Virtual File System (VFS)

- A virtual file system (VFS) is an operating system kernel component that provides a standard interface for accessing diverse file systems, allowing them to be treated uniformly.
- It acts as an abstraction layer, separating the generic file system operations (like open(), read(), write()) from their specific implementations for different file systems (e.g., ext4 or NTFS).
- This promotes modularity, allows new file systems to be added easily, and provides a transparent way for applications to access local and network storage without knowing the underlying details of each system.
- Key Concepts and Functions
    - Abstraction Layer:
        - The VFS acts as a middleware between the kernel's system call interface and the actual file systems, hiding the complexities of each concrete file system.
    - Uniform Interface:
        - It presents a single, unified Application Programming Interface (API) to users and applications, simplifying file management regardless of whether the data is on a local hard drive or a remote network server.
    - Modularity and Extensibility:
        - VFS allows operating systems to support many different file systems simultaneously by defining a standard set of objects (like superblocks, inodes, and file objects) and operations.
        - Any new file system can be integrated by implementing these VFS objects and operations.
    - Transparency:
        - Users and applications don't need to be aware of the underlying file system format or the physical location of the data; the VFS handles all the details, making access appear seamless.
    - Integration:
        - VFS facilitates the integration of local file systems (like ext4, NTFS) with network file systems (like NFS) and other special types, such as memory-based procfs (for process information) or sysfs (for kernel and device information).
- How it Works?
    - System Call: An application makes a file-related system call (e.g., open("myfile.txt")).
    - VFS Routing: The VFS receives the call and resolves the path to identify the correct target file system.
    - Delegation: The VFS then delegates the actual task to the specific driver or module responsible for that file system.
    - Hardware Operation: The concrete file system implementation performs the necessary input/output (I/O) operations on the disk or network.
- Benefits
    - Simplicity for Users:

        - Applications interact with a consistent interface, making it easier to use different storage types.
    - Flexibility for OS:
        - New file system types can be supported without modifying the core kernel or applications.
    - Code Reuse:
        - VFS handles generic system calls, allowing implementers to focus on the specific details of their file system.

VFS Objects and Their Data Structures

- The VFS is object-oriented.
- A family of data structures represents the common file model.
- The data structures are represented as C structures. The structures contain both data and pointers to filesystem implemented functions that operate on the data.
- The four primary object types of the VFS are
    - The superblock object, which represents a specific mounted filesystem.
    - The inode object, which represents a specific file.
    - The dentry object, which represents a directory entry, a single component of a path.
    - The file object, which represents an open file as associated with a process.
- An operations object is contained within each of these primary objects.
- These objects describe the methods that the kernel invokes against the primary objects. Specifically, there are
    - The super_operations object, which contains the methods that the kernel can invoke on a specific filesystem, such as read_inode() and sync_fs().
    - The inode_operations object, which contains the methods that the kernel can invoke on a specific file, such as create() and link().
    - The dentry_operations object, which contains the methods that the kernel can invoke on a specific directory entry, such as d_compare() and d_delete().
    - The file object, which contains the methods that a process can invoke on an open file, such as read() and write().
- The operations objects are implemented as a structure of pointers to functions that operate on the parent object.
- Each registered filesystem is represented by a file_system_type structure.
- This object describes the filesystem and its capabilities.
- Furthermore, each mount point is represented by the vfsmount structure. This structure contains information about the mount point, such as its location and mount flags.

IO Subsystem

- The I/O (Input/Output) subsystem in an operating system manages the flow of data between the central processing unit (CPU) and peripheral devices like keyboards, printers, and disks.
- Its primary roles include enabling communication and data transfer, handling device interfaces, and providing essential services like buffering, caching, scheduling I/O requests, and spooling to improve efficiency and manage different device speeds.
- It also ensures secure and coordinated access to devices, protecting against rogue programs.
- Key Functions of the I/O Subsystem
    - Device Management:
        - Handles communication with various input and output devices, such as keyboards, mice, monitors, disk drives, and printers.
    - Interface to Hardware:
        - Acts as an intermediary between the processor and devices, converting signals and handling data transfers through interfaces and I/O registers.
    - Buffering:
        - Uses temporary storage (buffers) to manage speed differences between devices and to handle data transfers of different sizes, ensuring smooth operation.
    - Caching:
        - Stores frequently accessed data in fast memory (cache) to speed up future access, such as when retrieving data from a hard disk or a web page.
    - Scheduling:
        - Determines the order in which I/O requests are executed, optimizing performance and ensuring fair access to devices.
    - Spooling:
        - A technique to manage devices that can only handle one task at a time, like printers, by placing requests in a queue and processing them when the device is free.
    - Security and Protection:
        - Safeguards the I/O devices and their data from unauthorized access or interference by malicious software or programs.
- How it Works
    - The kernel I/O subsystem builds upon the hardware and device drivers to provide these services to applications.
    - When a process needs to interact with an I/O device, it sends a request to the operating system.
    - The I/O subsystem then manages the request, using services like buffering and scheduling to efficiently transfer data between the process's memory and the device.