File System

File System Layers

I/O Control Level

- Includes device drivers and interrupt handlers to manage data transfer between memory and the storage device. Translates high level requests into device specific instructions

Basic file system (block I/O subsystem in Linux)

- Issues generic commands to the device driver based on logical block addresses. Handles I/O request scheduling

File - organization module

- Understands files and their logical blocks and translates logical block numbers to physical block numbers. Also manages free space manager

Logical file system

- Manages metadata, including directory structure and file control blocks (FCBs) or inodes. Also responsible for protection

Modern OS systems support multiple file system types. The Virtual File System (VFS) layer provides a standard interface, allowing OS to access different file system types (local or remote) transparently

Operations and usage

Support operations like create, write, read, reposition (seek), delete, truncate.

- To avoid searching directory repeatedly, files are often not opened before use, and closed when no longer needed. The open() call returns a file handle or file descriptor that is used for subsequent I/O operations. OS maintains open-file tables (per process and system wide) for active files

Access methods - Determine how info in a file is accessed

- **Sequential access** processes data in order (like a tape) using `read next()` and `write next()`
- **Direct access** allows reading and writing records in any order (like a disk) using `read(n)` and `write(n)` or `position file(n)`
  - Direct access uses relative block numbers
  - Other methods can be built on direct access using indexes

File Locking

- Mechanisms (shared or exclusive) that can be used to mediate access to shared files. These can be mandatory (OS enforces) or advisory (processes check status and decide)

Directories

Directories organize files, acting like a symbol table translating file names to file control blocks. Often implemented as files themselves

- **Single Level Directory**
  - All files in one directory. Simple but has naming and grouping problems for multiple users or many files
- **Two Level Directory**
  - Separate directory for each user, indexed by a master file directory. Solves naming conflicts between users but isolates users. Files are referenced by path names (user name + file name)
- **Tree Structured Directories**
  - Arbitrary height tree with root directory & subdirectories. Files have unique path names. Path names can be absolute (from root) or relative (from current directory)
- **Acyclic Graph Directories**
  - Files or subdirectories can appear in multiple directories using **links**:
    - ***Hard link*** — points directly to the file's data (multiple directory entries reference the same data on disk).
    - ***Soft link (symbolic link)*** — stores the path to another file; acts like shortcut.
  - More flexible sharing, but complicates deletion and file system traversal.
- **General Graph Directory**
  - Allows cycles. Even more flexible but complex, requiring garbage collection to manage free space

Remote file systems

Allow access to files stored on machines across a network. Methods range from manual transfer to automated Distributed File Systems (DFS)

- DFSs typically use a client-server model, where servers provide access to files. Challenges include client identification and handle features
- Consistency semantics specify how multiple users accessing a shared file concurrently will observe modifications
  - Examples include
    - UNIX semantics (writes immediately visible to others with file open),
    - Session semantics (writes visible only in sessions starting *after* the file is closed,
    - Immutable shared files semantics (files cannot be modified once declared shared)
  - **NFS (Network File System)** is a widely used client server DFS example
    - NFSv3 is stateless, simplifying server recovery but impacting security and consistency. NFSv4 is stateful

File System Implementation

What's on Disk - File systems reside on secondary storage devices, like HDDs and NVMs. Devices can be divided into **partitions**, and a file system typically resides on a **volume**, which can be a partition or span multiple partitions

- Partitions can be
  - Cooked - contain a file system
  - Raw - Contain no file system, used for swap space, databases)
- Volumes containing an OS have a **boot control block** (boot block) for bootstrapping the system
- Volume control block contains volume specific details like size, free blocks, and FCB pointers
- Directory structure and per file File Control Blocks (FCBs) or inodes are also stored on the storage device

What's in Memory - OS uses several in-mem data structures for file system management and performance

- **In-memory mount table** - Info about mounted volumes
- **In-memory directory-structure cache** - For recently accessed directories
- **System-wide open-file table** -Holds copies of FCBs for all open files
- **Per-process open-file table** - With entries pointing to the system wide table for files a specific process has open
- **Buffers** - To hold file system blocks during I/O transfers

Control Blocks (Inode)

File control block (FCB, known as an **inode** in UNIX)

- Contains metadata about a specific file, such as permissions, dates, size, owner, group, and info needed to locate the file's data blocks on the storage device
- Structure of the FCB/inode is central to how data blocks are allocated and accessed (contiguos, linked, indexed)

