# Lecture Notes - File system, VFS API, Organization

Debadatta Mishra
Indian Institute of Technology Kanpur

## File system interfacing

- Most OSs provide file abstraction to the user processes to persist data across the life time of processes. Files also provide a method of sharing data across user processes. Standard input/output libraries are built on top of the OS system call API for file operations. For example, standard input/output library methods like fopen( ), fread( ), fwrite( ) and fclose( ) internally use system calls like open( ), read( ), write( ) and close( ), respectively.
- In the UNIX OS, file abstraction is not limited to just files but extend to devices, sockets, pipes etc., and, even used to represent in-memory objects maintained by the OS (e.g., procfs and sysfs of Linux kernel). The user view of the file system is a big-fat hierarchical tree (referred to as file system tree in this document) where internal nodes of the tree represent directories and the leaves represent files. Any user process can refer to a file through a *path* from the root (can be relative or absolute). More often than not, the file system tree contains files stored across multiple storage devices, sometimes even the actual storage crosses the physical machine boundary (e.g., NFS mount points). Further, not all file descriptors used by a user process have a representation in the file system tree. For example, TCP sockets, pipes etc. does not correspond to any path in the file system tree.
- One of the vital design challenges is to bridge the gap between the user's view of the file system tree which is encapsulated in the form of file objects like files, directories etc. with various system calls to perform operations on the file objects, and, at the same time, efficiently deal with the storage hardwares like hard disk drives (HDD), solid state devices (SSD) etc. To understand the differentiation in the above two aspects (user interfacing and storage organization), consider the example of a user executing 'ls -ltr /home/user1/code.c'. The shell must find out the location (path in the file system tree) of *ls* by trying out all the paths mentioned in the PATH environment variable. System calls like *stat* can be used to determine the existence of a file and its execute permissions. Considering the possibility of the *ls* executable residing in any persistent medium, all the file systems implementing a lookup mechanism is inflexible. Further, the path to *ls* can cross storage boundaries which implies that no single storage management entity (i.e., file system) can resolve the path in a self sufficient manner. One more aspect of file system interfaces is to match the access permissions of a process with all the elements in the path which requires understanding the OS semantics of process and file level ownership and permissions. Therefore, most of the UNIX variant OSs, divide the responsibilities into two different layers as explained in the following point.
- A generic file system layer (say OS file system multiplexer) interfaces the user API (system calls) and the process abstraction which remains unchanged for different storage mediums. The responsibility of efficiently managing and accessing the storage is left to a specialized piece of software, commonly known as the file system.

The challenge is to correctly design the interface between the OS file system multiplexer and the file systems in a modular manner. This is non-trivial because, to carry out operations initiated from user requires communication across the two layers most of the times. For example, during path translation, the OS level multiplexer may require the persistent file permission information of each intermediate path element to enforce process level file access checks.

## Linux virtual file system (VFS)

- Linux virtual file system (VFS) layer bridges the gap between system call API, process abstraction and user's view of a file system tree by interfacing with the file related system call handlers and different file systems. The VFS layer enforces all file systems to adhere to standard representation of the file system objects and provide implementations for operations on different objects. The advantages of the VFS layer are portability and flexibility where a new system call may not require changing all the file systems. Moreover, design of a new file system is much easier as the file system designer does not have to worry about the system call interfaces and can focus on designing efficient file storage and retrieval mechanisms.
- Superblock VFS object is the entry point to any file system, therefore, it is mandatory for all file systems irrespective of whether the file system maintains any equivalent persistent structure or not. Some of the information maintained in the superblock are: logical device identifier, block size, file size limits, information regarding the root of the file system etc. The superblock also provides an operations interface which is required to be implemented by the file system. For example, the super block must provide interfaces to unmount the file system, allocate new file or directory object etc. Most file systems maintain an on-disk version of the superblock which is stored in the first block of the logical device (disk partition).
- Inode is the traditional representation of a file in a UNIX system. Every regular file and directory in the file system is identified with an inode object (with a unique number known as the inode number). The inode VFS object expects information like file permissions, ownership, size, access time etc. to be provided by the file system. Most file systems maintain a similar on-disk structure as the information regarding a file persists across FS mount/unmount and system restart. Two of the most critical operations on the directory inode are *lookup the child* and *create/delete/update the child* for a given child name. The child lookup operation is necessary to perform path translation while the create, delete and update operations are required for file/directory creation, deletion and rename operations, respectively.
- Another important abstraction is dentry, a short form for *directory entry* that represents a single path element in a file path and explicitly maintain the parent-child relationship between directories and files/directories. The file system may not maintain an equivalent on-disk state and maintain explicit parent-child relationship. For example, a file system can create the child dentry by searching the child name using the parent dentry provided by the VFS. Note that, dentry contains a pointer to the VFS inode object which can be used to perform any operations on the inode which is very useful during path translation, read, write etc.

