# Lecture 21: Storage Device and File Systems

## (Storage Methods, Abstraction, Sharing, Directory, FAT, ext2)

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

# Overview

**Review**

- Operating System
  - Manager of machine states
  - Objects + API

**Questions Answered in This Class**

- **Q1:** How is the current state of the machine stored?
- **Q2:** How are more persistent states stored?
- **Q3:** How do applications share access to storage devices?

# Outline

**Main Topics for This Class**

- 1-bit storage methods
- Volatile/non-volatile storage
- Storage device abstraction
- Storage device sharing
- File system APIs
  - Namespace
  - Directory
- Classic File Systems
  - File Allocation Table (FAT)
  - Second Extended Filesystem (ext2)

# Storage Methods

How to Store 1-Bit Data?

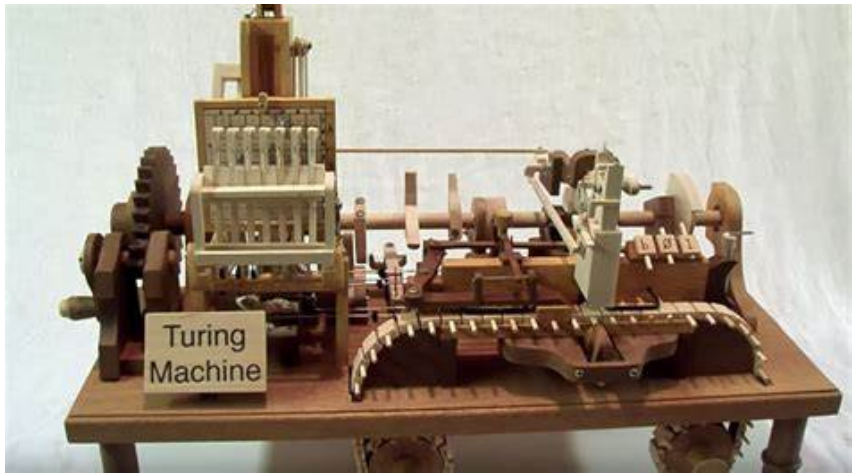
# Computers Need to Store the "Current State"

**Instruction Set Architecture** only has "two" states

- Registers: rax, rbx, ..., cr3, ...
- Physical memory

**Requirements for Storing the "Current State"**

- Must be addressable
  - Read and write data based on encoding
- Access speed should be as fast as possible
  - Even at the cost of losing state after power-off
  - This is why we have a memory hierarchy

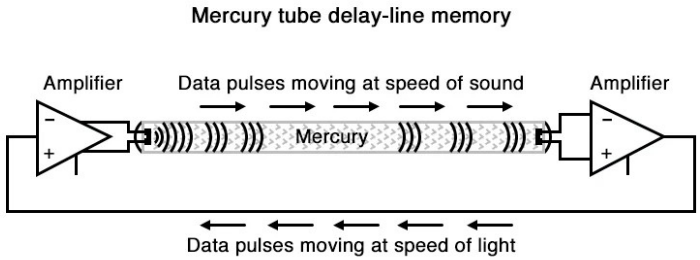



The mechanical Turing machine does not lose state after power-off.

# Storage of "Current State"

## Delay Line: Acoustic Delay Line

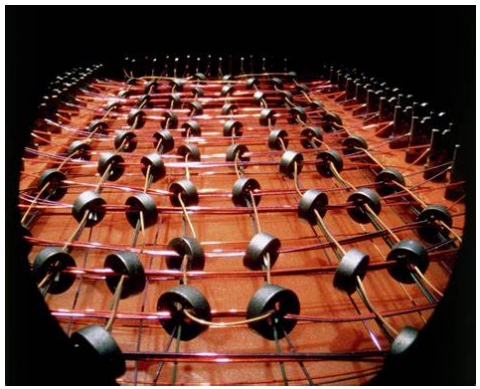
- Due to signal decay, requires continuous amplification




An acoustic delay line used to store data as sound waves. Data pulses travel along the line, and amplifiers boost the signal to counteract decay.

## Magnetic Core Memory

- The origin of "Segmentation fault (core dumped)"
- Non-volatile memory!



Magnetic core memory, storing data by the magnetization direction of tiny ferrite cores. Each core represents a single bit, retaining data even when powered off.

## SRAM/DRAM: Flip-Flop and Capacitors

- Today's implementation approach

# The Journey Towards Persistence

## Persistence

*"A firm or obstinate continuance in a course of action in spite of difficulty or opposition."*

Beyond just the "current state," we want larger and more data to be "retained" (and managed efficiently by the operating system).

**The Nature of Persistent Storage**

- The foundation of all file structures
  - Conceptually a bit/byte array
  - Managed in blocks, allowing us to read and write in "chunks" according to locality
- Evaluation criteria: cost, capacity, speed, reliability
- Witnessing yet another highlight of human civilization!

# Storage Medium: Magnetic

# ”Persistence” May Not Be As Difficult As Imagined

Magnetic drawing board where the magnetic particles change direction to display drawings. Erased by resetting particles’ direction.

**Further Step:** Representing 1-Bit Information Using the ”Magnetization Direction” of Iron Particles

- **Read**: Amplify the induced current
- **Write**: Magnetize the iron particles with a magnetic needle

# Magnetic Tape (1928)

## 1D Storage Device

- Rolls up bits
  - Iron magnetic particles evenly coated on the tape
- Requires only one mechanical component (rotation) for positioning

Fritz Pfleumer with the first magnetic tape recording device, 1928

# Magnetic Tape: Analysis as a Storage Device

## Analysis

- Price
  - Very Low - Made from inexpensive materials
- Capacity
  - Very High
- Read/Write Speed
  - Sequential Access: Strong - Requires waiting for positioning
  - Random Access: Nearly impossible
- Reliability
  - Contains mechanical components, requires stable storage environment

