Experiment No 1

Title : Linux File System Management
Aim : To demonstrate hoe Linux kernel implements and manages files.
Objective: To study following

1. Filesystem abstraction
2. Filesystem operations
3. Linux VFS
4. Overview of Linux I/O Management

Relevance: To get knowledge about Linux File System

Theory:

1. Filesystem Abstraction

A fileystem is a way to organize files and directories on storage devices such as hard disks, SSDs or flash memory. There are many types of filesystems (e.g. FAT, ext4, btrfs, ntfs) and on one running system we can have multiple instances of the same filesystem type in use. While filesystems use different data structures to organizing the files, directories, user data and meta (internal) data on storage devices there are a few common abstractions that are used in almost all filesystems. Some of these abstractions are present both on disk and in memory while some are only present in memory:

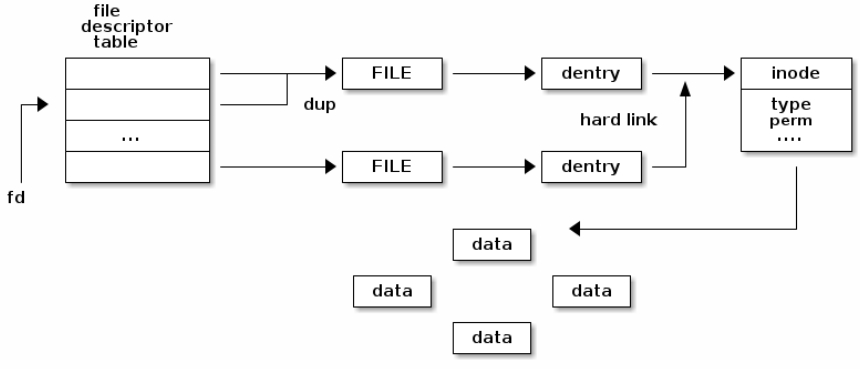
i. Superblock- The *superblock* abstraction contains information about the filesystem instance such as the block size, the root inode, filesystem size. It is present both on storage and in memory (for caching purposes).

ii. File- The *file* abstraction contains information about an opened file such as the current file pointer. It only exists in memory.

v. Inode- The *inode* is identifying a file on disk. It exists both on storage and in memory (for caching purposes). An inode identifies a file in a unique way and has various properties such as the file size, access rights, file type, etc.

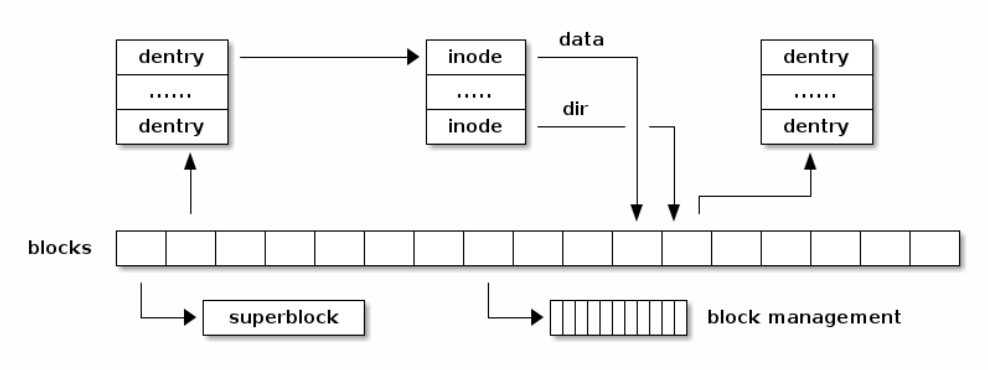
vi. Dentry- The *dentry* associates a name with an inode. It exists both on storage and in memory (for caching purposes).

The following diagram shows the relationship between the various filesystem abstractions as they used in memory:



Multiple file descriptors can point to the same *file* because we can use the **`dup()`** system call to duplicate a file descriptor.
Multiple *file* abstractions can point to the same *dentry* if we open the same path multiple times.
Multiple *dentries* can point to the same *inode* when hard links are used.