- A physical disk is partitioned into several logical partitions using disk partitioning utilities like *fdisk* where the partition information (a.k.a. partition table) is stored in the boot sector. Many OSs support a generic block layer to implement logical partitions as separate devices and hide the interactions with the device driver for the hard disk. In other words, the file system can address any logical partition as a separate disk device starting at offset 0 and extending to the end of the partition. A blank logical disk is not of much use until a file system is created on the partition using utility like *mkfs.* The file system creation must be done before the file system can be used. Therefore, the *mkfs* utility creates the file system organization information starting with an on-disk superblock structure which provides an entry point to the file system, and at the minimum, provide access the root dentry and the root inode.
- In Linux VFS architecture, when a file system module registers itself with the VFS layer, it must initialize a structure (*struct file_system_type*) containing a unique file system type string (e.g., ext4, btrfs etc.) and an implementation of *mount* operation. The VFS layer maintains a list of registered file systems by maintaining a list of *file_system_type* structures. When user mounts a file system using the command 'mount -t fstype [logical device] [mount point]' which in turn results in an invocation of **mount( )** system call with equivalent parameters, the VFS layer searches the list for the matching file system type and invokes the file system specific **mount** operation. The mount implementation of the file system populates the superblock information and returns the VFS dentry of the root along with the pointer to the root inode structure.
- After file system mount, the VFS has the following information regarding the mounted file system: the superblock, mount point information, root inode. At this point, the file system is ready to self expand by implementing file and directory creation operations. For example, during Linux installation, after disk partitioning and file system creation, the installer OS (which also contains code for file system implementation) mounts the '/' partition of the target machine to a directory in the installer file system tree and creates the file layout for the installed operating system.
- Consider the example of a user process executing **open("/home/user/file1")** in the Linux system where '**/**' is the mount point. The VFS layer iterates through each path element---home, user and file1, starting with '/' as the parent and invokes dentry lookup of the next path element (home) and checks file permissions at each level. At every step, the next level dentry and the inode is requested from the file system. This process is continued till the last element in the path (i.e., file1) is resolved by accessing the inode of the file. Internally, the file system may perform required disk I/O operations using the generic block layer to satisfy the VFS requests. For example, **lookup (home_dentry, "user")** may trigger disk I/O to read the contents of **home** directory from the disk. After the iterative path resolution, the VFS layer creates a file object, registers the file operations (implemented as the inode object's file operations) and returns the file object to the system call layer. The system call layer creates a file descriptor in the PCB pointing to the file object and returns the file descriptor to the user space.
- When the user process invokes **read (fd, ...),** the system call handler invokes the VFS layer read operation using the file object corresponding to the file descriptor (fd). The VFS layer in turn invokes the file system read implementation using the VFS

inode. Note that, in this case, file descriptor to file object mapping is the responsibility of the system call handler layer where file object semantic is shared across the system call layer and the VFS layer. The VFS layer is responsible for maintaining the mapping between file objects and the inode where inode is the handshake point for the file system and the VFS layer.

- The VFS layer invokes **lookup(parent, "next level name")** multiple times to translate every path. Given that some paths are used frequently, caching the translation (a.k.a. dentry cache) can make the translation process more efficient. For example, the VFS layer may implement a key-value data structure where the key = {parent dentry, child name} and value is the child dentry. One important aspect of dentry cache maintenance is to update or purge cached elements when either the key or value changes. For example, when a file or directory is renamed, the dentry cache must be updated with the new name or the old entry must be purged. Recent advancements in dentry cache mechanism propose to supplement the individual dentry cache elements with full path caching where a full path like '**/home/user/file.c**' is used as the key to extract the dentry and inode for **file.c**. Using full path caching, not only the disk I/O overheads can be reduced, but also repeated lookup operations can be avoided.

