

Disk organization

Lecture 12

Filesystem implementation

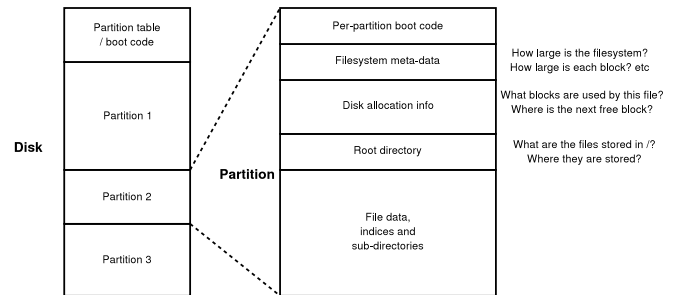
The raw disk is addressed like memory: using integer addresses. For files to be stored into and retrieved from the disk, the disk is divided into **partitions**, each of them is used to hold a **filesystem (FS)**. In this lecture we will see how the FS is organized.

References:

- [OSC] 12.1–12.8 (File-System Implementation), 14.1–14.2 (Disk Scheduling).
- [ULK] pp. 495–505 (The Ext2 Filesystem).

POS(0230A)

Conceptually, the disk is organized like this:



Actual organization depends on the FS used by the partition. E.g., some have two FAT, some repeat the whole structure many times, etc.

POS(0230A)-12.1

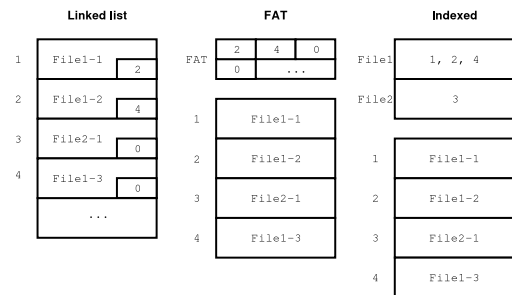
Locating disk blocks for each file

At its core, a FS is used to find **disk blocks** for **files**. Some popular on-disk data structure to allow this to be done efficiently:

- **Linked list:** the files are organized as a *linked list of blocks*; the last few bytes of the block contains the address of the next block.
We need to know the disk address of the first block by some other means.
- **FAT (File allocation table):** like linked list, but all the *links* are grouped and stored in a few blocks called the *file allocation table*.
- **Indexed:** Each file has an *index node (inodes)* that contains the disk addresses of its blocks. Index depth determines the maximum file size.
We need to know the location of the inode by some other means.
- **Variable level indexing:** Like indexed structure, but each inode contains a few indices of different depths.

POS(0230A)-12.2

How they look like?



FAT improvement: one can load the whole FAT to the memory without scanning the whole disk. From then on, seeking can be done without touching disk.

Indexed improvement: one don't need to keep the whole FAT. Instead, just keep the index of the files that are currently used. Also, it needs constant time to seek.

POS(0230A)-12.3

Free-space management

The FS must be able to find blocks ready for use by new files. An on-disk data structure called **free-list** supports this operation:

- **Marks in FAT:** FAT has an entry for each block, so a special mark (e.g., all bits 1) can be used to denote a free block there.
- **Linked list:** Each free block stores an address that gives the disk address of the next free block.
- **Free bitmap:** The FS has a bit-array, one bit per block, that is 1 when the corresponding block is used and 0 otherwise.

The free blocks are expected to be **heavily fragmented**. OS should prevent this from causing **files** to be heavily fragmented as well, since seeking in disk is rather slow.

Contrast this with paging mechanism in VM.

POS(0230A)-12.4

Directory structure

Given the name of a file, one must be able to find the block-location structure of that file. This is done by a **directory**.

