UNIX File Systems & Journaling

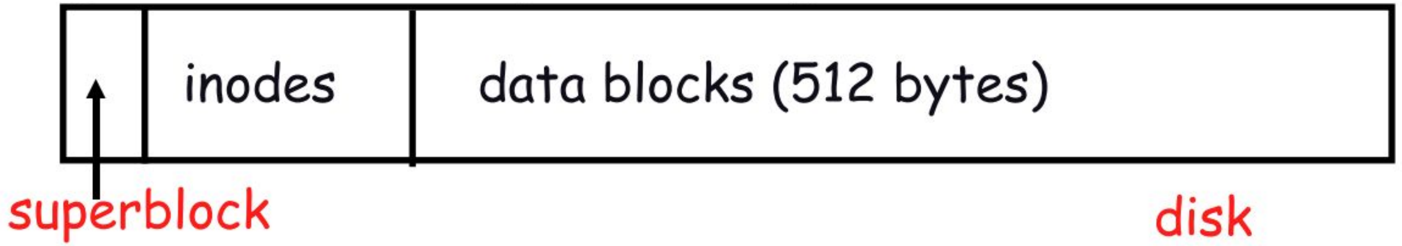
W4118 Operating Systems I

https://cs4118.github.io/www/2024-1/

Credits to Jae and David Mazières

Original Unix FS

Simple and elegant



Components

- Data blocks
- Inodes
- Superblock (specifies number of blks in FS, counts of max # of files, pointer to head of free list)

Problem: Slow

Performance Costs

Blocks too small (512 bytes)

- File index too large
- Too many layers of mapping indirection
- Transfer rate low

Poor clustering of related objects

- Consecutive blocks not close together
- Inodes far from data blocks
- Inodes for file in the same directory are not close together
- Poor enumeration performance: e.g., “ls -l”, “grep foo *.c”

More Modern UNIX File System Architecture

Multi-level indexed block allocation

- I(ndex)node is the internal representation of a file, holds data block pointers and other metadata
- Used by FFS, ext2, ext3

Design filesystem with disk geometry in mind

- **Cylinder groups**: same concentric track across platters
- Since modern devices don’t expose geometry, could also use **block groups**: contiguous regions of the logical block address space.
- Keep related data within the same group to minimize seeks!

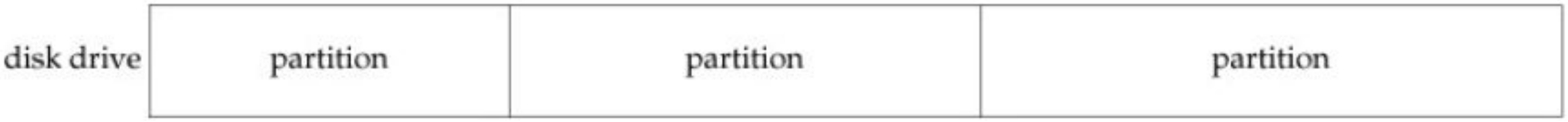
Berkeley Fast File System (FFS) Layout

Disk drive can be partitioned into multiple operating systems

- e.g., dual-boot Linux and Windows

Within a single OS, can also partition disk into several filesystems

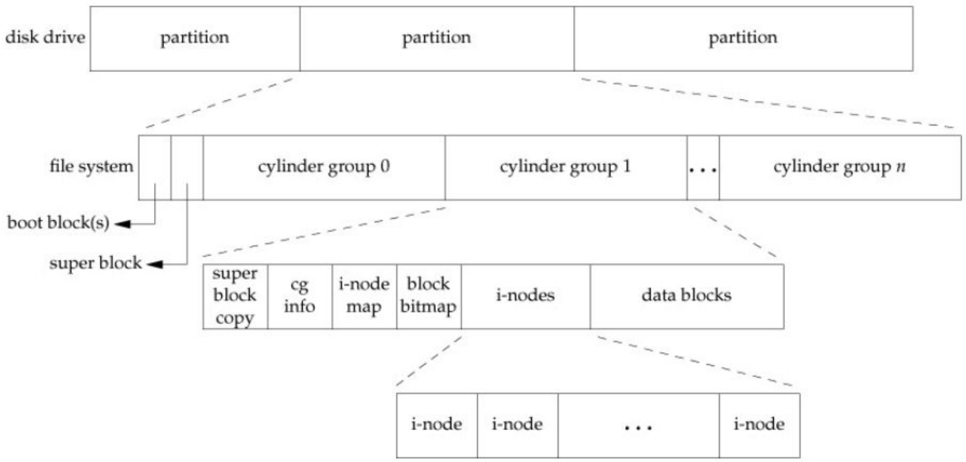
- use different filesystems for different purposes
- in UNIX, all mounted filesystems are grafted into the directory hierarchy tree



