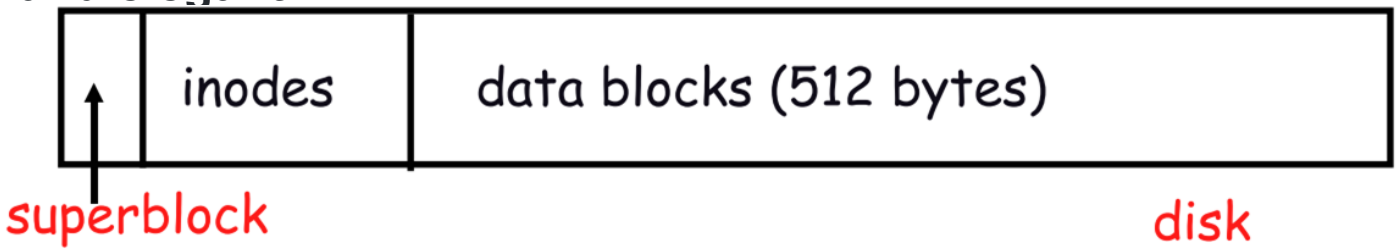
File Systems

W4118 Operating Systems I

columbia-os.github.com

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!

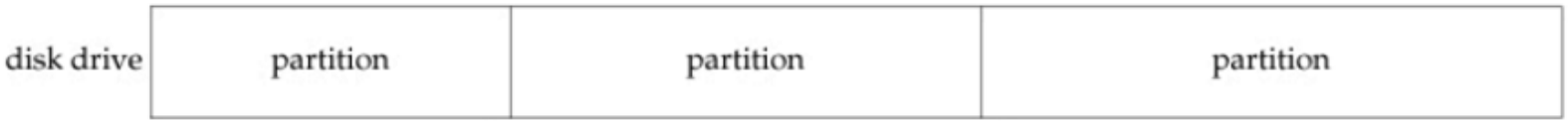
Berkeley Fast File System (FFS) Layout

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

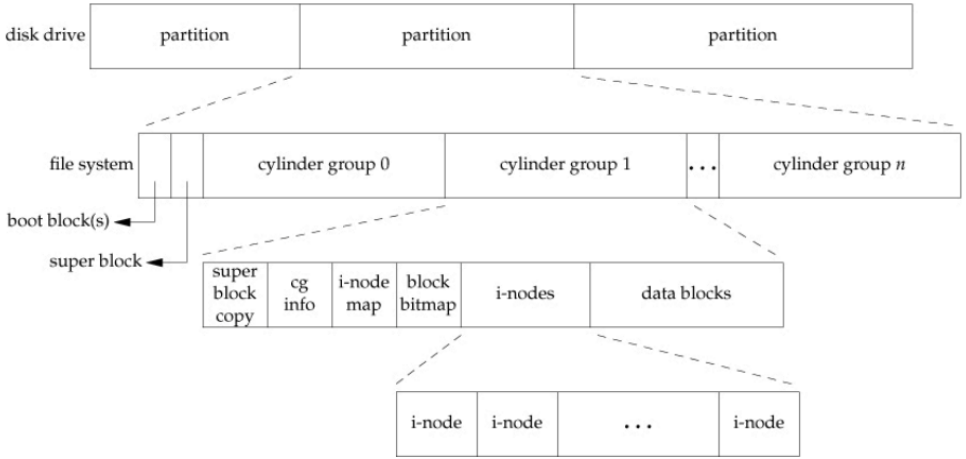
- use different filesystems for different purposes
- in UNIX, all mounted filesystems are grafted into the directory hierarchy tree



Berkeley Fast File System (FFS) Layout

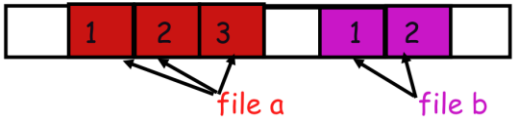
A file system occupies a disk partition. At the top-level of FFS we have:

- **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

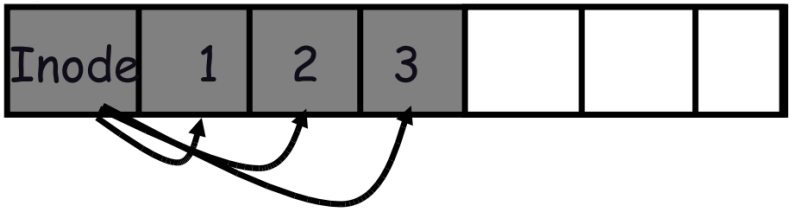


Clustering Related Objects

- **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.