The directory itself is stored as file data, storing a name-to-file-object mapping (except the root directory might be stored in FS header). Its content might be:

- An array of **fixed size file records**. (e.g., DOS FAT, fix length filename)
- An array of fix edsize records, but one file **occupies multiple records** to support long filename (e.g., VFAT).
- An unordered records of **variable sizes**. (e.g., Linux ext2.)
- A **B+-Tree** of records. (e.g., Linux Reiserfs, NTFS, ext2+htree.)

They affect the **efficiency to search** and **add files** into directories. E.g., the unordered solutions are inefficient if one directory contains many files.

POS(0230A)-12.5

Case study: Linux ext2 file system

Default FS for Linux (ext3 is the most widely used extension).

- Files occupies a **integral number of fixed-sized blocks**.
- Block size can be chosen from 1024, 2048 and 4096. A small block size wastes less space, a large block size keeps files more contiguous.
- Some blocks are reserved for **index nodes** when **creating** the FS.
- Block bitmap** is used to keep track of free blocks. Directories are in **list** form. Standard **variable-level indexing** is used to locate data.
- Default data block size is 1024-bytes. Files should be mostly **contiguous** to for this small size to be efficient.
- Disk is partitioned into equal-sized groups.

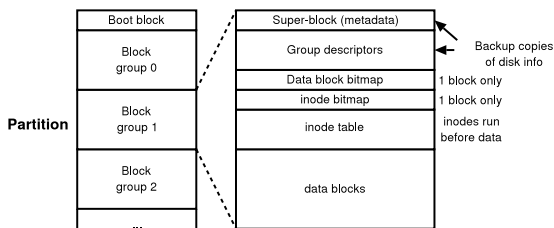
POS(0230A)-12.6

Ext2: technical overview

- The first 1024 bytes forms the **boot block** for storing a boot program.
- The remainder of the partition is divided into **groups**. Each group contains some information about the group, and then the data.
- The first group (“master group”) has the following **summary data structure**. They are also present in some other groups as **backups**.
 - A **superblock** stores information like block size, last fsck time, etc.
 - Information about all groups are stored in a **group descriptor**.
- Each group has a block bitmap, which is in exactly one block. So the **size of block in bits is the same as size of group in blocks**.
- Each group has an **inode table** to store inodes. Each inode is 128-bytes. Each group has a corresponding **inode bitmap** to record whether each inode is in use. Other data are stored after the inode table.

POS(0230A)-12.7

Disk organization with ext2



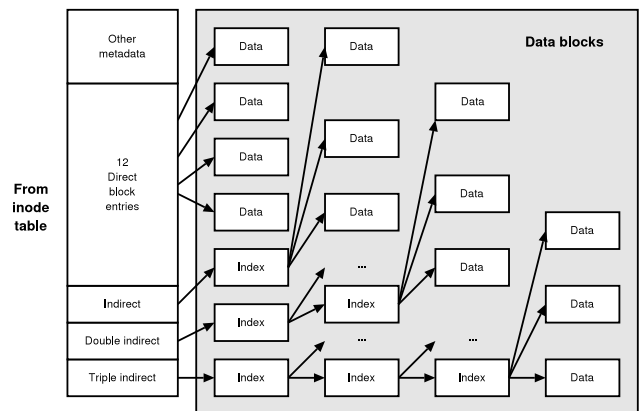
Each file needs one inode. It is used to hold some **indices** to data blocks. There are 4 indices, at depths 0, 1, 2 and 3.

Inodes also stores some descriptive information of the file: file type, file size, file dates, file permission info, link counts, etc.

The **filename is contained in the directory**, not the inode.

POS(0230A)-12.8

Index organization in ext2 inodes (Variable level indexing)



POS(0230A)-12.9

Ext2: Fighting against fragmentation

For performance, it is vitally important for ext2 to be kept relatively unfragmented—especially when ext2 uses block size as small as 1K.