Berkeley Fast File System (FFS) Layout

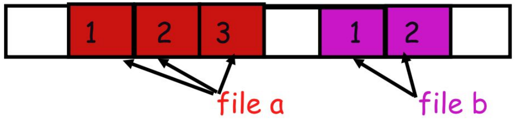
A file system occupies a disk partition. At the top-level of FFS we have:

- **super block**
  - metadata about the filesystem (#blocks, #groups, block size, etc.)
- **boot block(s)**
  - for OS partition, place boot loader at a known place (e.g. at the very start of the partition) for the hardware to locate and execute
- **cylinder group partitions**
  - place inodes and data blocks into the same cylinder group to minimize disk seeks

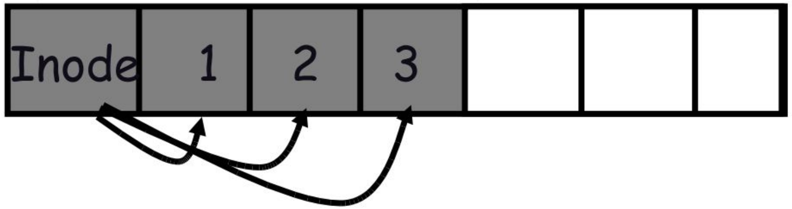


Clustering Related Objects

- **Tries to put sequential blocks in adjacent sectors**
  - Access one block, probably access next



- **Tries to keep inode in same cylinder group as file data**
  - If you look at inode, most likely will look at data too.

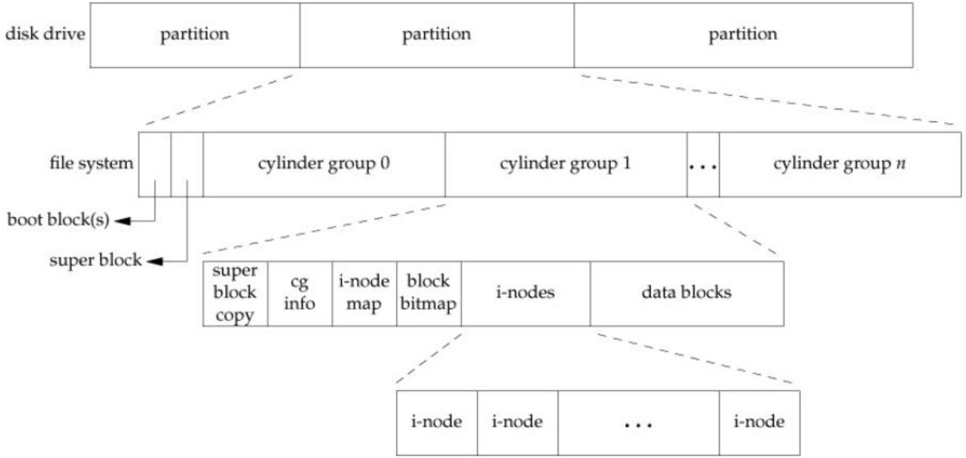


- **Tries to keep all inodes a dir in same cylinder group**
  - Access one name, frequently access many, e.g., “ls -l”

Berkeley Fast File System (FFS) Layout

A cylinder group maintains a copy of the superblock and some cylinder group metadata for performance. The crucial parts of the file system are:

- **inode bitmap**
  - **which inodes are used/unused**
- **block bitmap**
  - **which data blocks are used/unused**
- **array of inode blocks**
  - **stores per-file inodes**
  - **note that an inode uniquely identifies a file, NOT the filename – more on this later**
  - **#inodes is effectively the #files you can have on the filesystem**
  - **sizeof(inode) ~ 128B, sizeof(datablock) ~ 4KB, should be able to fit quite a few**
- **array of data blocks**



Finding space for related objects

Old Unix: Linked list of free blocks

- Just take a block off of the head. Easy!
- Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow

FFS: switch to bit-map of free blocks

- 10101011111110000001111111000101100
- Easier to find contiguous blocks
- Small, so usually keep the entire thing in memory
- Time to find free block increases if fewer free blocks

Using the bitmap

Usually keep entire bitmap in memory

- 4G disk / 4K blocks. How big is the map?