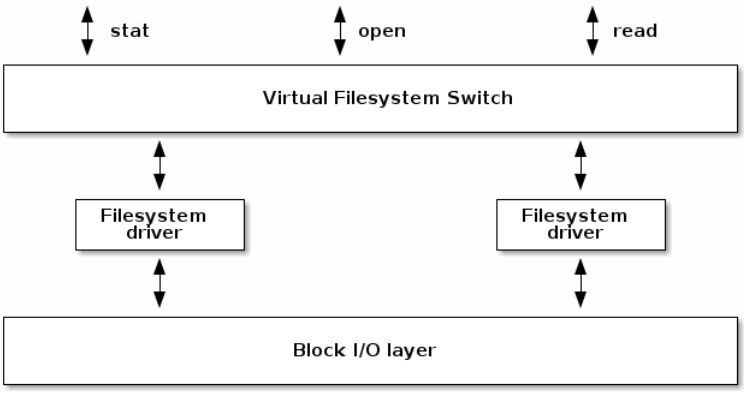
The following diagram shows the relationship of the filesystem abstraction on storage:



2. Filesystem Operations

The following diagram shows a high level overview of how the file system drivers interact with the rest of the file system "stack". In order to support multiple filesystem types and instances Linux implements a large and complex subsystem that deals with filesystem management. This is called Virtual File System (or sometimes Virtual File Switch) and it is abbreviated with VFS.

VFS translates the complex file management related system calls to simpler operations that are implemented by the device drivers. These are some of the operations that a file system must implement:



i. Mounting a filesystem

A summary of a typical implementation is presented below:

- Input: a storage device (partition)
- Output: dentry pointing to the root directory
- Steps:
  - ✓ check device, determine filesystem parameters, locate the root inode

- ✓ Example: check magic, determine block size, read the root inode and create dentry

ii. Opening a file

A summary of a typical implementation is presented below:

- Input: path
- Output: file descriptor
- Steps:
  - ✓ Determine the filesystem type
  - ✓ For each name in the path: lookup parent dentry, load inode, load data, find dentry
  - ✓ Create a new *file* that points to the last *dentry*
  - ✓ Find a free entry in the file descriptor table and set it to *file*

iii. Querying file attributes

A summary of a typical implementation is presented below:

- Input: path
- Output: file attributes
- Steps:
  - ✓ Access *file->dentry->inode*
  - ✓ Read file attributes from the *inode*

iv. Reading data from a file

A summary of a typical implementation is presented below:

- Input: file descriptor, offset, length
- Output: data
- Steps:
  - ✓ Access *file->dentry->inode*
  - ✓ Determine data blocks
  - ✓ Copy data blocks to memory

v. Writing data to a file

A summary of a typical implementation is presented below:

- Input: file descriptor, offset, length, data
- Output:
- Steps:
  - ✓ Allocate one or more data blocks
  - ✓ Add the allocated blocks to the inode and update file size
  - ✓ Copy data from userspace to internal buffers and write them to storage

vi. Closing a file

A summary of a typical implementation is presented below:

- Input: file descriptor
- Output:
- Steps:
  - ✓ set the file descriptor entry to NULL
  - ✓ Decrement file reference counter
  - ✓ When the counter reaches 0 free *file*

vii. Creating a file

A summary of a typical implementation is presented below:

- Input: path
- Output:
- Steps:
  - ✓ Determine the inode directory
  - ✓ Read data blocks and find space for a new dentry
  - ✓ Write back the modified inode directory data blocks

viii. Deleting a file

A summary of a typical implementation is presented below:

- Input: path
- Output:
- Steps:
  - ✓ determine the parent inode
  - ✓ read parent inode data blocks
  - ✓ find and erase the dentry (check for links)
  - ✓ when last file is closed: deallocate data and inode blocks

Directories

Directories are special files which contain one or more dentries.

3. Linux Virtual File System

Although the main purpose for the original introduction of VFS in UNIX kernels was to support multiple filesystem types and instances, a side effect was that it simplified fileystem device driver development since command parts are now implement in the VFS. Almost all of the caching and buffer management is dealt with VFS, leaving just efficient data storage management to the filesystem device driver.

In order to deal with multiple filesystem types, VFS introduced the common filesystem abstractions previously presented. Note that the filesystem driver can also use its own particular fileystem abstractions in memory (e.g. ext4 inode or dentry) and that there might be a different abstraction on storage as well. Thus we may end up with three slightly different filesystem abstractions: one for VFS - always in memory, and two for a particular filesystem - one in memory used by the filesystem driver, and one on storage.