## Current Applications

- Storage and backup of cold data


PowerVault LTO-9

$6,839.00

Price Match Guarantee

Financing Offers

Learn More | Pre-Qualify Now

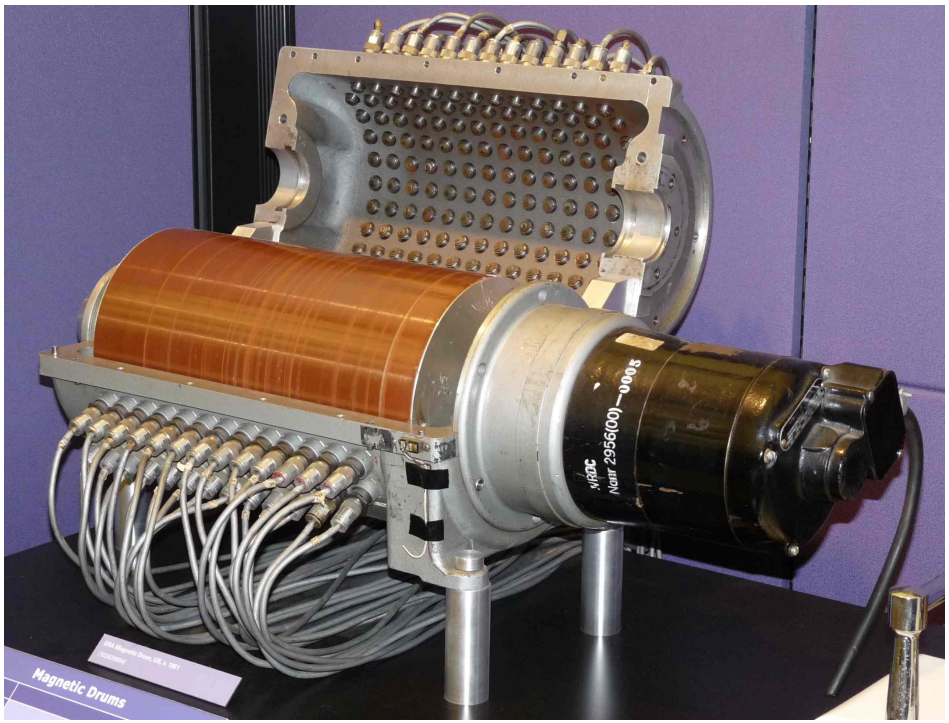
Add to Cart

Example: DELL LTO-9 Magnetic Tape, 18TB Capacity

# Magnetic Drum (1932)

**1D $\rightarrow$ 1.5D (1D $\times$ n)**

- Stores data on a rotating 2D surface
  - No inner roll, limited capacity
- Read/write delay does not exceed the rotation period
  - Significant improvement in random access speed

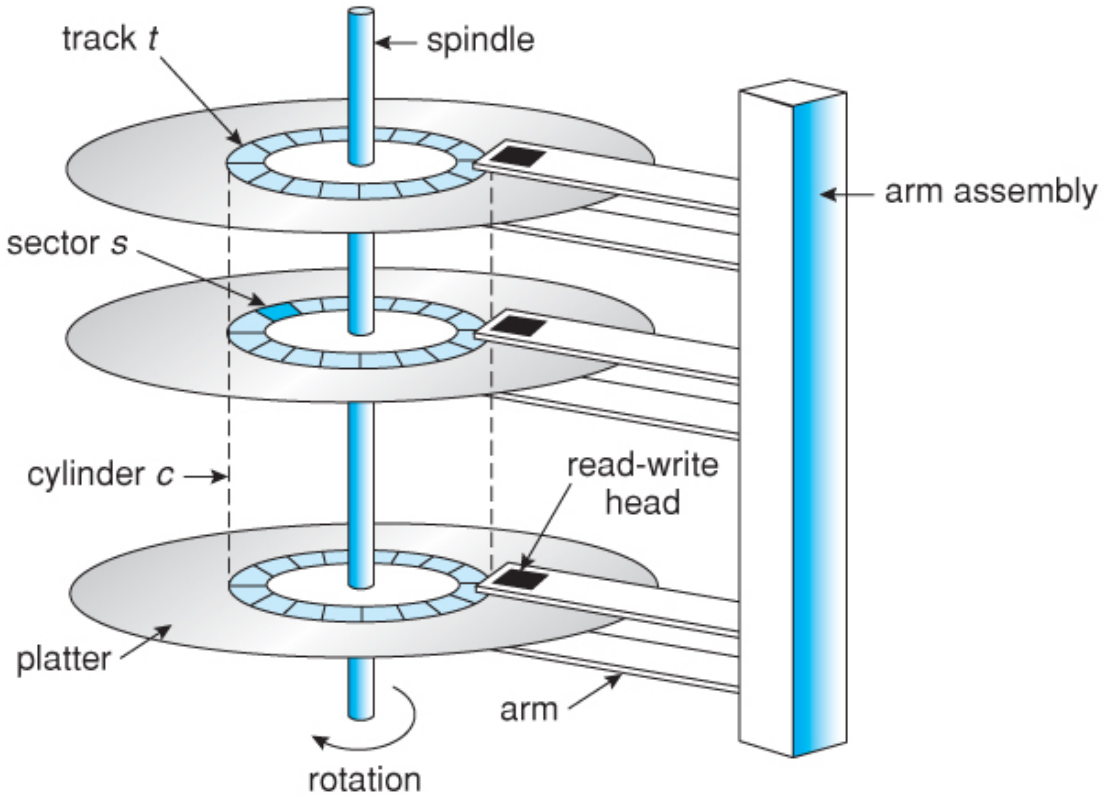



Magnetic Drum, an early form of storage using magnetic coatings on a rotating drum to store data.

# Hard Disk (1956)

## 1D $\rightarrow$ 2.5D (2D $\times$ n)

- Multiple magnetic platters on a two-dimensional plane




Hard Disk structure with multiple platters and a read-write head for each platter, allowing 2D access within each platter and across multiple platters.

# Hard Disk: Analysis as a Storage Device

## Analysis

- Price
  - Low - Higher density, lower cost
- Capacity
  - High (2.5D) - Tens of thousands of tracks on each platter
- Read/Write Speed
  - Sequential Access: Relatively High
  - Random Access: Strong
- Reliability
  - Mechanical parts present; head crash can damage platters and lead to data loss

## Current Applications

- Primary data storage for computer systems (large data volume; low cost is key)

# Hard Disk: Performance Optimization

To read/write a sector:

1. The read/write head must reach the correct track
   - 7200 rpm $\rightarrow$ 120 rps $\rightarrow$ "Seek" time of 8.3 ms
2. The spindle rotates the platter to align the sector with the read/write head
   - Head movement time also typically takes several ms

Optimizations through caching/scheduling:

- Examples include the well-known "elevator" scheduling algorithm
- Modern HDDs have advanced firmware to manage disk I/O scheduling
  - `/sys/block/[dev]/queue`
  - `[mq-deadline]` none (priority to reads; writes do not starve)

**Note:** Modern operating systems no longer handle disk operations directly; everything is managed by the disk controller. Therefore, traditional disk scheduling algorithms are outdated!

# Floppy Disk (1971)

Separating the read/write head and the disk enables data movement:

- Floppy disk drive (Break Time! ) on computers + removable floppy disk
  - Sizes: 8” (1971), 5.25” (1975), 3.5” (1981)
    - The earliest floppies were very cheap, essentially just a piece of paper
    - The 3.5-inch floppy was made ”hard” to improve reliability


Various sizes of floppy disks, including 8”, 5.25”, and 3.5”.

# Floppy Disk: Analysis as a Storage Device

**Analysis**

- Price
  - Low - Made of plastic, diskette, and some small materials
- Capacity
  - Low (Exposed storage medium, limited density)
- Read/Write Speed
  - Sequential/Random Access: Low
- Reliability
  - Low (Exposed storage medium)

**Current Applications**

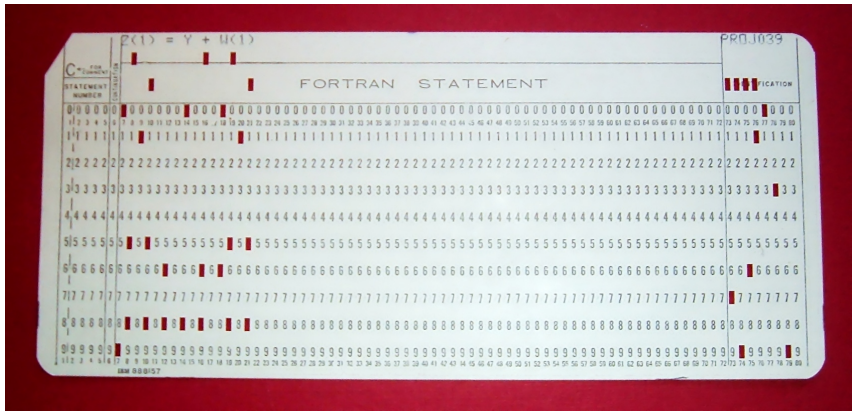
- Displayed in museums for public viewing
- Completely replaced by USB Flash Drives

# Storage Medium: Pit

# Holes: Naturally Readable Data Storage

- Example of data storage that is intuitively easy to ”read”


An ”SOS” message written in the sand, easily readable by humans.