Allocate block close to block x

- If the disk is almost empty, will likely find one near
- As disk becomes full, search become more expensive and less effective

Keep a reserve (e.g., 10%) of disk always free, scattered across the disk

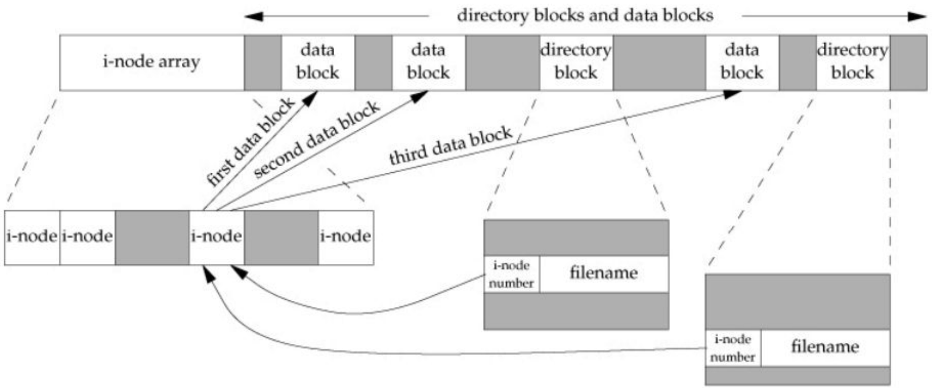
- Don’t tell users
- Only root can allocate blocks once FS 100% full
- With 10% free, can almost always find a nearby free block

Inodes and Data Blocks

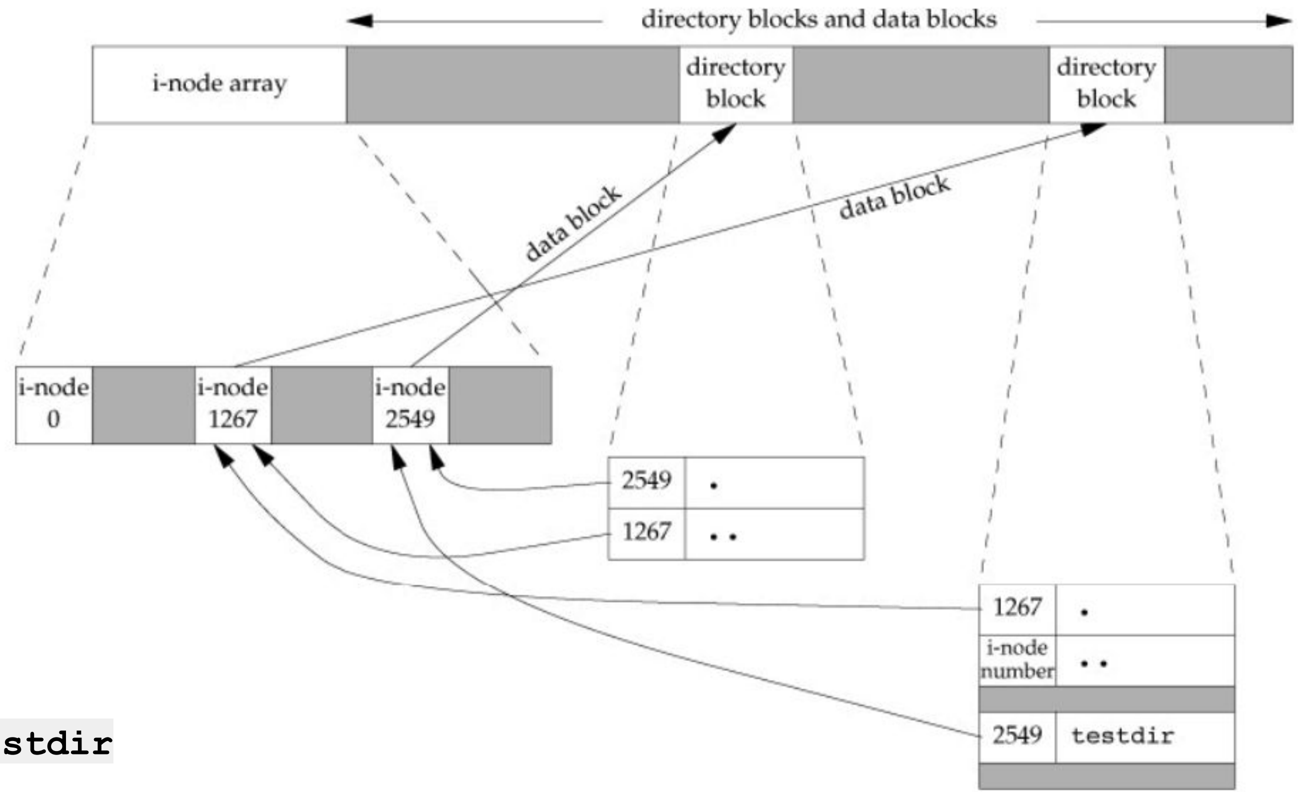
A given inode in the inode array represents a single file

Directories are pretty much just "special" files – they also occupy data blocks. A directory's data block houses directory entries:

- one dentry per file in the directory
- each dentry has the name of the file and the inode
- notice that two different dentries can refer to the same inode – files are uniquely identified by inode number in a filesystem, not the filename!



