Lecture Note 7

Gianmaria Rovelli

EPFL

1 Storage Device

1.1 Memory Hierarchy and storage devices

1.2 Hard-Disk Drives

Your computer could have astonishing components (GPU, CPU, RAM), but a poor storage device can still represent a huge performance bottleneck.

Disks are slow since they are made of mechanical parts. To seek data, the head may move the head, therefore losing time, and causing performance degradation.

2 File System

2.1 How to organize data stored on storage devices

The storage is split into blocks (fixed-sized chunks). In the case of a disk, the blocks must be mapped to cylinder, header, and sector to find the correct data associated with each block.

For instance, 1 block represents 8 sectors and occupies 8 * 512 bytes = 4KB.

2.2 File system abstraction

Simply storing data is not enough. A system to associate data with its metadata is needed. In particular, we want to locate and store files.

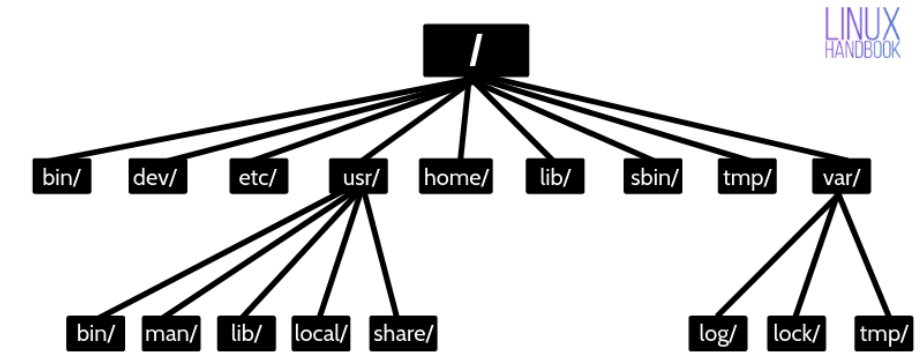
A file is a named collection of related information (blocks) stored in the secondary storage (SSD, HDD). It is composed of two parts: data (bytes), metadata (information about the data, such as size, owner, creation date, etc.)

There exists a special type of file, the directory. Directories are used to group and names files in human-friendly manner.

Moreover, we have links. Links are file pointers, they do not contain data, but they are references to other files. There are two types of links: Hard links and Symbolic (soft) links. Hard links are a literal mapping of the original file (same inode). Symbolic links are a new file that points to another one (new inode). If you have two hard links and remove one, you do not lose the data. If you have the file and a symbolic link, if you remove the file you lose the data.

2.3 File system implementation

The file system manages the data for the users. It has to avoid fragmentation, reduce overhead, and implement accessible API (read, open, write, etc.)



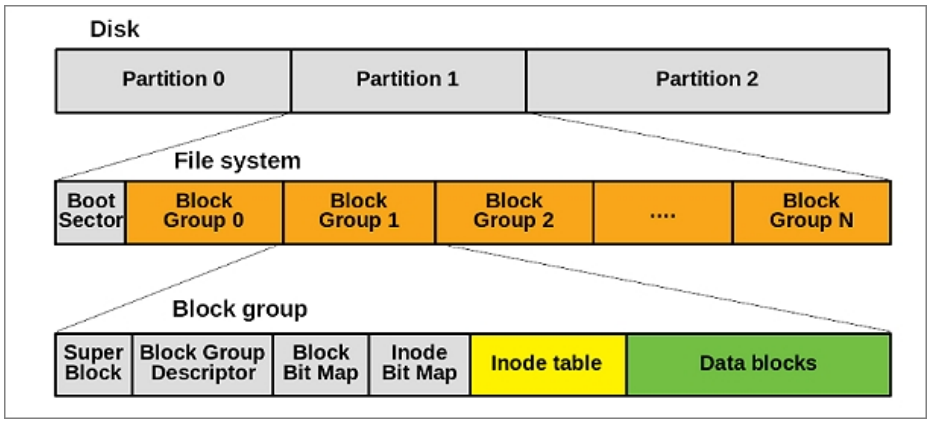


Fig. 1. Ext2 file system structure

2.4 File system layout

The file system is stored on disks (it has to be persistent). The disks can be divided into multiple partitions.