A punch card used for data storage, with holes representing coded information.

# Compact Disk (CD, 1980)

- Pits (0) are etched on a reflective surface (1) to store data
- A laser scans the surface to read information from the pits
  - Invented by Philips and Sony (for digital audio)
  - Capacity of  700 MiB, which was very large at the time

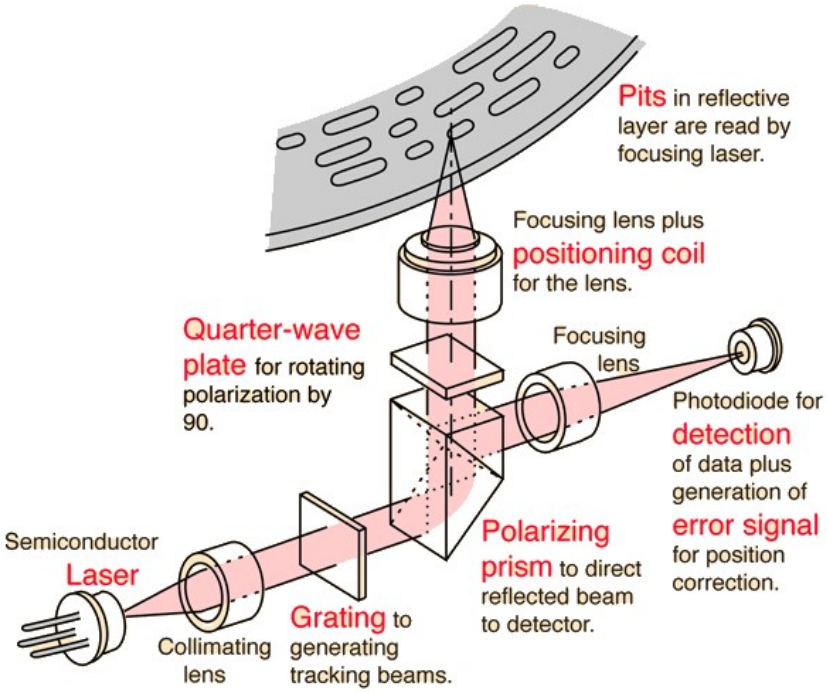



Diagram of how a CD works, with a laser reading pits on the reflective layer.

Close-up view of a CD surface, showing the pattern of pits and lands.

# CD-RW

Can the limitation of read-only be overcome?

- Method 1
  - Etch a pit using a laser (”write once”)
  - Use an append-only data structure
- Method 2: Alter the reflective properties of the material
  - PCM (Phase-change Material)
  - How do rewritable CDs work?

# Advancements in ”Pit” Technology

- **CD (740 MB)**
  - 780 nm Infrared Laser
- **DVD (4.7 GB)**
  - 635 nm Red Laser
- **Blu-ray (100 GB)**
  - 405 nm Blue-violet Laser

PS5 Drive

# Optical Disk: Analysis as a Storage Device

**Analysis**

- Price
  - Very low (and easy to replicate with ”pressing” technology)
- Capacity
  - High
- Read/Write Speed
  - High sequential read speed; strong random read performance
  - Low write speed (pits are hard to form and fill)
- Reliability
  - High

**Current Applications**

- Distribution of digital media (increasingly replaced by internet-based ”on-demand” distribution)

# Storage Medium: Electrical

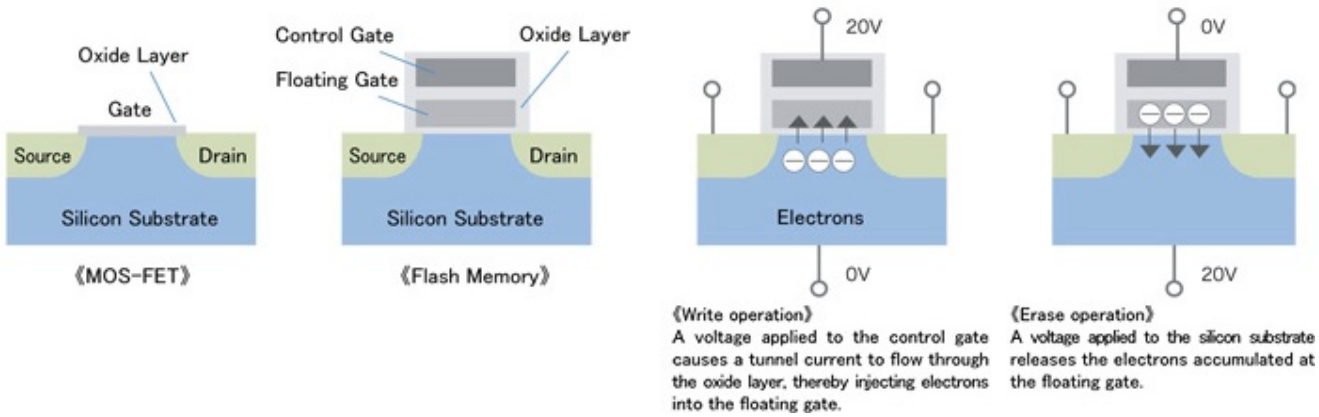
# Solid State Drive (1991)

Previous persistent storage mediums had critical drawbacks:

- Magnetic storage: Mechanical components cause delays in the millisecond range
- Optical storage (CDs): Once a pit is created, it is difficult to modify (CDs are read-only)

Eventually, electricity (circuits) provided a solution:

- Flash Memory
  - Floating gate technology stores 1-bit information by charging/discharging



# USB Flash Disk (1999)

USB drives have large capacity, fast speed, and are relatively affordable.

- Quickly replaced floppy disks and became a common storage medium
  - Compact Flash (CF, 1994)
  - USB Flash Disk (1999, developed by M-Systems)

