Operating Systems - File Systems and Virtual File System (VFS)

Introduction to File Systems

- A file system is a crucial part of an operating system responsible for managing and organizing data storage on devices like hard disks, SSDs, and flash drives.
  It abstracts the physical details of storage devices and presents a logical view to users and applications.
  Main functions include file creation, access, modification, and deletion. File systems also manage metadata, space allocation, and data protection.

File Naming

- Each file is uniquely identified by a name, which may include extensions and path information.
  Key points:
  • File names can vary by operating system (case-sensitive in UNIX, case-insensitive in Windows).
  • Naming rules define allowed characters and lengths.
  • Paths can be absolute (starting from root) or relative (based on current directory).
  • Example: /home/user/document.txt or C:\Users\Admin\file.docx

File Structure

- File structure refers to the logical organization of data within a file. Common structures include:
  - Byte sequence: files are treated as sequences of bytes (used in UNIX).
  - Record sequence: files are composed of fixed or variable-length records (used in databases).
  - Tree-structured files: used in hierarchical storage (e.g., XML, JSON).

  The structure affects how data is accessed and interpreted by programs.

File Types

- File systems support multiple types of files, including:
  - Regular files: contain user data (text, binary, media).
  - Directory files: store information about other files.
  - Character special files: represent serial I/O devices like terminals.
  - Block special files: represent storage devices like disks.
  - Symbolic links: point to other files for quick access.

  Example: UNIX uses file type flags in inodes to differentiate file types.

File Access Methods

- Access methods define how data within files is read or written.
  • Sequential access: Data is processed in a linear order (e.g., reading a log file).
  • Direct access: Random access to any part of the file using byte offsets (used in databases).
  • Indexed access: An index structure maps keys to specific file blocks (used in databases and file allocation tables).
  Efficient access methods improve performance and flexibility.

File Attributes

- Attributes provide metadata about each file. They describe properties and control access permissions.
  Common file attributes include:
  • Name, Type, Size
  • Location (disk address or inode number)
  • Protection and permissions (read, write, execute)
  • Time stamps: creation, last modification, and last access times
  • Owner and group information
  These attributes are stored in data structures like File Control Blocks (FCBs) or inodes.

File Operations

- The file system supports a variety of operations to manage files:
  - Create: Allocate space and initialize metadata.
  - Open/Close: Establish and release connections to files.
  - Read/Write: Transfer data between user memory and file system.
  - Delete: Remove files and reclaim space.
  - Seek: Reposition file pointer for direct access.

  Example system calls: open(), read(), write(), close(), unlink().

Example Program Using File-System Calls

- Example in C:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    int fd = open("example.txt", O_CREAT | O_WRONLY, 0644);
    write(fd, "Hello OS File System!", 21);
    close(fd);
    return 0;
}
```

Explanation:

• open(): Creates or opens a file.
• write(): Writes data to the file.
• close(): Closes the file descriptor.

This demonstrates low-level file I/O operations handled by the kernel.

File-System Layout

- A typical disk-based file system is divided into several sections:
  1. Boot Block: Contains code for system startup.
  2. Superblock: Contains metadata like file system size, block size, and inode count.
  3. Inode Table: Stores information about files such as ownership and permissions.
  4. Data Blocks: Contain the actual file contents.
  5. Free Space Management: Tracks available disk blocks.

  Diagram: Boot Block → Superblock → Inodes → Data Blocks

Implementing Files

- Files can be implemented using different allocation methods:
  1. Contiguous Allocation: Stores files in consecutive disk blocks.
     - Advantage: Fast sequential access.
     - Disadvantage: External fragmentation.
  2. Linked Allocation: Uses pointers between blocks.
     - Advantage: Flexible file growth.
     - Disadvantage: Slow random access.
  3. Indexed Allocation: Uses index blocks that store addresses of file blocks.
     - Advantage: Efficient random access.

  File systems like ext4 and NTFS use hybrid approaches combining these techniques.

Introduction to Virtual File System (VFS)

- A Virtual File System (VFS) provides an abstraction layer between user-level operations and the underlying physical file systems.
  It allows multiple file systems (e.g., ext4, FAT32, NFS) to coexist and be accessed through a common interface.
  VFS simplifies system call management and improves portability across operating systems.

Role of the Virtual File System

- The VFS manages communication between user programs and specific file system implementations.
  Roles include:
  - Abstracting differences between file systems.
  - Managing mount points and file descriptors.
  - Providing uniform APIs for file operations (open, read, write).
  - Enabling dynamic file system mounting/unmounting.
  - Maintaining consistency and caching mechanisms for performance.

VFS Data Structures

- Core data structures of VFS include:
  - Superblock: Represents a mounted file system and contains metadata.
  - Inode: Represents an individual file's metadata and pointers to data blocks.
  - Dentry: Represents directory entries and pathname components.
  - File Object: Represents an open file instance used by processes.

  These structures are interconnected, enabling VFS to manage files across different storage systems.

Filesystem Types

- Common file system types handled by VFS include:
  - Disk-based: ext4, FAT32, NTFS
  - Network-based: NFS, CIFS
  - Memory-based: tmpfs, ramfs
  - Pseudo filesystems: procfs, sysfs

  Each type implements its own operations but interacts uniformly through VFS interfaces.

  Example: Linux allows mounting NFS and ext4 simultaneously using VFS abstraction.

Filesystem Handling

- VFS handles file system registration and mounting:
  - Each file system registers its structure and methods via file_system_type.
  - VFS maintains a mount table for active file systems.
  - Mounting associates a filesystem with a directory path.
  - Example: mount -t nfs server:/shared /mnt

  VFS routes file operations through function pointers defined in file_operations structures.

Pathname Lookup

- Pathname lookup converts a textual path into an inode reference.
  Steps:
  1. Parse the path string component by component.
  2. Traverse directory hierarchy using dentries.
  3. Retrieve the inode corresponding to each path component.
  4. Cache results in the dentry cache (dcache) for faster future access.
  Diagram: /home/user/file.txt → [dentry → inode → file data]

Implementation of VFS System Calls

- System calls like open(), read(), write(), and close() are implemented in VFS as generic functions.
  VFS translates these generic calls into specific file system operations.
  Example: open() → vfs_open() → ext4_open() or nfs_open()
  This modular design improves flexibility, allowing new file systems to integrate seamlessly without modifying kernel code.

File Locking

- File locking ensures synchronization and data consistency when multiple processes access the same file.
  Types of Locks:
  • Shared Lock: Multiple readers can access simultaneously.
  • Exclusive Lock: Only one writer can access the file.
  Locking Modes:
  • Advisory: Voluntary, cooperative locking mechanism.
  • Mandatory: Enforced by the operating system.
  Example: UNIX uses fcntl() for advisory locks, while Windows uses LockFile().
  File locking prevents race conditions and ensures database integrity.