## File system organization

### Physical block management

- One of the primary design concerns to manage storage is the efficient implementation of allocation and free operations. Most file systems manage the storage at a block granularity which is mostly of same size as the memory page size for convenient physical memory accounting and management. The biggest challenge in managing disk blocks is scalability as the disk size can be in the range of terabytes. For example, for a disk of 4TB with file system block size of 4KB, even a 4-byte metadata to maintain the per-block status information consumes 4GB storage space which can be the worst case memory (RAM) requirement as the structures should be brought into memory before accessing them.
- A simple method can be to maintain a list of free blocks in a linked list which provides constant time allocation and free operations. The challenge is to maintain the next pointers (if maintained in the block itself) which can consume a lot of space and make the block data extraction non-trivial and cumbersome. If the link is maintained in separate storage space, searching the storage space for allocation and free operation becomes a bottleneck. One more issue with the linked list approach is difficulty implementing fault recovery as one broken link can jeopardize access to the information regarding the free blocks.
- One of the commonly used techniques is to maintain a bitmap of blocks where value of the $i^{th}$ bit in the bitmap reflects the status of the $i^{th}$ block. For example, if bit 100 in the bitmap is 0, then $100^{th}$ block is free. Performance of allocation depends on the fillup factor where finding a free block in a densely used disk can take almost a full scan of the bitmap. Performance of free operation is constant time. This approach scales nicely with disk size as the metadata for block status is just 1-bit. For example, for a 4TB disk with block size 4KB, 128MB of storage is required to maintain the

bitmap which can fit in the main memory of most modern computer systems. Fault tolerance aspects are not great with block bitmaps because a corrupt block bitmap can impact the correctness of the file system. However, file systems can maintain redundant copies of the bitmap (helped by the small size of the block bitmap) in the disk.

## File offset indexing

- Logical file offset to physical block mapping is probably the most important aspect of any file system organization because of the following factors,
    - File sizes can vary from few bytes to gigabytes. An indexing method must efficiently translate the logical file offset to the physical block for all file sizes.
    - Files can be accessed either in a sequential or random manner depending on the workload behavior.
- One of the simplest approach is to statically allocate contiguous disk blocks for a file, maintain the address of the starting block in the inode along with the file size information. Both sequential and random access is a direct access to the device block without any intermediate metadata access. However, there are several issues with this approach.
    - Preferably, the file system must know the file size during the creation of the file which is impractical.
    - Contiguous allocation results in external fragmentation issues where there are enough free blocks scattered across the device but cannot serve an allocation request because they are not contiguous.
    - Append operation may require relocating the existing blocks of a file in the worst case.
- Another scheme can be to implement a linked organization of the file blocks where the first file block provides an pointer to the second block and so on. In the inode, the first and last block address can be maintained where using the first block, any file offset can be reached and the last block is used to implement efficient append operations. This method is flexible as the file can grow and shrink without any issues. However, random access to a file offset is very bad as many blocks are to be navigated from the start block to reach the file offset resulting in a lot of disk I/O operations.
- A simple technique to manage small files without static contiguous allocation or linked allocation is to maintain a list of block pointers from the inode itself. The inode contains block addresses in an array where the $i^{th}$ elements in the array contains the block address corresponding to the file offset *BlockSize* * *i*. Using this approach, any file offset can be accessed without any other intermediate metadata access. However, the limitation on file size because of limited size of inodes is an unrealistic assumption. Implementing dynamically sized inodes to support variable number of block pointers can address this problem to an extent at the expense of complicated inode organization structures in the file system.
- Indirect block pointers are similar to hierarchical paging structures used to translate virtual address to physical address. In this case, the inode contains a pointer to a block which contains the actual translation information. Single indirect pointers can support file sizes larger than the direct pointer based organization, but still impose a limit on the maximum file size. Extending the indirect pointers to multiple levels of

indirection i.e., double indirection, triple indirection etc., very large files can be supported without expanding the inode. Indirect organization provides flexibility in terms of file expansion. However, additional indirect block accesses are required depending on the levels of indirection to determine the file offset to device block mapping.