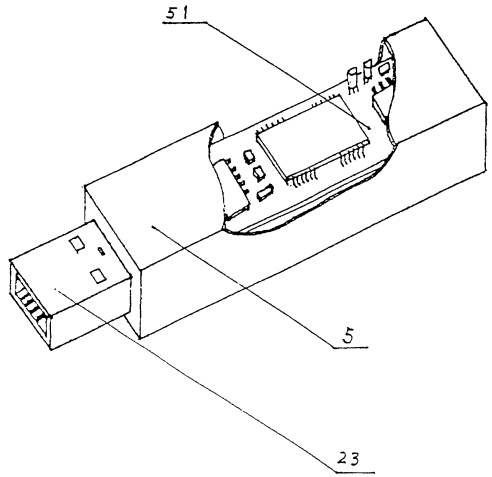



*Fig.2*

Image from Patent US6829672

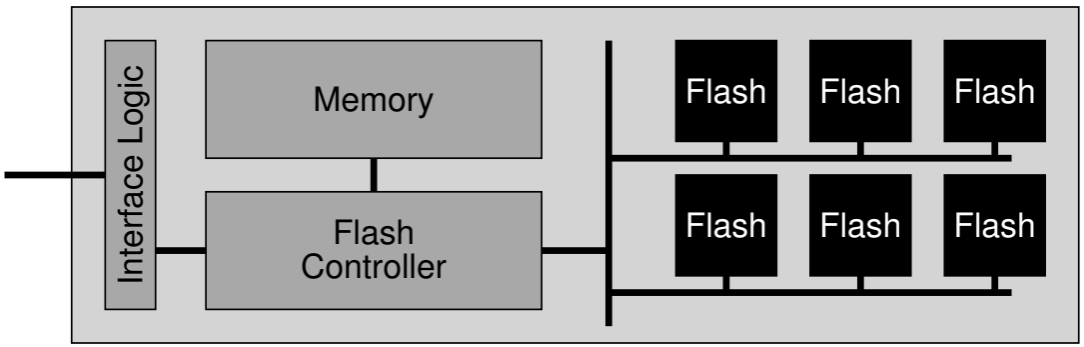
**Challenges with Flash Memory:**

- Erasing (discharging) cannot completely clear cells.
  - After thousands/millions of erase cycles, cells act "charged."
  - Leads to dead cells and "wear out" issues.
- This issue must be resolved for SSDs to be viable.

# Solution to NAND Wear-Out

Software-Defined Storage: Every SSD contains a complete computing system.

- FTL: Flash Translation Layer
  - **Wear Leveling**: Software manages blocks that may become problematic
  - Similar to a managed runtime (with garbage collection)




SSD internal structure with Flash Translation Layer (FTL)

# Interfaces of Storage Devices

## Common Storage Devices

- **Hard Disk Drive (HDD)**
  - **ATA (PATA):** Primarily used in older HDDs and optical drives; provides parallel data transfer.
  - **SATA:** Replaced PATA for faster serial data transfer; still used in HDDs and some SSDs.
- **Solid State Drive (SSD)**
  - **Interface:** SATA, NVMe (Non-Volatile Memory Express)
  - SATA SSDs use the same interface as HDDs but are faster due to no moving parts.
  - NVMe SSDs, utilizing PCIe (Peripheral Component Interconnect Express), are optimized for high-speed data access.
- **Optical Drives (CD/DVD/Blu-ray)**
  - **Interface:** ATA, SATA for internal drives
  - Usually used for media playback and data storage in a sequential access manner.

# Interfaces of Storage Devices (Cont.)

- **USB Flash Drives**
  - **Interface:** USB (Universal Serial Bus)
  - Plug-and-play, portable, used widely for data transfer and temporary storage.
- **Network-attached Storage (NAS)**
  - **Interface:** Ethernet, Wi-Fi
  - Allows data access over a network, suitable for shared storage in a networked environment.

# Storage Device Abstraction

# Disk Controller and ATA Interface

**ATA (Advanced Technology Attachment)**

- **IDE (Integrated Drive Electronics) Interface for Disk**
  - **Primary** Channel: `0x1f0 – 0x1f7`
  - **Secondary** Channel: `0x170 – 0x177`
- **Modern OS no longer manages disk scheduling:**
  - The OS only requests specific data blocks through the ATA interface.
  - The internal controller of the disk performs scheduling and optimization independently (e.g., seek optimization), without OS intervention.
- **Role of Device Driver:**
  - Sends commands via the ATA interface, specifying the location of the data block.
  - Does not involve specific scheduling algorithms; only responsible for data request transmission.

# Disk Controller and ATA Interface

```
void readsect(void *dst, int sect) {
    waitdisk();
    out_byte(0x1f2, 1);             // sector count (1)
    out_byte(0x1f3, sect);          // sector
    out_byte(0x1f4, sect >> 8);     // cylinder (low)
    out_byte(0x1f5, sect >> 16);    // cylinder (high)
    out_byte(0x1f6, (sect >> 24) | 0xe0); // drive
    out_byte(0x1f7, 0x20);          // command (read)
    waitdisk();
    for (int i = 0; i < SECTSIZE / 4; i++)
        ((uint32_t*)dst)[i] = in_long(0x1f0); // data
}
```

# Abstraction of Storage Devices

## Characteristics of Disk (Storage Device) Access

1. **Access in Data Blocks**
   - Data is accessed in **blocks**, not individual bytes.
   - Transfer has a ”minimum unit size” and does not support arbitrary access.
   - Optimal transfer mode depends on the type of storage (HDD vs. SSD).
2. **High Throughput**
   - Data transfer is managed using **Direct Memory Access (DMA)** for efficient handling.
3. **No Direct Access by Applications**
   - Access is typically managed through the **file system**, which maintains data structures on the disk.
   - Concurrency is handled as multiple processes access the file system simultaneously.

## Comparison with Other Devices (e.g., Terminal, GPU)

- **Terminal:** Small data volumes with direct streaming transfer.
- **GPU:** Large data volumes transferred using DMA, optimized for high throughput.

# Linux Block I/O Layer

## Interface between the File System and Disk Devices

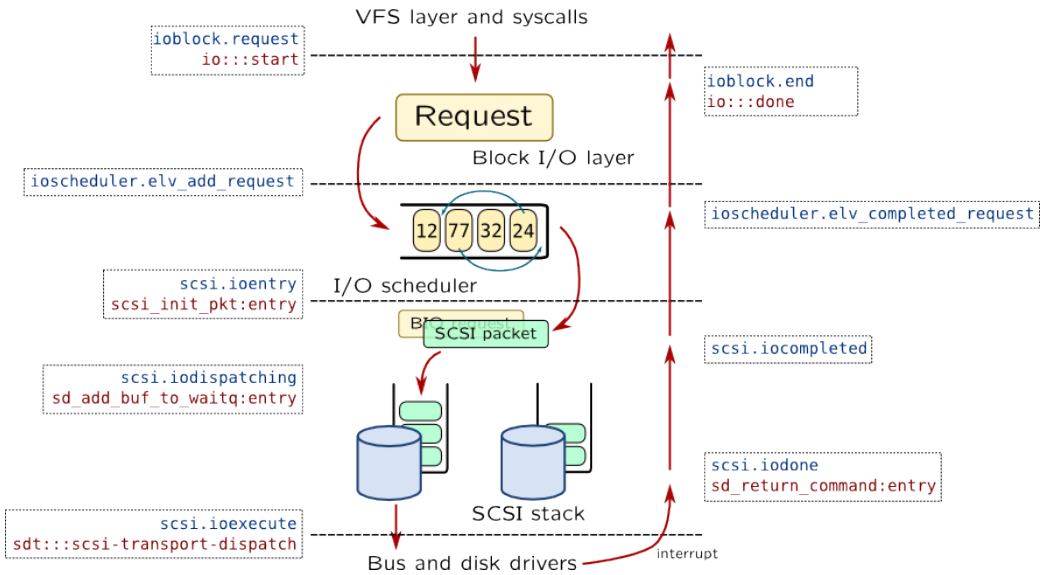



Diagram showing the Linux Block I/O layer, including the I/O scheduler and SCSI stack.

