changed often.
  - Disadvantages: many passwords to remember; single password for all files (all-or-none protection).
  - Some systems: password with subdirectory.
  - More commonly: encryption of partition/files, with key password management.
- Directory protection in multilevel structures:
  - Control creation/deletion of files in directory.
  - Control user's ability to determine file existence (listing directory contents).
  - If path name refers to file, user needs access to both directory and file.
  - Different access rights depending on path name in acyclic/general graphs.

Table 2: Section glossary

| Term | Definition |
|---|---|
| **access-control list** | A list of user names allowed to access a file. |

# 5 Memory-mapped files

Alternative file access method: **memory mapping** a file.

- Treat file I/O as routine memory accesses using virtual memory techniques.
- Can lead to significant performance increases.

## 5.1 Basic mechanism

- Map disk block to page(s) in memory.
- Initial access: demand paging, page fault.
- Page-sized portion of file read into physical page.
- Subsequent reads/writes: handled as routine memory accesses.
- Simplifies and speeds up file access by avoiding `read()` and `write()` system call overhead.
- Writes to memory-mapped file not necessarily immediate to secondary storage.
- Generally, updates written back when file closed.
- Under memory pressure, intermediate changes may go to swap space.
- Some OS (e.g., Solaris) memory-map all file I/O, even with standard calls, to kernel address space.
- Multiple processes can map same file concurrently for data sharing.
- Writes by one process visible to others mapping same section.
- Implemented by virtual memory map pointing to same physical page.
- Supports copy-on-write: processes share read-only, get own copies for modification.
- Processes use mutual exclusion for shared data coordination.
- Shared memory often implemented by memory mapping files.

## 5.2 Shared memory in the Windows API

- Outline for shared memory using memory-mapped files:
  1. Create a **file mapping** for the file.
  2. Establish a **view** of the mapped file in process's virtual address space.
- Second process opens and creates view of same mapped file.
- Mapped file acts as shared-memory object for inter-process communication.
- Example: Producer writes, Consumer reads.
- Steps:
  1. Open file with `CreateFile()` (returns `HANDLE`).
  2. Create file mapping with `CreateFileMapping()` (uses file `HANDLE`).
  3. Establish view with `MapViewOfFile()` (uses mapped object `HANDLE`).
- `CreateFileMapping()` creates a **named shared-memory object** (e.g., `SharedObject`).
- `MapViewOfFile()` returns pointer to shared-memory object; accesses to this memory are accesses to the file.
- Entire file or portion can be mapped.
- Mapped file may be demand-paged.
- Both processes remove view with `UnmapViewOfFile()`.

Table 3: Section glossary

| Term | Definition |
|---|---|
| **memory mapping** | File-access method where file is mapped into process memory space for direct memory access |
| **file mapping** | In Windows, the first step in memory-mapping a file. |
| **view** | In Windows, an address range mapped in shared memory; second step in memory-mapping a file. |
| **named shared-memory object** | In Windows API, a section of a memory-mapped file accessible by name from multiple processes. |

# 6 Summary

- File: abstract data type, sequence of logical records (byte, line, complex data). OS may support record types or leave to application.
- OS task: map logical file concept to physical storage (hard disk, NVM). May order logical records into physical records.
- Directories: organize files.
  - Single-level directory: naming problems in multiuser systems (unique names required).
  - Two-level directory: separate directory for each user, solves naming problems. Lists file name, location, length, type, owner, times.
  - Tree-structured directory: generalization of two-level, allows subdirectories for organization.
  - Acyclic-graph directory: allows sharing of subdirectories/files, but complicates searching/deletion.
  - General graph structure: complete flexibility in sharing, but may require garbage collection for unused space.
- Remote file systems: challenges in reliability, performance, security. Distributed information systems manage user, host, access info for shared state.
- File protection: needed on multiuser systems.
  - Access controlled by type: read, write, execute, append, delete, list directory.
  - Protection via access lists, passwords, other techniques.