3. File Systems

We will look at file systems (FS) from two viewpoints:

- Static view: what's the layout of a file system?
- Dynamic view: what operations does a file system support?

An OS can support more than one FS.

A FS can be accessed by more than one OS.

Files

Why do we need files?

Storing information in memory is good since memory is fast, but memory vanishes after reboot.

Files provide long-term information storage. They are persistent.

Files can also be shared objects for processes to access concurrently.

Filenames: When a file is created, a name must be given.

Pathname vs. Filename: the pathname is unique across the entire file system; the filename is not unique, but unique within the directory.

File attributes: important to the file system. FS-dependent, not OS-dependent. stat() system call can read file attributes, or check whether a file exists or not without opening that file.

File types

- Regular file: inspect text files using cat; binary files using xxd. The command file can distinguish the type of a file, without checking the extension of the filename.
- Directory file
- Link file
- Device file

Distinguishing binary files: magic number

Directories: A directory is also a file. A directory file is an array of directory entries. It records all the files and directories that belongs to it.

Links: A link is an alias of a file.

- In Windows, we have the shortcut.
- In Unix/Linux and macOS, we have the hard link and symbolic link.

Both the shortcut and the symbolic link share the same concept.

- A link file is created and stores the pathname of another file.
- To create a symbolic link: ln -s filename linkname

We will come back to the hard link when we discuss the file system implementation.

Device files: Unix/Linux creates some pseudo-devices for convenience.

- /dev/zero: output an endless sequences of zeros.
- /dev/urandom: output an endless sequence of random bytes.
- /dev/null (a.k.a. the black hole): whatever data written to this device will be discarded.

File creation = update of the directory file

File deletion: Removing a file is the reverse of the creation process.

File system design

A file system is about...

- How the OS stores and locates a file.
- How the OS stores and locates a directory.

It's also about how the OS manages the storage in an efficient and reliable way.

- Efficiency is about the speed and the storage utilization.
- Reliability is about "can we always get back what have been saved?"

File system structure:

- Application level (user)
- Logical representation level (partially user and mostly kernel)
- File organization level (kernel)
- Device control level (kernel)

File organization: goal is to store files on the disk, read and write the stored files in an efficient and reliable way

linked list to store the next block at each block $\rightarrow$ store all the linked lists as a table

File allocation table (FAT) approach

- All the information about the next block are centralized as a table.
- The entries in the table are stored contiguously as an array.
- Each entry corresponds to one file allocation block in the storage device.

Drawback: the file allocation table takes a lot of space (thousands of blocks).

In FAT, a block is called a cluster. Different versions of FAT support different number of clusters.

L = length of the cluster's address, S = size of a cluster, then maximum file size = $2^L \cdot S$.

Index node (inode) approach

- The file system table maintains the mapping from the filenames to the index nodes (inodes).
- An inode stores the addresses of all the data blocks.
- All the inodes are stored in the inode table.

Each inode has a fixed size. At the beginning, the addresses of the data block are stored in the direct block addresses of the inode. After they have been used up, the indirect block addresses will be used, which point to other data blocks. There can be indirect blocks of indirect blocks.

Summary: block-based addressing, one large table (FAT) → many small tables (Inode)

Free-space management: keep track of every free block in the file system

A bitmap of block allocation: a series of boolean values

FAT32: hints of first free cluster & number of free clusters; little space needed but inaccurate & slow

Support directories: a directory is just a list of directory entries

File attributes: stored inside a directory entry (FAT16, FAT32) / inside an inode (ext2, ext3, ext4)

File system information

Boot sector (FAT, NTFS): the first 512 bytes of the FS to store all the FS-specific data

Superblock (ext2, ext3, ext4): a 1024-byte region to store all the FS-specific data

Disk partition: a logical space to host a file system

A smaller file system is more efficient

“formatting” a disk — creating and initializing a file system

FAT32 in action

Reading a directory

Step 1: locate the root directory

Step 2: find the directory entry

Step 3: follow cluster number, read and return all directory entries

Step 4: read and return the entire directory entry structure

Directory entry (32 bytes)