- **Request:** The file system sends a block request to the I/O layer.
- **I/O Scheduler:** Orders requests for optimal performance, previously using methods like the elevator algorithm.
- **SCSI Stack:** Converts requests into SCSI commands and handles communication with disk drivers.
- **Bus and Disk Drivers:** Final layer handling the actual data transfer between the system and the storage device.

# Block I/O: The Foundation of Persistence

**File System**

- Persistent data structures built on top of the Block I/O API

**File System Implementation**

- `bread`: Read blocks from disk
- `bwrite`: Write blocks to disk
- `bflush`: Flush block cache to disk
- Supports file and directory operations

# Why do we need file system?

# Sharing Devices Among Applications

**Terminal**

- Multiple processes printing concurrently – how to ensure no mix-up? (see `man 2 printf`)
- Multiple processes reading concurrently can lead to conflicts.
  - Race condition – who wins, who loses (sometimes acceptable).
  - Background processes may receive `SIGTTIN` when trying to read from the terminal (RTFM).

**GPU (CUDA)**

- Each CUDA application is a series of CUDA API calls.
  - `cudaMemcpy`, kernel calls, etc.
- All scheduling and isolation are handled by the device driver.
  - Kernels must wait for available thread warps to execute, and control is returned upon completion.

# Sharing Devices Among Applications (Cont.)

**Persistent Data on Disk**

- **Program Data**
  - Executable files and dynamic libraries
  - Application data (high-resolution images, cutscenes, 3D models, etc.)
- **User Data**
  - Documents, downloads, screenshots, replay files, etc.
- **System Data**
  - Manpages
  - System configuration files

A byte sequence is not an ideal abstraction for disks.

- Should all applications share a disk? A single program bug could compromise the entire operating system.

# File System: Virtual Disk

## File System: Design Goals

1. Provide a reasonable API for multiple applications to share data
2. Offer some level of isolation to ensure that malicious or erroneous programs cannot cause widespread harm

## Virtualization of "Storage Device (Byte Sequence)"

- Disk (I/O device) = a readable/writable byte sequence
- Virtual Disk (file) = a dynamically readable/writable byte sequence
  - **Name Management**
    - Naming, indexing, and traversal of virtual disks
  - **Data Management**
    - `std::vector<char>` (random access/write/resize)

# Virtual Drvie: Namespace Management

# Virtual Drive: Namespace Management

**Organizing Information:** Structure virtual drives (files) into a hierarchical system.

**Key Points:**

- Organize virtual drives (files) into a hierarchical structure for easy access.
- Enable efficient retrieval of data by maintaining logical order.
- Example: Similar to a library categorization system, files can be arranged based on names or categories for quick access.

A library categorization system example, which helps in quick location and retrieval of books, analogous to file system organization.

# Organizing Virtual Dsik

**Directory Tree**

- Store logically related data in nearby directories
  - Using Locality of Information

# File System "Root"

- **Windows:** Each device (driver) is a separate tree
  - New drive letters are assigned for new devices
    - Simple, direct, convenient, but can be cumbersome (e.g., `game.iso`)
- **UNIX/Linux**
  - Only one root, `/`
  - What about the second device?

An early computer setup demonstrating the concept of distinct device roots in legacy systems.

# In-Class Quiz

# Mounting in Directory Trees

**UNIX:** Allows any directory to be **mounted** as a representation of a device's directory tree.

- Highly flexible design:
  - Devices can be mounted anywhere in the desired location.
  - "Mount points" during Linux installation:
    - `/`, `/home`, `/var` can each be separate drvie devices.

## Mount System Call

```
int mount ( const char *source, const char *target, const
    char *filesystemtype,
          unsigned long mountflags, const void *data );
```

- Example: `mount /dev/sdb /mnt`
- Linux `mount` tool can automatically detect the filesystem

# Mounting a File

Mounting a file introduces an interesting loop:

- File = Virtual drive on a hard drive
- Mounting a file = Mounting a virtual drive on a virtual drive

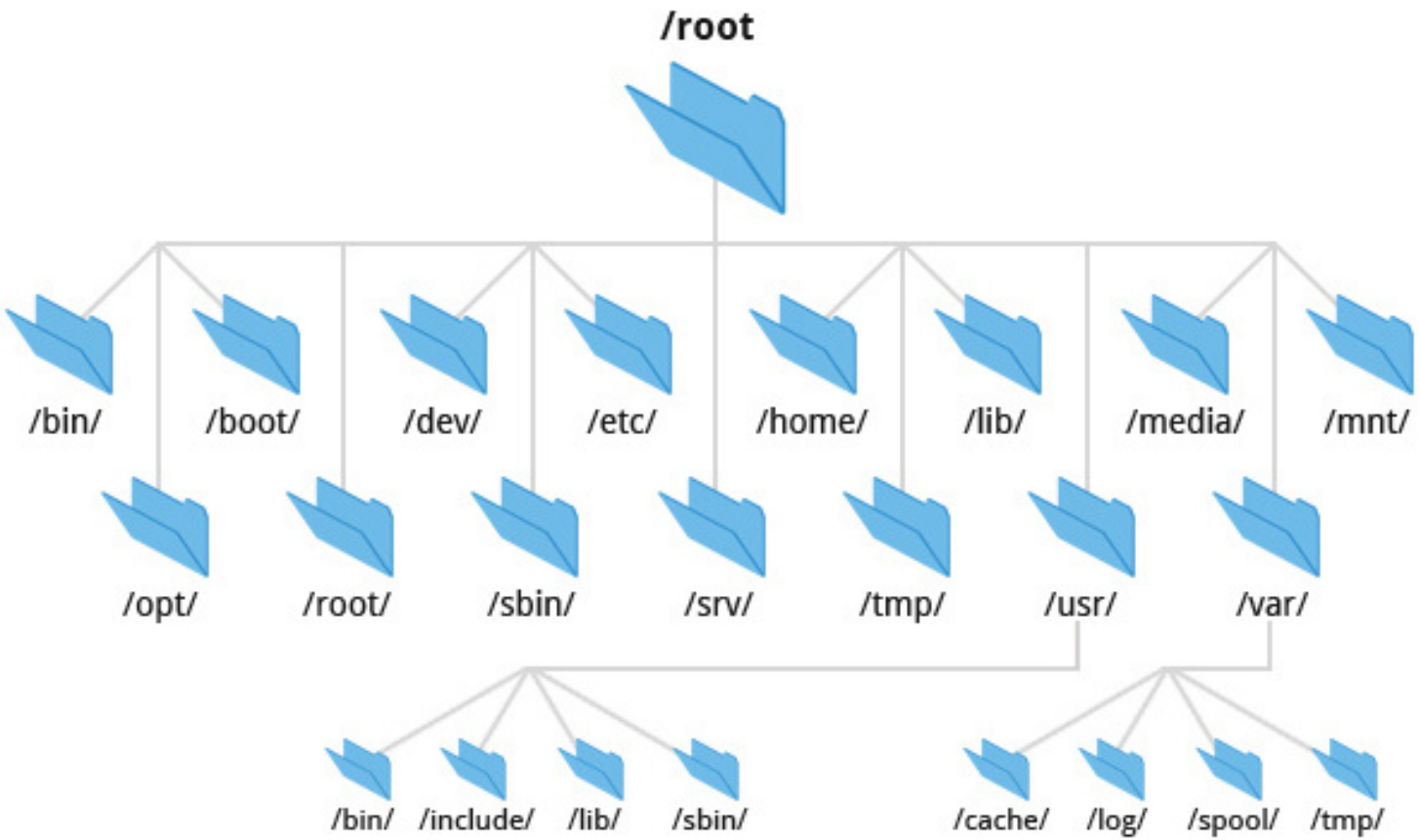
Linux handling:

- Create a **loopback** device
  - Device driver translates device read/write operations to file read/write operations

# Filesystem Hierarchy Standard (FHS)

FHS enables *software* and *users* to predict the location of installed files and directories.