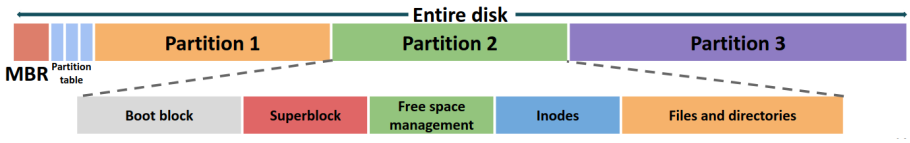
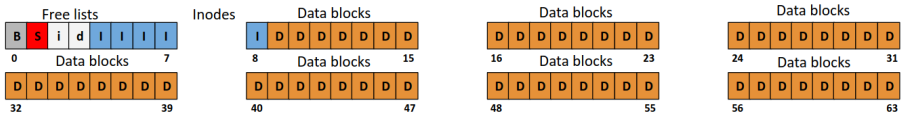


Fig. 2. File system layout

We have multiple boot blocks (one for partition) to support various OSes.

2.5 Peeking inside a partition

The file system must be able to distinguish among data, metadata, and free memory in the storage devices. To achieve such distinction, the file system uses an array of inodes, a bitmap to track free inodes and data blocks.



2.6 Consistency problems

If a crash or power outage occurs between writes, we could have an inconsistent view of the data.

Common problems are : - data is written, but not accessible - wasted memory, data bitmap says block is free, while inode says it is used - data bitmap says if data blocks area used, but no inode points to it

The solution to those problems is Journaling

3 Log-structured file system (LFS)

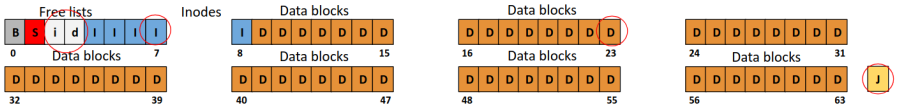
3.1 Journaling

The goal is to limit the amount of required work after a crash (recover files) and maintain a correct state, not only consistent.

The idea is to "write ahead" a short node to "log". So that every time the FS does something, it first logs and then acts. In case of a crash, consulting the log is enough to determine the correct course of action.

The logs reside in the journaling partition -> could be bad for temporal locality.

Journaling uses transactions' atomicity to provide crash consistency.



4 LFS

The idea is to instead of adding a log to the existing disk, use the entire disk as a log.

This should help with temporal locality. LFS buffers all updates into an in-memory segment, when such a segment is full it is written to the disk.

So for every modification of a file, you have a new "version" of it, pushed somewhere in the memory (LFS could be used for versioning indeed). Now we have a problem, the inodes are scattered throughout the disk. No organization, no logical partitioning, and no fixed patterns. Moreover, each inode has different versions.

To keep track of the latest versions, LFS uses an inode map (imap). It takes an index number as input and outputs the disk address of the most recent version of the inode. The imap is written to the disk as any other data.

Therefore to maintain consistent pointers to imap, LFS has a fixed region (Checkpoint Region) for imap lookup. The idea is to update periodically the mapping between imap and Checkpoint Region.

4.1 Read data

To read a file from the disk: - read from the checkpoint region to get imaps - read the most recent inode from the imap (using the version) - read a block from the file with pointer in the imap

Directories are handled similarly to a file, simply their content are file names and their inode numbers as pairs.

4.2 Downside

Downside of LFS: it continues to write into the memory new versions of the files, leaving old versions all over the places (literally garbage). So, it is necessary a way to identify and remove such garbage.

4.3 Garbage Collection

LFS uses garbage collection to remove the garbage accumulated over time. It is performed in a segment-by-segment basis and periodically. Only the latest versions of the files are kept.

4.4 Extensions

LFS can possibly be used to support snapshot and versioning. Keeping some of the old versions, and reuse them when necessary.