Bytes 0-10: 8+3 characters of filename + extension

Bytes 20-21: high 2 bytes of the first cluster address

Bytes 26-27: low 2 bytes of the first cluster address

Bytes 28-31: file size

Reading a file

Step 1: read the first cluster from the directory entry

Step 2: look up the next cluster in the FAT and read that cluster

Step 3: repeat the process until an EOF entry is found in the FAT

Writing a file

Step 1: for appending, locate the cluster of the end of the file

Step 2: look up the FSINFO structure to find the next free cluster

Step 3: allocate new cluster by changing the FATs and FSINFO

Deleting a file

Step 1: find the directory entry and the locations of the clusters

Step 2: set all the next address fields in the FATs of that file to 0

Step 3: update the FSINFO structure

Step 4: change the first character of the filename to 0xE5

The file is not really deleted! Perform a search in all the free space to find all deleted file contents. Those data persists until the deallocated clusters are reused. This is a trade-off between performance (during deletion) and security. There are ways to delete a file securely.

How to recover a deleted file?

Pull the power plog immediately! It prevents the target clusters from being overwritten.

If the file size is no more than one cluster, the recovery can be easily done.

- The first cluster address is still searchable.
- Note that files with the same suffix may also be found.

If the file size is larger than one cluster, other clusters' addresses are all gone...

- However, due to next-available search, clusters of a file are likely to be contiguously allocated.
- If not, we'd better have the exact file size and the checksum of the deleted file beforehand so that we can use a brute-force method to recover the file.

ext2/3/4 in action

The primary file system on Linux is ext4 (fourth extended filesystem).

default block size = 4KB (configurable from 1KB to 64KB), default block address = 32-bit

The file system is partitioned into block groups.

- Each block group has the same internal structure.
- The group descriptor table (GDT) stores important information.

Why doing so?

- For performance and reliability.
- To keep the metadata and the file contents close together. Thus, the disk

head does not need to travel a long distance.

- The metadata is scattered, so there is no single point of failure.

The metadata structure of a block group

Block bitmap stores the block allocation status of the same block group.

Inode bitmap stores the inode allocation status of the same block group.

Inode table stores the contents of all the inodes of the same block group.

- Each inode has a fixed size specified in the superblock.
- The inodes are stored sequentially.

The numbering of inodes starts from 1. The first 10 inodes are reserved for special purposes.

Inode #1 — bad blocks; Inode #2 — root directory; Inode #8 — journal data blocks.

Link files

Hard links

A hard link is a directory entry pointing to an existing file.

- No new file content is created!

Conceptually, this creates a file with two filenames.

- Deleting only one of the directory entries will not delete the file content!

The link count field in the inode keeps track of how many directory entries are pointing to this file.

- unlink() system call to remove a link & link count -= 1; remove file content when link count = 0

Special hard links

- The directory "." is a hard link to itself.
- The directory ".." is a hard link to its parent directory.

When a regular file is created, the link count is always 1.

When a directory is created, the link count is always 2.

Removing a directory is as simple as removing the directory entry and decrementing its link count.

- The rmdir() system call is used

Symbolic links

A symbolic link is a file.

- Unlike a hard link, a new inode is created for a symbolic link.

Where is the target path stored? It depends on the length of the path.

- If the path is fewer than 60 characters, it is stored in the 12 direct block and the 3 indirect block pointers.
  - $(12 + 3) \times 4 = 60$
- If the path is more than 60 characters, one data block is allocated to store the path.

Writing a file

Step 1: Use a linear search in the inode bitmap to find an unallocated inode for the new file.

Step 2: Use a linear search in the block bitmap to find unallocated data blocks for the new file.

Step 3: Read the inode of the root directory, i.e., inode #2.

Step 4: Following the data block pointers, read the directory entry structure for "/dir1"

Step 5: Read inode #123 and the data blocks. Construct the directory entries.