Free Space Management

- The process of tracking and reusing space from deleted files
- System maintains a free-space list (or equivalent structure)
- **Bit vector** - Represents each block with a bit (1 for free, 0 for allocated)
  - Simple and efficient for finding contiguous space
  - Can require significant memory for large devices
- **Linked List** - Linked all free bloks
  - Simple but inefficient for finding contiguous blocks
- **Grouping** - Stores pointers to multiple free blocks in the first free block
- **Counting** - Stores the addy of the first free block and number of contiguous free blocks following it. Useful with allocation methods like contiguous or extents

Copy-on-write

- Updates are written to new blocks instead of overwriting existing ones
- Metadata pointers are updated atomically to the new blocks
- Snapshots can be created by retaining old pointers
- Backups (full or incremental) are also a crucial part of data recovery

Getting a file caching

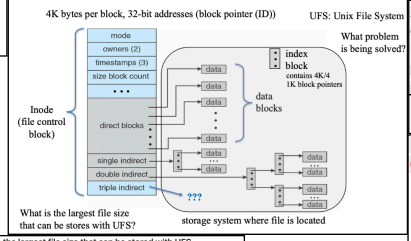
Caching improves performance by storing frequently accessed data in faster storage (like main mem)

- **Buffer Cache** - Stores disk blocks in main memory
- **Page Cache** - Caches file data using virtual memory techniques, treating file data as pages
- **Unified Buffer Cache** - Where page cache is used for both memory-mapped files and regular file I/O, avoiding double caching
- Caching also applies to **metadata**, Directory caches and inode caches store frequently needed directory entries and FCBs in memory for fast lookup
- **Write policies** - Determine when modified data is written back to storage
  - **Synchronous writes** wait for data to hit the device. Ensures reliability, impacts performance
  - **Asynchronus writes** buffer data and return control quickly
  - **Delayed-write (write back)** flushes modified blocks later, improving write performance but risking data loss on crashes
  - **Write on close** is a variation used in OpenAFS
  - **Read-ahead** optimize sequential access by **pre-fetching data**

Life Cycle of an I/O Request

Transforms an app's I/O request into device operations through layered steps:

1. **Application Layer**
   App issues system call → OS validates params.
2. **Kernel I/O Subsystem**
   - If data cached → return immediately.
   - Else → queue I/O request, move process to wait queue.
3. **Device Driver Layer**
   - Prepares request.
   - Allocates buffers, sends commands to device controller.
4. **Device Controller + Hardware**
   - Executes operation (seek, read, write).
   - If DMA used → DMA controller transfers data.
5. **Completion & Interrupt Handling**
   - Device signals interrupt.
   - ISR runs → updates state, notifies driver.
6. **Kernel & Process Wakeup**
   - Kernel moves process to ready queue.
   - Transfers data/status to process buffers.
7. **Process Resumption**
   - Process resumes → system call returns result.



What is the largest file size that can be stored with UFS

- Block size = 4KB
- Each block pointer = 32 bits → 4 bytes → an index block of 4KB/4B = 1024 pointers

Level	Number of blocks reachable	Size in bytes
Direct blocks (12)	12	12 x 4 KB = 48 KB
Single indirect	1024	1024 x 4 KB = 4 MB
Double indirect	1024 x 1024	1,048,576 blocks = 4 GB
Triple indirect	1024 x 1024 x 1024	1,073,741,824 blocks = 4 TB

File Allocation

This is how storage space is assigned to files

- **Contiguous allocation** requires file to occupy set of adjacent blocks
  - **Pros** - Offers a simple access and good sequential performance
  - **Cons** - (1) External fragmentation (2) Need to pre-declare file size (3) Difficulty in extending files
  - **Extents**- mod allocating contiguous chunks that are linked
- **Linked Allocation**
  - represents a file as a linked list of blocks scattered anywhere on the device. Directory entry points to the first and last blocks. Each block contains a pointer to the next
  - **Pros** (1) This avoid external fragmentation and need to pre-declare size - any free block can be used. (2) Simple to grow files — just add another block and update the pointer

    Cons (1) No random access — must follow pointers block by block to find data. (2) Overhead of storing a pointer in each block (3) Less efficient for large files or random reads
- **File Allocation Table (FAT)**
  - A table in mem stores a linked list of block pointers for each file
  - Each FAT entry corresponds to one disk & points to next block

    Pros (1) Faster access - FAT is in memory → faster traversal than following on-disk pointers (2) Easier to implement random access (can walk FAT in memory)

    Cons (1) FAT can become large and consume a lot of memory (especially on large disks). (2) Must update FAT in memory and on disk → some extra overhead
- **Indexed allocation** Brings all pointers for a file together into an *index block*.
  - Directory points to the index block
  - Index block contains pointers to the file's data blocks