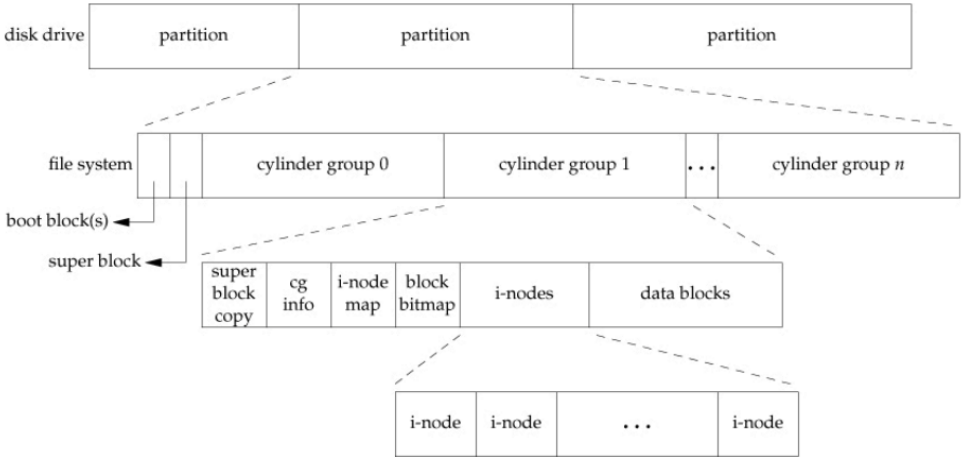


- **Tries to keep all inodes a dir in same cylinder group**
  - Access one name, frequently access many, e.g., “ls -l”

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

- 10101011111110000011111111000101100
- 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

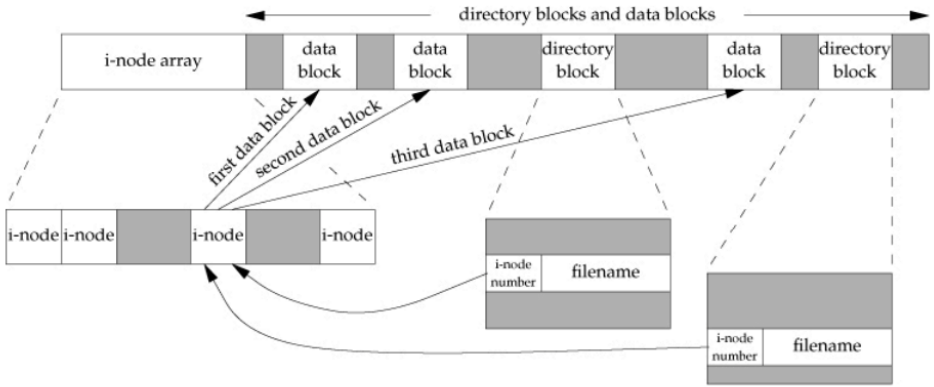
- Don’t tell users
- Only root can allocate blocks once FS 100% full
- With 10% free, can almost always find a nearby free block