- Example: macOS has a UNIX kernel (BSD) but does not follow the Linux FHS.



# Directory API (System Calls)

# Directory Management: Create/Delete/Traverse

This is straightforward:

- `mkdir`
  - Creates a directory
  - Allows setting access permissions
- `rmdir`
  - Deletes an empty directory
  - No system call for "recursive delete"
    - (If achievable at the application level, it is not implemented at the OS level)
    - `rm -rf` traverses directories, deleting each item (try `strace`)
- `getdents`
  - Returns `count` number of directory entries (used by `ls`, `find`, `tree`)
  - Dot-prefixed entries are returned by the system call, but `ls` does not display them by default

# More User-Friendly Directory Access

Appropriate API + Programming Language

- **Globbing**
- This is a user-friendly approach
  - C++ filesystem API is quite difficult to use

# Hard Links

**Requirements:** The system may have multiple versions of the same library.

- Examples: `libc-2.27.so`, `libc-2.26.so`, ...
- Also requires a "current version of libc"
  - Programs need to link to `libc.so.6` to avoid duplicating the file.

**Hard Link:** Allows a file to be referenced by multiple directory entries.

- Directories only store pointers to the file data.
- **Limitations:**
  - Cannot link directories
  - Cannot link across file systems

Most UNIX file systems use hard links for files (check with `ls -i`).

- System call to delete a link is `unlink` (reference count).

# Symbolic Links

**Symbolic Link:** Stores a ”jump pointer” in a file.

- Symbolic links are also files.
  - When referencing this file, it points to another file.
  - Stores the absolute/relative path of another file as text in the file.
  - Can link across file systems, can link directories, etc.
- Similar to a ”shortcut.”
  - It doesn’t matter if the linked target currently exists.
  - Examples:
    - `~/usb` ⇒ `/media/xinliu-usb`
    - `~/Desktop` ⇒ `/mnt/c/Users/xinliu/Desktop` (WSL)

`ln -s` to create symbolic links.

- `symlink` system call.

# Symlinks: You Can Even Make a Galgame


(Many games of that era looked like this.)

**Game = State Machine**

- Represent the current state with the **current directory**.
- Use **symbolic links** to encode transitions between states.

Have A Try: ggmaker.py

# Symlinks: You Can Even "Fake" a Filesystem!

## Nix

- **Store all versions of all packages in one place.**
- Example path:
  `/nix/store/b6gvz...54ad73z-firefox-33.1`
  - Then use **symbolic links** to assemble a fully virtual environment.
  - It is **fully deterministic**: decided by the package **hash**.
- Compose **arbitrary** environments on demand:
  - `nix-shell -p python3 nodejs`

# Process "Current Directory"

**Working/Current Directory**

- `pwd` command or `$PWD` environment variable can be used to check.
- `chdir` system call for modification.
  - Corresponds to `cd` in the shell.
  - Note that `cd` is a shell built-in command.
    - It does not exist in `/bin/cd`.

**Question:** Do threads share a working directory, or does each have its own?

# File API (System Calls)

# Review: Files and File Descriptors

**Files: Virtual Drives**

- A drive is a ”sequence of bytes.”
- Supports read/write operations.

**File Descriptors: Pointers for Process Access to Files (Operating System Objects)**

- Obtained through `open` or `pipe`.
- Released through `close`.
- Duplicated through `dup`/`dup2`.
- Inherited during `fork`.

# File Access Offset (Seek Pointer)

File read/write operations come with a ”seek pointer,” so it’s unnecessary to specify the read/write location every time.

- This feature makes it convenient for programmers to access files sequentially.

**Example:**

- `read(fd, buf, 512);` - Reads the first 512 bytes.
- `read(fd, buf, 512);` - Reads the next 512 bytes.
- `lseek(fd, -1, SEEK_END);` - Moves to the last byte.
  - *so far, so good*

# Offset Management: Not So Simple

File descriptors are inherited by child processes during `fork`.

**Should parent and child processes share an offset, or should each have its own?**

- This choice determines where the offset is stored.

**Consider application scenarios:**

- When parent and child processes write to a file simultaneously
  - Each has its own offset → parent and child need to coordinate offset updates
    - (Race condition)
  - Shared offset → OS manages the offset
    - Although shared, the OS ensures the atomicity of `write` operations ✓

# Offset Management: Behavior

Every API in the operating system may interact with other APIs

1. During `open`, a unique offset is obtained.
2. During `dup`, two file descriptors share the offset.
3. During `fork`, the parent and child processes share the offset.
4. During `execve`, the file descriptor remains unchanged.
5. For files opened with `O_APPEND` mode, the offset is always at the end (regardless of `fork`).
   - Modification of the file offset and the write operation are performed as a single atomic step.

This is also one reason why `fork` is often criticized.

- (At the time) a good design may become a burden in the evolution of the system.
- Today's `fork` might be considered "overloaded"; *A fork() in the road*.

# File Allocation Table (FAT)

# What is File System Implementation?

- Implement all file system APIs on a block device (I/O device)
  - `bread(int id, char *buf);`
  - `bwrite(int id, const char *buf);`
    - Assumes all operations complete in synchronized queue
    - (Can be implemented with queues at block I/O layer)

**Directory/File API**

- `mkdir`, `rmdir`, `link`, `unlink`
- `open`, `read`, `write`, `stat`

# Back to Data Structures Class...

- A file system is essentially a data structure (Abstract Data Type; ADT)
  - Just with different assumptions than those in data structures class

**Assumptions in Data Structures Class:**

- Von Neumann machine
- Random Access Memory (RAM)
  - Word Addressing (e.g., 32/64-bit load/store)
  - The cost of each instruction is $O(1)$
    - Memory hierarchy challenges this assumption (cache-unfriendly code may encounter performance issues)

**Assumptions in File Systems:**

- Block-based (e.g., 4KB) access; building a RAM model on disk is entirely unrealistic

## Device Abstraction Provided by Block Device

```
struct block blocks[NBLK]; // Disk
void bread(int id, struct block *buf) {
    memcpy(buf, &blocks[id], sizeof(struct block));
    }
void bwrite(int id, const struct block *buf) {
    memcpy(&blocks[id], buf, sizeof(struct block));
    }
```

## Allocation and Deallocation in `bread`/`bwrite` (Similar to PMM)

```
int balloc(); // Return an available data block
void bfree(int id); // Release a data block
```

# Implementation of Data Structures (cont'd)

- Virtualizing the disk with `balloc`/`bfree`
  - File = `vector<char>`
    - Maintains using linked lists, indexes, or any data structure
    - Supports arbitrary position modifications and resizing
- Implementing directories based on files
  - `Directory file`
    - Interprets `vector<char>` as `vector<dir_entry>`
    - Stores continuous bytes for each directory entry

