Lecture 19: File System API

(Namespace, Directory, FAT, ext2)

Xin Liu

Florida State University

xl24j@fsu.edu

COP 4610 Operating Systems

https://xinliulab.github.io/cop4610.html

November 13, 2024

Review

- Abstraction of I/O devices
  - Device layer: I/O devices (registers and protocols)
  - Driver layer: read / write / ioctl
  - Physical layer: how to use magnetic / pit / electrical to store 1 bit of data
  - Block device layer: block read/write

Question to Answer in This Lecture

- **Q:** How can applications share storage devices?

Main Content of This Lecture

- File system APIs
  - Namespace
  - Directory
- Classic File Systems
  - File Allocation Table (FAT)
  - Second Extended Filesystem (ext2)

Why do we need file system?

Sharing Devices Among Applications

- Multiple processes reading / writing concurrently can lead to conflicts.
  - Race condition
  - A single program bug could compromise the entire operating system
- Should all applications share a drive?

File System: Virtual Drive

Design Goals of File System:

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)"

- Drive (I/O device) = a readable/writable byte sequence
- Virtual Drive (file) = a dynamically readable/writable byte sequence
  - **Namespace Management**
    - Naming, indexing, and traversal of virtual drives
  - **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

- **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.

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

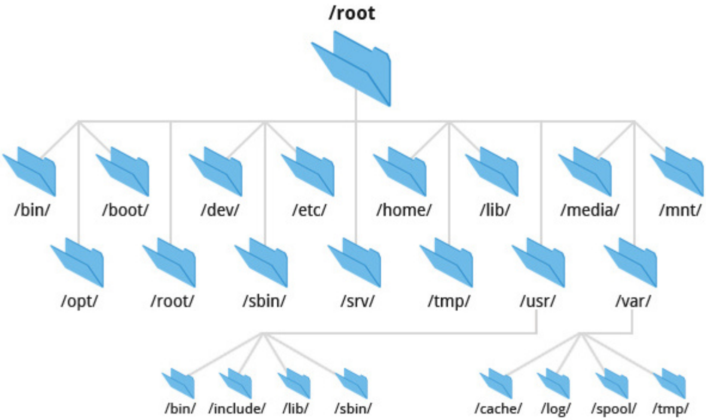
Linux handling:

- 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.

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 $\rightarrow$ parent and child need to coordinate offset updates
    - (Race condition)
  - Shared offset $\rightarrow$ 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

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

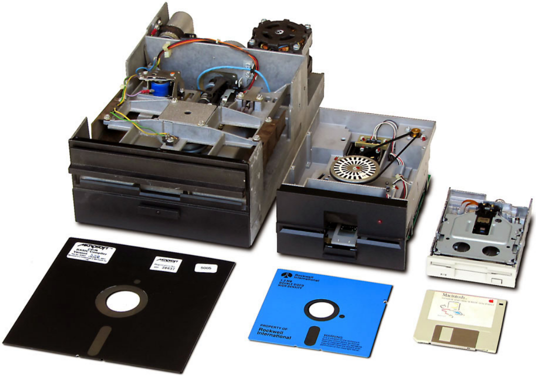
Implementation of a Simple File System

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

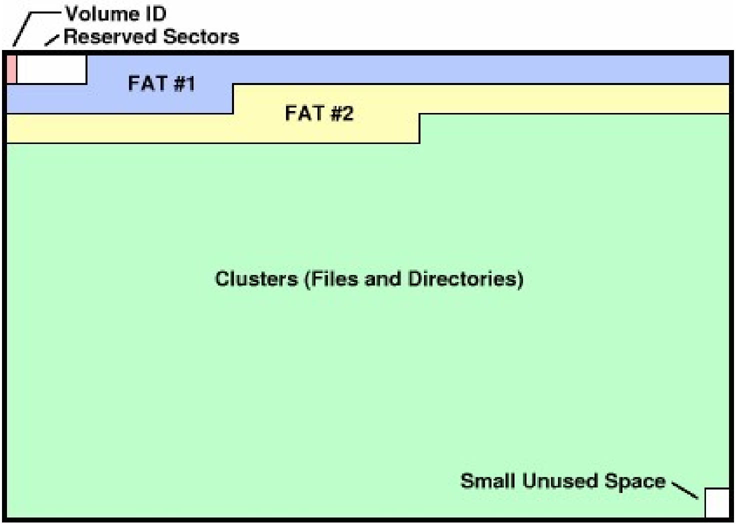
Using Linked Storage Data: Two Designs

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”)



FAT: Linked Storage Files

- Structure of FAT's "next" array:
  - `0`: free; `2 ... MAX`: allocated;
  - `0xFFFFFF7`: bad cluster; `0xFFFFFF8 – 0xFFFFFFE`, `−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 cluster.
(MAX + 1) to 0xFFFFFF6	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

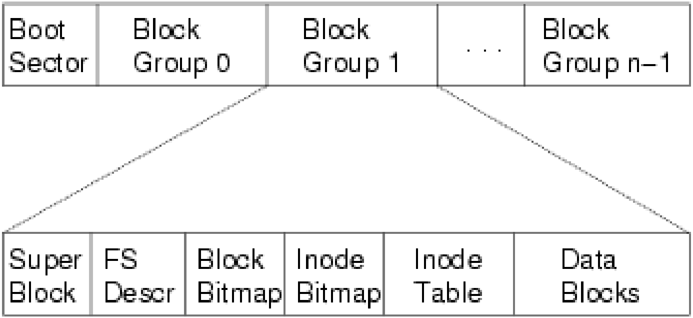
- Enhances locality (easier for caching)
- Supports linked files

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: Disk Image Format

Dividing the disk 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

This Lecture's Key Question

- **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`, ...