**VFS**

* The VFS acts as an intermediary layer between the user-space applications and the specific file systems. It provides a consistent set of system calls and data structures that allow programs to perform operations on files and directories without being concerned about the underlying implementation details of each file system.
* When an user makes a file-related request, such as opening, reading, or writing a file, the VFS translates these requests into operations that are specific to the relevant file system thus providing uniformity.
* It provides standardized interface for all user space applications by implementing an abstraction layer.
* The process how kernel interacts with the vfs layer is as follows:

The user-space program makes a system call, for example, **write(fd, buf, len)**.

**sjjssjsjs**

The result is propagated back through the layers to the user-space program.

The filesystem carries out the specific write operation using its internal logic.

The VFS forwards the request to the appropriate filesystem based on the file descriptor (fd).

The generic system call interacts with the Virtual Filesystem (VFS) layer.

At the kernel level, a generic system call is invoked to handle the request.

**SUPER BLOCK**

* The superblock contains metadata that describes the overall properties and configuration of the filesystem. This includes information such as the size of the filesystem, the total number of inodes and blocks, and the block size.
* Keeps track of the mount status, indicating whether the filesystem is currently mounted or not.
* Contains information about the underlying storage device, including the device identifier (e.g., a device name like **/dev/sda1**), which is crucial for the kernel to associate the mounted filesystem with a specific storage device.
* The superblock object is represented by struct super\_block and defined In <linux/fs.h>.

struct super\_block {

struct list\_head s\_list; **// Linked list for maintaining superblocks**

dev\_t s\_dev**; // Device identifier**

struct block\_device \*s\_bdev; **// Associated block device**

const struct super\_operations \*s\_op; **// Function pointers for filesystem operations**

...

};

**SUPERBLOCK OPERATION**

* The struct super\_operations defines a set of function pointers that constitute the operations available on a filesystem's superblock.
* alloc\_inode : Allocates and initializes an inode structure
* destroy\_inode : Frees resources associated with an inode
* write\_inode : Writes an inode to disk
* sync\_fs : Synchronizes the filesystem, ensuring that all pending changes are flushed to disk.
* Freeze\_fs : Freezes the filesystem, preventing further writes.
* Unfreeze\_fs : Unfreezes the filesystem, allowing writes again.
* Remount\_fs : Remounts the filesystem with new options. The third argument is a string containing the new options.

**INODE**

* In Unix-style filesystems, traditional inodes store information about a file or directory. This information includes details such as file type, permissions, owner, size, timestamps, and pointers to data blocks on disk.
* Inodes store metadata specific to individual files and directories, while the superblock stores metadata about the entire file system.
* An inode is identified by a unique inode number within the filesystem. The operating system uses this number to locate and manage the inode.
* The inode information is organized in an inode table, which is typically stored in a fixed location on the disk. The inode table is an array-like structure, and each entry in the table corresponds to a unique inode number.
* Inodes are dynamic and can be modified during the lifetime of a file. For example, when a file is modified, the inode's timestamps (modification time) are updated to reflect the changes.
* The struct inode\_operations defines a set of function pointers that constitute the operations available on inodes in a Unix-like file system.
* **create:** Creates a new inode associated with the given directory (dir) and directory entry (dentry) with the specified initial access mode (mode).
* **lookup:** Searches a directory represented by the inode (dir) for an inode corresponding to a filename specified in the given directory entry (dentry). This function is typically called during file lookup operations.
* **link:** Invoked by the link() system call to create a hard link of the file specified by old\_dentry in the directory represented by the inode (dir) with the new filename specified by dentry.
* **unlink**: Called from the unlink() system call to remove the inode specified by the directory entry (dentry) from the directory represented by the inode (dir). This operation deletes a hard link to the file.