- Many file systems implement mixed indexing mode--- a combination of direct, single indirect, double indirect and triple indirect pointers. For example, ext2 file system uses fifteen (15) block pointer addresses in the inode where twelve (12) block pointers are direct block pointers, one single indirect block pointer, one double indirect block pointer and one triple indirect block pointer. Considering the block address to be 4-bytes, the maximum file size supported by this system can be approximately 4 terabytes (TB). Note that, to translate random file offset, three intermediate metadata access is required in the worst case. The hybrid organization favors the small files as they can be accessed without any intermediate metadata access. Files of size more than 48KB require indirect block accesses which is not a very good design. In the modern high capacity disks, 4KB block size acts as an impeding factor resulting in many indirection, especially in a case when contiguous free blocks are available. Ext4 file system extends the hybrid design to realize variable block sized organization in an indirect manner.
- The ext4 extent structures use the fifteen block pointers of the ext4 disk inode structure to store pointers to the data blocks. The extent organization is a tree like structure where each level of the tree contains a special header known as the extent header which contains the information regarding the number of entries and the type of entries in that level. The entry type can either be extents or pointers to next level header. An extent entry contains the start block and length pair allowing variable length file chunks. For example, if the extent tree depth is zero (implying direct extents), the extent header contains information regarding number of extent entries where each extent entry point to a chunk of blocks. For extent tree with height more than 0, the extent header points to indirect extent pointers which in turn points to the real extents. The advantages of using extents are---faster access (both random and sequential), better spatial locality in the storage device and scalability. However, the benefits of the extents can be undermined if the disk is almost full and/or highly fragmented.

## Inode and directory organization

- Every file system maintains file and directory information in a persistent manner such that the information is not lost across system reboots. Moreover, VFS compliant file systems have to provide an implementation of an abstract inode. Therefore, inode organization on the disk play a crucial role in file system performance. Many file systems access the inode using a unique number known as the inode number. In such systems, deriving the inode location on the disk from the inode number is crucial. Most modern file systems implement fixed size inodes as opposed to variable sized inodes for simplicity.
- Just like block bitmap, one way of organizing the inodes is to statically reserve the space for inodes at some block offset and implement an inode bitmap structure along with the starting address of the inode blocks. Inode bitmap is accessed when a file is created or deleted. Maintaining a single inode bitmap for the whole disk can make

the inode block and the data locations of a single file separated by a large gap which require long seek times in a magnetic disk. One approach to address this issue is to divide the disk device into groups based on their physical layout (e.g., based on the cylinder address). Other advanced dynamic inode allocation and management schemes (used in btrfs and XFS) implement balanced search tree (e.g., B-tree) data structures.

- Directory is also represented by an inode, contains data like files but the data contained in a directory can be structured by the file system. The content of the directory is the information regarding the children inside the directory which can be files, directories or links to other files. Operations on the directory include search for a file/directory with a name (VFS path translation), create/delete a file/directory and file rename operations inside the directory.
- File system can maintain directory entries using data structures like array of fixed length entries, linked list of variable length entries, hash maps, search trees etc. One of the simplest implementation (used in ext4) is to maintain a list of variable length entries where each entry is defined as follows,

```
struct dir_entry{
      inode_t inode_num;
      u16 entry_len;
      u8 name_len;
     char name[name_len];
};
```

- To locate a file/directory inside a parent directory, a linear search of all entries is required which can be inefficient. Further, before creating a new entry, all existing entries must be checked to ensure no duplication of names in any given directory. To remove an entry the entry must be located and removed resulting in holes which may be compacted in a latter stage. Advanced file systems use data structures like hash maps and balanced search trees to improve the search efficiency.
- Ext4 file system partitions the storage space into block groups and maintains the information regarding the block groups in the group descriptor table. The base address of the group descriptor table is stored in the superblock. Each block group contains the following information,
    - A copy of the superblock, group descriptor table. This is useful to enhance the reliability of the file system in cases of disk sectors becoming unusable.
    - The block bitmap and inode bitmap structures represent the information regarding inode and block usage in the given block group.
    - Inode table is the statically preallocated space for the inodes in the group. To map the file system inode number to the group level inode offset or vice-a-versa, superblock maintains the information regarding number of inodes per group.
- Let us consider the operations required to open the file '/etc/hosts' in a ext4 like file system assuming no VFS layer dentry caches. First, the VFS layer extracts the mount information of '/' to determine the root dentry and root inode structures. The VFS invokes lookup of 'etc' in the root inode, the file system reads the contents of the root directory (using the inode mapping table/extents) and searches for the directory entry with name 'etc'. The dentry for 'etc' is created after reading the inode of 'etc'

(found from the FS directory entry as shown in the above structure) and the VFS dentry of ‘etc’ is returned to the VFS layer. In the next step, VFS layer checks the access permissions and ownership using the inode for ‘etc’ before invoking another lookup for ‘hosts’.