- When allocating data blocks for **directories**, Linux tries to spread out to all **different groups** of the disk.
- When allocating data blocks for **files**, Linux tries to allocate it in the **same group as the directory** holding it.
- To **append** to a file or directory, Linux first tries to find free blocks **close to the last block**, and allocate those blocks if found.
- If that can't be done, the file must be fragmented. It tries to locate a place in the group that has **at least 8 consecutive free blocks**. (Actually, a “free byte” in the block bitmap.)
- If found, all 8 blocks are **pre-allocated** to the file, even though the file might just need one. They are released when the file is closed.

POS(0230A)-12.10

Filesystem consistency

During system failures, the data in cache usually does not have a chance to be written back to disk.

In particularly bad situations, the whole FS can be at risk:

E.g., if the superblock is being modified when the system crash or power is out, the vital information about the FS is gone.

Consistency checker: During bootup, the system first checks whether the FS is consistent, and correct mistake.

Consistency checker is slow. Usually a dirty flag is set the last when the FS is unmounted, so that the consistency checker can be skipped.

Changing metadata can leave the system in dangerous states. **The order of operations** usually have important role to play to prevent them. E.g., decrementing inode reference count before directory entry is dangerous.

POS(0230A)-12.11

Journaling (or log-based) filesystems

A **journal** can be used to eliminate the need for FS checking.

- All operations on meta data are done first to a place in the FS called the **journal**. It records exactly what needs to be done, e.g., changing a directory entry, allocating a particular inode to a file ,etc.
- Then the journal is then marked as active. At this time the change of meta data is said to be **committed**.
- Then the actual modification is done, and the journal is marked inactive.
- If the journal is not empty during boot time, **all the committed journal entries are redone**. This brings the FS back to consistent. It must be okay to **redo** the operation.
- Journaling also **help reducing FS latency**, since the time-critical meta-data writes are all done in the small journal. Otherwise operations like changing block bitmap must be done ASAP.

POS(0230A)-12.12

Delaying reads and Disk scheduling

Consider that you want to perform a file read from the FS, leading to 10 blocks being read. The OS suddenly find that 5 other processes wants to use the device as well. It can...

- Wait until these processes finish, then perform the read. This is called **first-come-first-serve Problem**: the disk head moves back and forth 5 times before your request, when it go back or forth once more.
- **Solution**: make 10 new buffer, **request** the disk scheduler to fill the buffer, and we wait for all the buffers to be filled. The OS rearrange the ordering of the reads when filling the buffers.
- Question 1: **what** algorithm to use for rearranging the requests? We usually call such algorithms an "elevator".
- Question 2: **how** to **implement** the algorithm? We can of course directly implement it, but it would be more efficient if we can do it implicitly. E.g., FCFS can be implemented in terms of a queue of buffers.

POS(0230A)-12.13

CLOOK: a disk scheduling algorithms

CLOOK is a common way to scedule disk. Basically, it tries to keep the disk head moving in the **same direction** as long as possible.

- Suppose currently the disk head is at position x.
- Find the request with position **after x** that is closest to x.
- If found, serve the request. Otherwise, find the request with the **smallest position**. Serve the request.

How to implement it? One might uses a queue of requests:

- The request queue is always kept to be in the order of service. It has **two sorted parts**, first for those requests with position after the current request, then for those before the current requests.
- When a request is made, it checks whether the currently served request is before or after the new request, and adds to the corresponding part.

POS(0230A)-12.14

Problem of CLOOK

There are problems associated with CLOOK, though:

- Requests are not all the same. E.g., **meta-data** accesses are more important than other read and write, and needs to be serviced first.
- CLOOK can lead to **starvation**: when one process requests a large amount of consecutive data, no one else can proceed.

Linux 2.4 solution

- **Meta-data** accesses are put to the **front** of the queue, skipping the CLOOK algorithm.
- Each request has a "**sequence number**" which starts with some constants. Everytime a new request is put in **front** of it, the sequence number is decremented. If it ever reaches 0, it moves to the head. These constants can be set using the `elvtune` utility, and is initially 8k and 16k for read and write respectively. See `<kernel>/include/linux/elevator.c`.

POS(0230A)-12.15