Step 6: Add a new directory entry with inode #1024 to "/dir1 ".

Deleting a file

Step 1: Read the inode and the data blocks of "/dir1 " and locate the inode of "picture.jpg".

Step 2: Read the inode #894 and decrement the link count of "picture.jpg ". Now, the link count has become zero.

Step 3: Deallocate the data block and the inode by setting the corresponding bits in the block bitmap and the inode bitmap.

Step 4: Change the entry length of the previous directory entry of "picture.jpg".

The change in the entry length aims to skip the deleted entry. Such a change produces holes among the directory entries. Again, as the same happened in FAT32, the file is actually not deleted!

Linux file system internals

Logic representation level

It's about how the kernel models a file in a logical way.

- How does the kernel interact with the disk?
- How does the kernel interact with the user?

Kernel-process relationship

— a strong relationship between a process and the opened files.

The user program is given a file descriptor returned from the open() system call as an abstract representation of an opened file. It's so abstract that it's just a number!

All the opened files are stored inside a structure of type struct files_struct.

- By default, every process initially has three opened files.
- File descriptors 0 for stdin stream, 1 for stdout stream, 2 for stderr stream

When the process calls the open() system call to open a file...

- The file system module reads the metadata of the file from the disk.
- The opened files will be represented as a structure and added to the process.
- A new file descriptor will be allocated for the new file.

The structure created contains the information about the directory entry. In turn, the directory entry contains the name and the inode of that file.

The scope of a file descriptor is restricted inside the process. Different processes may use different file descriptors to refer to the same file.

Virtual file systems

For each opened file, a set of file system functions is associated with it.

- They are just function pointers. They are called VFS functions. They are FS-specific.

The VFS functions are invoked by the FS-related system calls.

Pros

VFS allows the OS to support multiple file systems at the same time.

- You know, the open() for ext4 is a lot different from the open() for FAT32.

It's easy to support a new file system in the future.

- All you need to do is implement the FS functions based on the VFS structure.

Cons

Cannot utilize special functions of a specific file system.

File system performance issues

Data fragmentation

It's about the allocated blocks of a file. A problem arises when the blocks of a file is randomly scattered on a disk.

Some basics about the hard disk drive: A hard disk drive has a movable disk arm, which has a disk head for reading and writing data. Data are stored sector-by-sector on the platters using magnetism. The platters are spinning. Therefore, a spinning of the platters can retrieve a set of continuous sectors.

If the data blocks are scattered, the number of disk head movements will be large. The number of disk head movements determines the time in accessing the file.

We need defragmentation.

Cache

To save disk access time, we want to keep frequently accessed files in the memory at all times. That piece of memory is called a cache.

Cache replacement algorithm

File system consistency and recovery

System crashed; detect & recover the inconsistencies after the operating system comes back

Error #1: Free space is allocated to nothing!

fsck can construct a bitmap of allocated blocks using the inode table, thus discover missing blocks.

Reset the free space bitmap; reset field(s) in the superblock.

Error #2: Allocated space are chained together now. But the space is not associated to any files.

fsck can discover missing blocks, and dangling inodes, i.e., inodes created without any directory entries referring to

Reset the free space bitmap; reset field(s) in the superblock; delete the dangling inodes.

Error #3: The directory entry is pointing to uninitialized data! The content of the file is wrong!

A journal is the log book for the file system. It's kept inside the file system,

A set of file system operations becomes an atomic transaction. A transaction marks all the changes that will be done to the file system. Every transaction is written to the journal.

After a crash, the recovery of file system is to replay the uncommitted transactions on the journal.

What if a crash happens while the system is writing the journal? During recovery, the transaction is found to be incomplete, and it is not able to be completed.

RAID

Idea: use redundancy to improve both performance and reliability.

- Use redundant array of inexpensive disks as one storage unit.
- Fast: simultaneously read and write disks in the array.
- Reliable: use parity (i.e., XOR) to detect and correct errors.