File Systems

Files and Directories

A file is simply a linear array of bytes, each of which you can read or write to. Each file has a low level name, called an inode number.

A directory contains a list of <user-readable-name, low-level-name> pairs, describing the files in the directory.

An arbitrary directory tree or directory hierarchy is built by placing directories inside each other. It starts at the root directory (usually /) and uses some kind of separator to name subsequent sub-directories until the desired file or directory is named.

A file can be referred to by its absolute pathname starting with / or its relative path name starting with ./ .

The Filesystem Interface

1. File creation

The open() system call is used to create files, by passing the O_CREAT flag. O_WRONLY specifies only write operations on the file and O_TRUNC truncates the file to zero bytes, removing existing content.

open() returns a file descriptor, which is just an integer, private per process, and is used to access files. It is also treated as a pointer to an object of type file.

2. Reading and Writing Files

The strace tool in Linux allows to trace every system call made by a calling program. The open() system call is used to read files, using the O_RDONLY flag. The first argument is the file descriptor to read to, the second is a buffer where the result of the read will be placed, and the third is the size of the buffer.

To write to a file, the write() call is used.

3. Non-sequential Read-Writes

The lseek() function is used to seek to some offset to start read or write from that offset. The first argument is a fd, the second is the offset, and the third is whence, which determines how the seek is performed. It can take three values: SEEK_SET, SEEK_CUR and SEEK_END.

lseek() does not perform a disk seek, it just changes a variable in the OS that tracks at which offset the next read or write starts.

Writing immediately with fsync()

Most times when programs call write(), the filesystem buffers the write to optimize writes done to the filesystem as a batch operation. However, this write is sometimes needed to be instantaneous, so the fsync(int fd) function is used, by forcing all the dirty bit data to be written to disk for the fd file descriptor.

fsync() on the directory may also be required to preserve file system integrity after the force write on a file.

Renaming Files

Renaming uses the rename() system call, which takes the old name and the new name. It is atomic w.r.t system crashes, which guarantees no inconsistencies.

Getting Information About Files

File systems store metadata, which keep information about the files in the file system. The stat() or fstat() system calls are used to view this metadata. This information is placed in a structure called an inode.

Removing Files

The unlink() system call takes the name of the file to be removed and removes it.

Directory APIs

The mkdir() system call allows the creation of directories in the filesystem. An empty directory is created, which has on entry to refer to itself and another that refers to its parent.

The ls call used to read directories uses the opendir(), readdir() and closedir() system calls, which prints the name and the inode number of each file in the directory.

The rmdir() is used to delete directories, provided the directory passed is empty.

Hard links

The link() system call takes an old pathname and a new one, and creates a link from the new pathname to the same file. The ln command is used to this. This refers to the same inode number of the file. This is why unlink is used to remove the file, to remove this hard link. This only decrements the reference count of the file. Only when this reference count is 0, the file is truly lost.

Symbolic Links

A symbolic link is used as hard links are limited in usage. For example, creating a hard link to a directory may create a cycle in the directory tree, and creating a hard link to files in other disk partitions is not allowed as inode numbers are unique only within a particular file system, not across file systems, etc.

Symbolic links links an actual file, but of a different type, which hold the pathname to a file. This causes the possibility of dangling references, when the upstream file is deleted.

Making and Mounting a Filesystem

Mkfs is used to create a file system, and the mount() system call is used to mount the new file system. mount() takes an existing directory as a target mount point, and pastes a new file system onto that directory tree at that point.

The Very Fast File System (VSFS)

The two key components of a filesystem are

1. Data Structures
2. Access Methods

Overall Organisation

First the disk is divided up into blocks, with a fixed size. The blocks are addressed from 0 to N-1. The region of the disk for user data is the user region, and the last section of the disk is allocated for this.

For metadata storage, we define an inode structure. To accommodate these inodes, we reserve some space on the disk, called the inode table, which holds an array of on-disk inodes.

To track inode information, we use allocation structures. Allocation techniques include having a free list that points to the first free block, and so on. Instead, we use a bitmap, one for the data region, called the data bitmap, and one for the inode table, called the inode bitmap.

Each bit in the bitmap indicates whether a block is free (0) or in use (1).

A superblock is added to contain information about the filesystem, including number of inodes, where the inode table begins, and so on. The OS reads this superblock to mount the filesystem correctly.

The Inode

Inode is short for index node. These are referenced by an inumber, which is the low level name of the file. Given an inumber, the corresponding inode is located. To read an inode, the fs firsts calculates the offset into the inode region which is the inumber*sizeof(inode), and add it to the start address of the inode table on disk. The sector to read from is calculated as

Blk = (inumber * sizeof(inode))/blockSize;

Sector = ((blk*blockSize) + inodeStartAddr)/sectorSize;

An inode stores

1. File type
2. Size
3. Number of blocks
4. Protection information
5. Time information
6. Pointers to the data blocks

The pointers can be direct or indirect, where direct refers to one-to-one pointers whereas indirect pointers point to a bunch of indirect or direct pointers.