Inodes and Data Blocks Example



`mkdir testdir`

Summary

Symbolic link

- Special file, designated by a bit in metadata
- File data is name to another file

Hard link

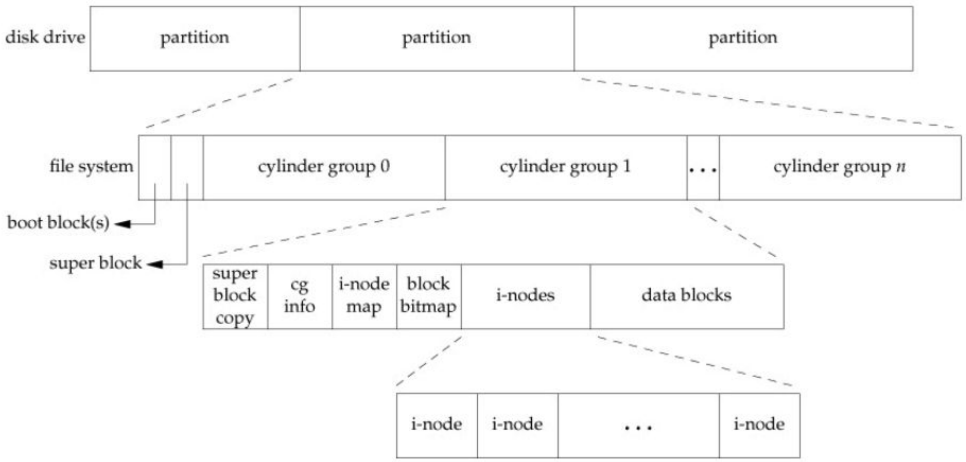
- Multiple dentries point to the same file
- All hard links are equal: no primary
- Store link count in file metadata
- Cannot refer to directories or files outside fs

What about consistency?

Writes require several steps:

- Update inode/block bitmaps
- Update inode
- Update data blocks

What if the system crashes?



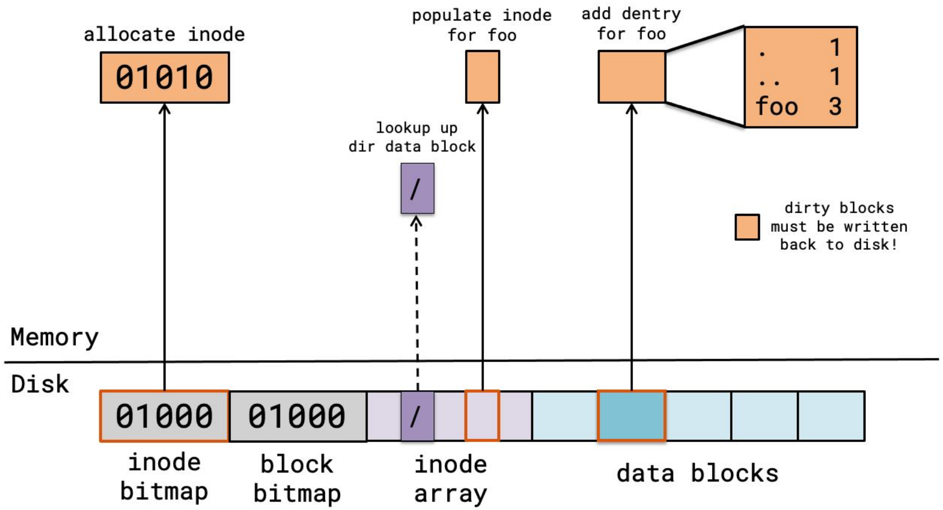
Example: ext2 empty `foo` file creation

Let's analyze possible crash scenarios. Define B, I, D as follows:

- inode bitmap update (B)
- add inode for foo (I)
- add dentry for foo to dir data block (D)

Assume that writes within a block happen atomically

```
B = 01000   ---> B' = 01010
I = garbage ---> I' = initialized
D = {., ..} ---> D' = {., .., foo}
```