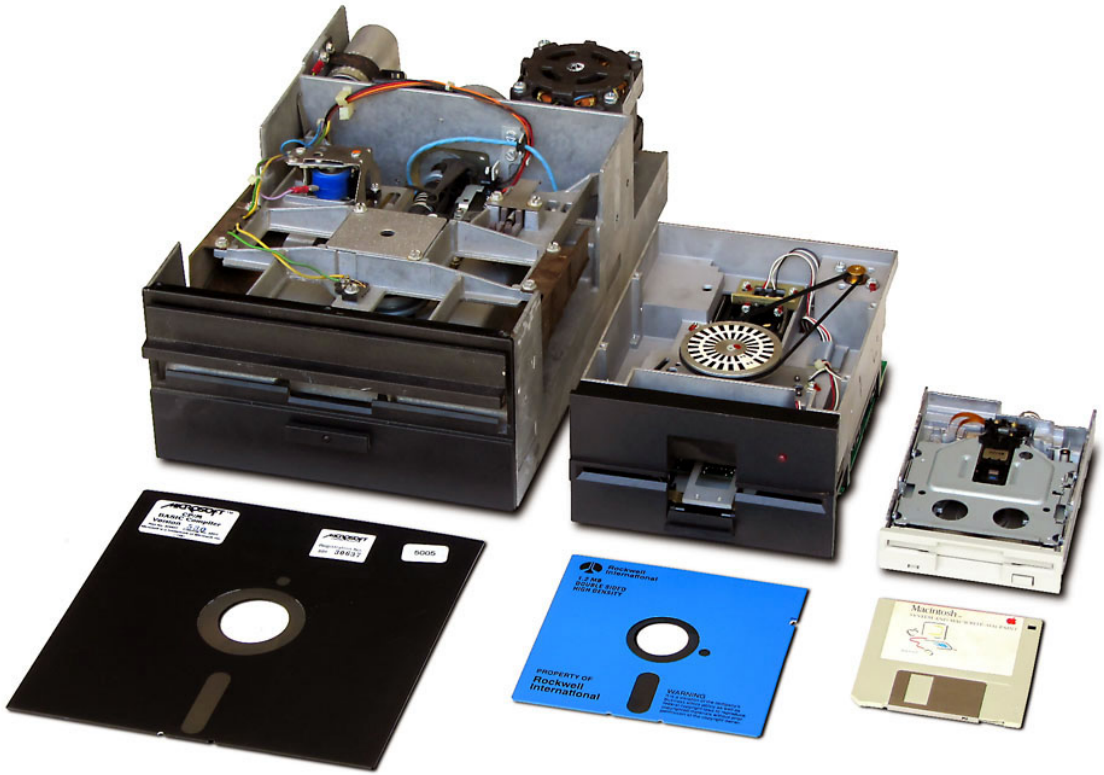
# Implementation of a Simple File System

We can implement a simple file system by treating files as sequences of blocks and using data structures to manage them.

- **File Representation**
  - Each file is represented by an **inode** (index node)
  - The inode contains metadata and pointers to data blocks
- **Inode Structure**
  - File type, permissions, owner, timestamps, etc.
  - Pointers to data blocks (direct, indirect, double indirect)
- **Data Blocks**
  - Fixed-size blocks storing the actual file content
  - Managed using block allocation algorithms

# Back to 1980: The 5.25” Floppy Disk

- **5.25” Floppy Disk:** Single-sided, 180 KiB capacity
- **Storage Specifications:**
  - 360 sectors, each with 512 bytes (sectors)
- **Question:**
  - What kind of data structure would be suitable to implement a file system on such a device?



# Files in the FAT File System

- **Characteristics:**
  - Relatively small file system
  - Tree-like directory structure
  - Primarily consists of small files (within a few blocks)
- **File Implementation:**
  - Linked list of `struct block *`
  - Complex high-level data structures are inefficient for this purpose

# Using Linked Storage Data: Two Designs

1. Place the pointer after each data block
   - **Advantage:** Simple implementation, no need for additional storage space.
   - **Disadvantage:** Data size is not necessarily $2^k$; pure `lseek` requires reading entire block data.
2. Centralize pointers in a specific area of the file system
   - **Advantage:** Better locality; faster `lseek`.
   - **Disadvantage:** Centralized pointer data corruption could lead to data loss.

**Question:** Which design's drawbacks are fatal and difficult to resolve?

# Centralized Storage of All Pointers

- Centralized pointers are prone to damage? Store *n* copies to mitigate!
- Example: FAT-12/16/32 (FAT entry represents the size of the ”next pointer”)

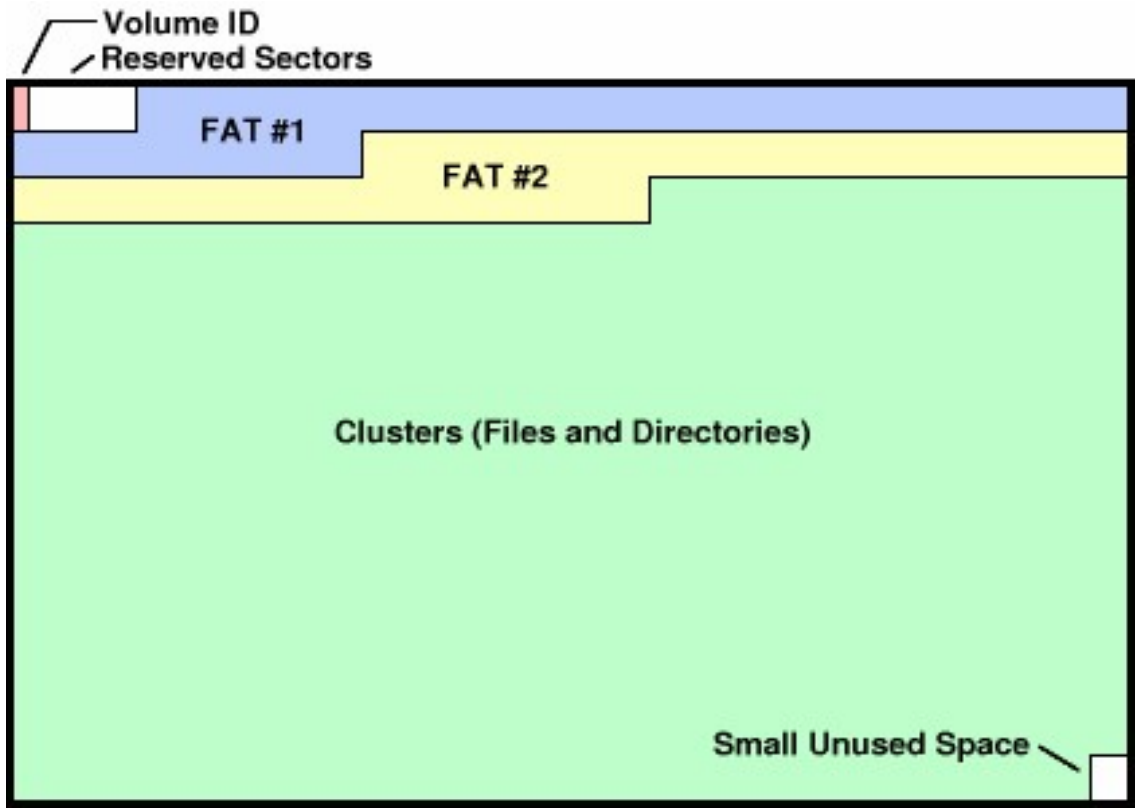



Figure: Illustration of FAT structure showing centralized pointer storage in FAT tables (FAT #1, FAT #2) for file clusters.