    Pros (1) Efficient direct/random access (2) Avoids external fragmentation

    Cons (1) Small files waste index block space (2) Large files → use linked index blocks or multi-level indexes (3) Requires accessing index block before data blocks

Def - File System -The file system provides the mechanism for on-line storage and access to both data and programs. Tt is the most visible aspect of the OS. File system offers a uniform logical view of information storage, abstracting away physical properties of the storage devices. Involves mapping logical files onto physical storage devices

Def - On-line storage - Storage devices that are *immediately* accessable by the computer system. AKA storage on the computer.

`Which actions take place in the OS when a process makes an I/O request (assume interrupts are used)?`

- **OS checks if request can be handled immediately: Correct** — common first step.
- **OS blocks process if needed: Correct** — blocks if request is asynchronous or device busy.
- **Device driver issues commands: Correct** — driver must initiate hardware operation.
- **Device driver polls device, transfers byte by byte: Incorrect** Wrong — polling is not used in interrupt-driven model.
- **If reading, OS wakes process & process reads buffer: Correct** — part of I/O completion handling.

`What file system structures are in kernel memory?`

- **System-wide open file table: Correct** — maintained in kernel.
- **Unmounted file system directory: Incorrect** Not loaded into kernel memory unless mounted.
- **Per-process open file table: Correct** — maintained per process in kernel.
- **File control block of unopened file: Incorrect** Loaded into kernel only when file is opened.
- **User file data blocks: Incorrect** Stored on disk or in file buffers, not persistent kernel structure.
- **Page cache: Correct** — kernel uses page cache to optimize file reads/writes.

`For the following devices, which could benefit from using DMA?`

- **Mouse: Incorrect** Low data volume, DMA unnecessary.
- **10 gigabit network interface: Correct** — high data rates benefit from DMA.
- **Keyboard: Incorrect** Very low data rate, no DMA needed.
- **Video camera: Correct** — large continuous data stream benefits from DMA.
- **Backing store: Correct** — disk I/O is high volume and benefits from DMA.
- **Bluetooth headphones: Incorrect** Typically managed through audio stack and buffering, not general-purpose DMA.

`A file system must be mounted before it can be used by an program.`

True Mounting attaches the file system to the directory tree so programs can access its files.

`A file control block contains the directory of a file system.`

False A file control block (FCB) stores metadata about **individual files**, not the directory itself.

`It is possible for different file systems to be mounted simultaneously and each file system can use a different block size for file allocation.`

True Different file systems can be mounted at the same time, and each can independently define its block size.

`The open file table for a process is allocated in the process memory.`

False Each process has a per-process open file table, but it is typically managed by the OS in **kernel memory**, not user process memory.

`Data in a file are referenced by byte with a logical byte address starting at position 0 (byte address 0) in the file and going to the file size minus 1.`

True File data is logically addressed from byte 0 to byte (file size - 1).

`Which of the following could be beneficial to improving the efficiency of a file system implementation and file operation?`

- **File block buffering: Correct** — reduces disk accesses.
- **Memory-mapped files + page caching: Correct** — speeds up file access via virtual memory.
- **Limiting number of files per volume: Incorrect** Not an efficiency technique, just an artificial limit.
- **Block prefetching: Correct** — prefetching reduces read latency.
- **Inode caching: Correct** — speeds up file metadata access.
- **High-speed disk drives: Correct** — faster hardware improves overall I/O speed.

`Which of the following are considered as a main part of a file system?`

- **Logical file system: Correct** — defines file naming, permissions, and logical view.
- **I/O buses: Incorrect** Hardware layer, not part of file system itself.
- **File system organization: Correct** — directory structures are essential to file systems.
- **Storage devices: Correct** — underlying physical media used by the file system.
- **Memory-mapped files: Incorrect** Technique for file access, not part of file system definition.
- **Vectored interrupts: Incorrect** Hardware-level, not related to file system.

`Which statements are true about synchronous and asynchronous I/O?`

- **Asynchronous I/O always more efficient: Incorrect** Not always — efficiency depends on context and device.
- **If process requires I/O done first, use synchronous: Correct** — synchronous blocks until done.
- **Asynchronous I/O blocks process until interrupt: Incorrect** False — asynchronous allows process to continue running.
- **Process must check for completion in async I/O: Correct** — must verify whether I/O completed.
- **None of the above: Incorrect** Incorrect because two statements are true.