Crashes can lead to inconsistencies

```
B   I   D  --->  Consistent (new data lost)

B'  I   D  --->  Inconsistency! Bitmap says I was allocated,
                 but no one is using it (leak)

B   I'  D  --->  As if nothing happened! we wrote to the inode
                 but map still says its garbage

B   I   D' --->  SERIOUS PROBLEMS: dentry exists, but points to garbage inode.
                 bitmap says that inode is free, can be taken by another file.

B'  I'  D  --->  Inconsistency! Bitmap says I was allocated, and we wrote to I,
                 but no one uses I.

B'  I   D' --->  MOST SERIOUS PROBLEM! FS is consistent according to bitmap and
                 dentry, but inode has garbage data.

B   I'  D' --->  Inconsistency! Dentry refers to valid I, but bitmap says I is free.
                 I can be taken by another file.

B'  I'  D' --->  Consistent (new data persisted)
```

fsck: file system consistency check

In the old days, reboot after crash and scan entire disk to make fs consistent

Disadvantages:

- slow to scan large disk
- cannot correctly fix all crash scenarios, e.g., `B' I D'`
- no well-defined consistency, e.g., what do we do for `B I D'`?

Solution: Journaling

Keep a write-ahead log

Persistently write intent to log/journal, then update filesystem

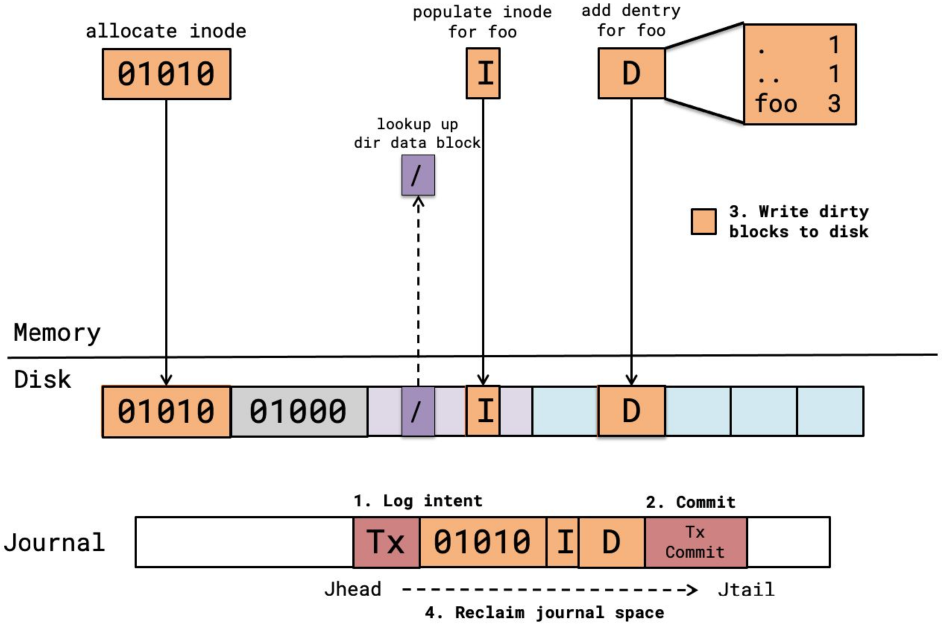
- crash before intent is committed: noop
- crash after intent is committed: replay op

Better than fsck:

- no need to scan entire disk
- well-defined consistency

Example: ext3 physical journaling

- Commit dirty blocks to journal as one transaction
- Write commit record (finalize journal entry)
- Write dirty blocks to real file system
- Reclaim journal space for transaction (we don't need it anymore)



Journaling Write Orders

1. **Journal writes, then FS writes**
   - otherwise, crash will leave FS inconsistent but no journal record to patch it up
2. **FS writes, then reclaim journal space**
   - otherwise, if you crash before you finish the FS write, the journal record to patch it up will already be gone!
3. **Journal writes, then commit record, then FS writes**
   - we need the commit record to tell us that we journaled the entirety of the change. Otherwise, the journal may have garbage in it!

ext3 Journaling Modes

Motivation: journaling is expensive. Every FS write requires two disk writes, two seeks. Balance consistency and performance…

Data journaling: journal all writes, including file data

- Problem: expensive to journal data

Metadata journaling: journal only metadata

- Used by most FS (IBM JFS, SGI XFS, NTFS)
- Problem: file may contain garbage data

Ordered mode: write file data to FS first, then journal metadata

- Default mode for ext3
- Problem: if crash before writing metadata, then you end up with old file metadata and new file data, where the journal says everything is OK