# FAT: Linked Storage Files

## RTFM

- Structure of FAT's "next" array:
  - `0`: free; `2 ... MAX`: allocated;
  - `0xFFFFFFF7`: bad cluster; `0xFFFFFFF8 - 0xFFFFFFFE`, `-1`: end-of-file

| FAT32 | Comments |
|---|---|
| 0x00000000 | Cluster is free. |
| 0x00000002 to MAX | Cluster is allocated. Value of the entry is the cluster number of the next cluster following this clus |
| (MAX + 1) to 0xFFFFF6 | Reserved and must not be used. |
| 0xFFFFFF7 | Indicates a bad (defective) cluster. |
| 0xFFFFFF8 to 0xFFFFFFE | Reserved and should not be used. |
| 0xFFFFFFFF | Cluster is allocated and is the final cluster for the file (indicates *end-of-file*). |

Table: FAT Entry Values and Their Meanings

# Directory Tree Implementation: Directory Files

Using regular files to store the ”directory” data structure

- **FAT**: Directory is a collection of fixed-length 32-byte directory entries.
- The operating system parses and treats directory entries marked as ”directory” as actual directories.
  - A sequence of directory entries can store long filenames.
- **Thought Exercise**: Why not store metadata (size, filename, etc.) at the head of `vector<struct block *> file`?

# FAT: Performance and Reliability

## Performance

- **+** Small files are ideal
- **-** However, random access for large files is inefficient
  - A 4 GB file jumping to the end (4 KB clusters) requires $2^{20}$ chain 'next' operations
  - Caching can partially alleviate this issue
- In the FAT era, sequential access performance on disks was better
  - Long-term disk usage leads to fragmentation
    - `malloc` also causes fragmentation, but the performance impact is less significant

## Reliability

- Maintain multiple copies of FAT to prevent data loss
  - Unexpected synchronous write-offs
  - Damaged clusters are marked in the FAT

# ext2 and UNIX File System

# ext2/UNIX File System

Centralized storage of file/directory metadata as objects

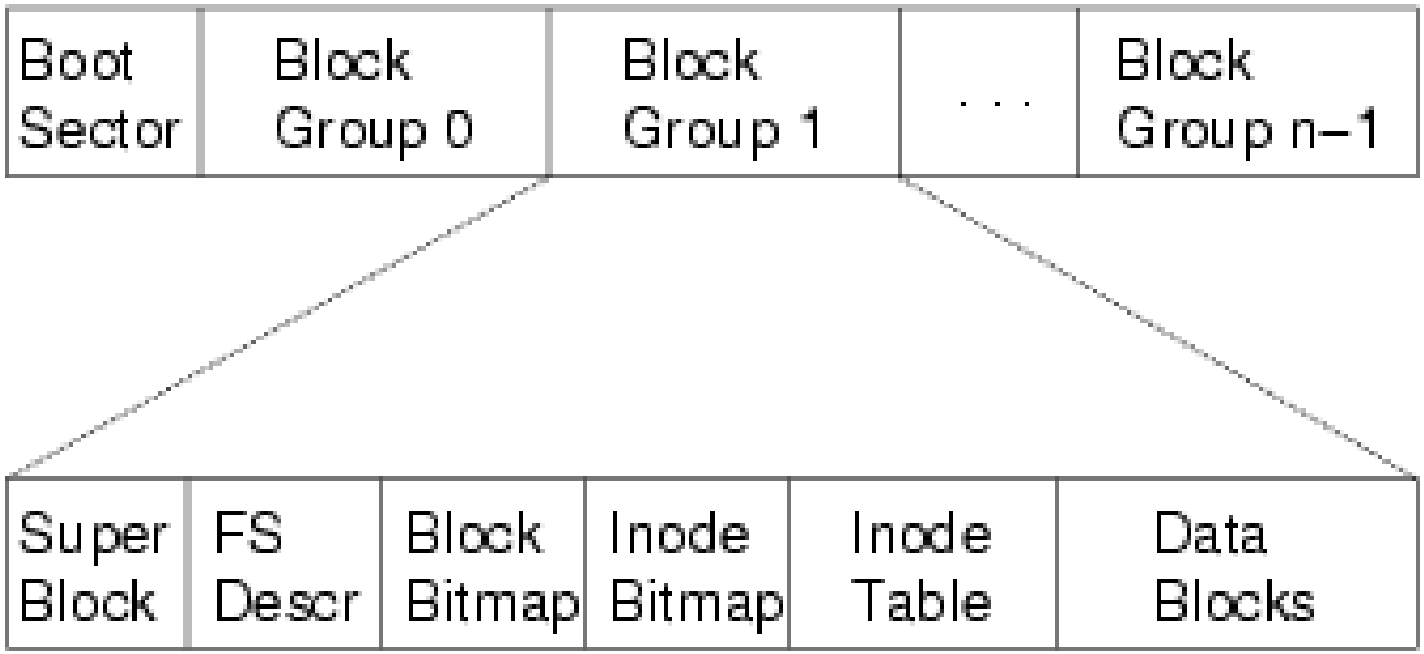
- Enhances locality (easier for caching)
- Supports linked files

Distinguishing fast/slow paths for different file sizes

- For small files, arrays should be used
  - Skips linked list traversal
- For large files, trees should be used (e.g., B-Tree, Radix-Tree)
  - Enables fast random access

# ext2: Drive Image Format

## Dividing the drive into groups




**Superblock**: Filesystem metadata

- Number of files (inodes)
- Block group information
  - `ext2.h` contains everything you need to know

# ext2 Directory Files

Similar to FAT: Establishes a directory structure on files

- Note that inodes are stored in a unified way
  - Directory files store a key-value mapping of file names to inode numbers

# ext2: Performance and Reliability

For large files, random read/write performance is significantly improved (O(1)):

- Supports linking (reduces space waste to some extent)
- Inodes are stored continuously on disk, which facilitates caching/prefetching
- Fragmentation remains an issue

However, reliability is still a major concern:

- Damage to the data block storing the inode can be very serious

# Takeaways

## Questions Answered in This Lecture

- **Q**: How is the ”current state” of a state machine and persistent state stored?

## Take-away Messages

- **1-Bit Information Storage**
  - Magnetic media (tape, disk), pits (optical disk), and electrical (Flash SSD)
  - Different types of storage devices with varied characteristics
- **Rethinking ”Storage of State”**
  - With NVM (Non-Volatile Memory), the state of main memory remains intact without power loss
    - Note: Cache/buffer storage is still volatile

# Takeaways

## Questions Answered in This Lecture

- **Q:** How to design a file system that allows applications to share storage devices?

## Takeaway Messages

- Two Main Components of File System
  - Virtual Drive (File)
    - Functions: `mmap`, `read`, `write`, `lseek`, `ftruncate`, ...
  - Virtual Drive Naming and Management (Directory Tree and Links)
    - Functions: `mount`, `chdir`, `mkdir`, `rmdir`, `link`, `unlink`, `symlink`, `open`, ...