To support bigger files, indirect pointers are used, as the number of direct pointers are limited.

Another different approach is to use extents instead of pointers, where extents are disk pointers + a length, which defines where the file is on disk. They allow for more than one extent if the files are not contiguously stored.

Pointer based approaches are more flexible, but store more metadata per file, and the extents are the opposite.

If single indirects are not sufficient, multilevel indexing can be done.

Another approach is to use a linked list for inodes, where each inode points to the first block of the file, and each data block has a pointer to the next data block of that file. This performs poorly when random access is required. So a table of link information can be stored, and this structure is the FAT system or the file allocation table file system.

Directories also gave inode numbers stored in the inode table, with the type field marked as ‘directory’. This structure is however, rarely used, and the directories are implemented using B-trees.

Free Space management

In vsfs, the two bitmaps manage free space efficiently.

Free lists can be used where a single pointer points to the first free block, and inside that block was the pointer to the next free block. Modern FS uses B trees.

In the bitmap version, the filesystem searches through the bitmap for inodes that are free and allocate it to requesting files, and update the bitmaps. Pre-allocation is also done when file systems try and allocate contiguous blocks to files.

Access Paths : Reading and Writing

1. Reading
    a. Issue open() or read() command
    b. Find the inode for the file and obtain file information by traversing through the directory tree
    c. Once the inode is read, FS looks inside it to find pointers to data blocks
    d. Use these data blocks to read through the directory
    e. Do a final permissions check, allocate file descriptor and return
    f. A read does not access allocation structures.
2. Writing
    a. First open file
    b. Writing to a file may allocate a block.
    c. 3 I/Os
        i. One to read the data bitmap to mark the newly allocated block
        ii. one to write the bitmap
        iii. one to write to the block.
3. Creation
    a. 6 I/Os
        i. Read inode bitmap for free inode
        ii. Write to inode bitmap to mark it allocated
        iii. One to write to the new inode
        iv. One to the data in the directory
        v. One read to directory inode
        vi. One write to update directory inode

Caching and Buffering

Caching and buffering reduces the number of I/Os required to read and write files. Most systems cache important blocks in the DRAM.

Earlier, fixed size blocks were used with LRU replacement. Now, unified page caches are used to integrate virtual memory pages and file system pages.

For writes, write buffering is done by

1. Delaying writes : Batch updates into a single I/O
2. Schedule subsequent I/Os, increase performance
3. Writes avoided by delays, if a file is created and deleted

There is a tradeoff of a potential of losing information when the system crashes. So this is countered by using fsync().

Drawbacks of the VSFS

Terrible performance is the main issue. This is because

1. The disk was treated like random access memory, with data spread all over the disk without regard for the medium, which had expensive operations
2. Filesystem could easily get fragmented as free space was not carefully managed
3. Block size is too small, making disk transfers very inefficient

The Fast File System

The fast file system is a disk aware filesystem, which kept the same interfaces for system calls, but changed the internal implementation.

Organizing Structure : Cylinder Groups

FFS divides the disk into a bunch of cylinder groups, and each group can contain two files, where accessing these one after the other does not result in long seek times. Each cylinder group has the same structure as the VSFS with the superblock, bitmaps, inodes and data.

Allocating Files and Directories

FFS keeps related files together.

For directories,

1. Find the cylinder group with a low number of allocated directories and a high number of free inodes
2. Put directory data and inode in that group

For files,

1. Make sure to allocate data blocks of the same file in the same group as its inode, for faster seeks
2. Place all files that are in the same directory in the cylinder group of the directory they're in.

Measuring File Locality

SEER traces are used to analyse how far away file accesses are from one another in the directory tree, as a heuristic. It is basically finding the least common ancestor of two files.

Large Files

Filling a block group with one file is undesirable, as it goes against keeping related files together.

For this,

1. Some blocks are allocated to the first block group
2. Place the next large chunk in another block group recursively for each chunk

This hurts performance, but choosing the chunk size carefully reduces the overhead by doing more work per overhead paid, which is termed as amortization.

This was made simple by putting all the blocks pointed by an indirect pointer in a different group and all the direct pointer blocks in the same group.

Other Innovations

1. Sub-blocks
    a. Reduces internal fragmentation
    b. 512 byte blocks to allocate to files
    c. Lot of overhead, so uses libc to buffer writes and issue them in 4KB chunks
2. Disk layout
    a. Sequential reads problem, when blocks kept rotating faster than the read
    b. Skips alternate blocks, called parameterization, to avoid extra rotations
    c. Track buffers used to read the entire track for faster access
3. Allows long filenames
4. Symbolic links
5. Usability improvements