Inodes and Data Blocks

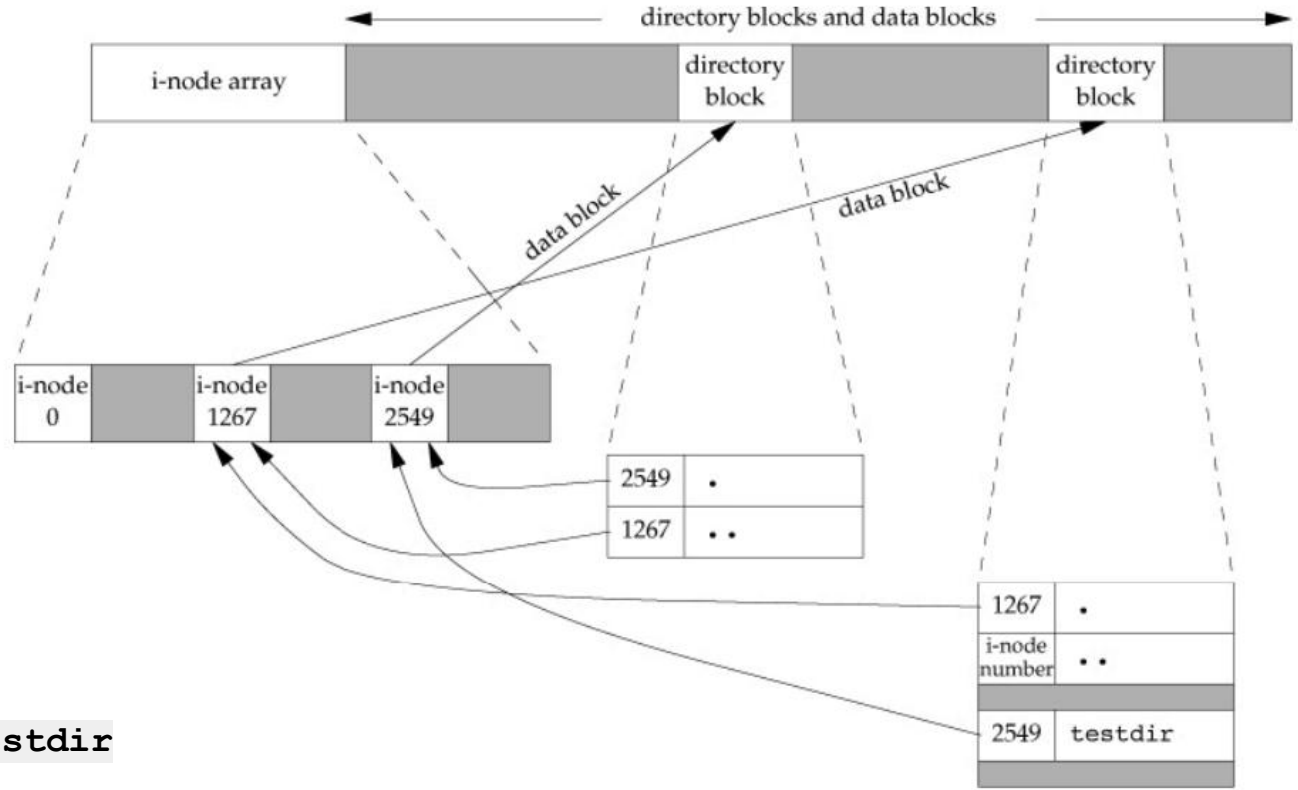
A given inode in the inode array represents a single file

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

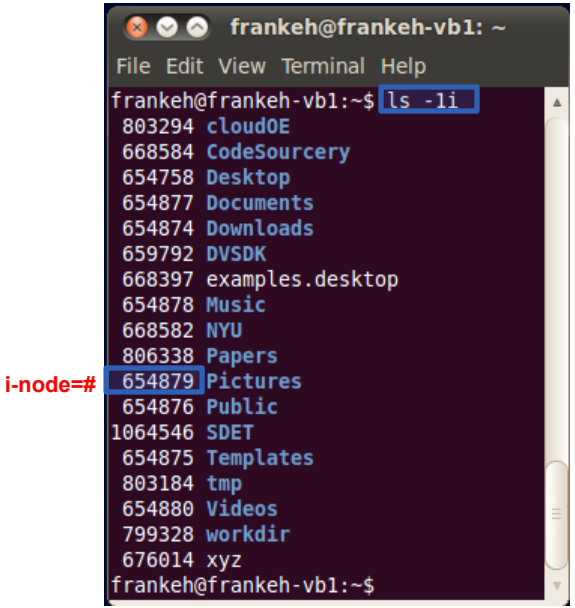
- Special file, designated by a bit in metadata
- File data is name to another file

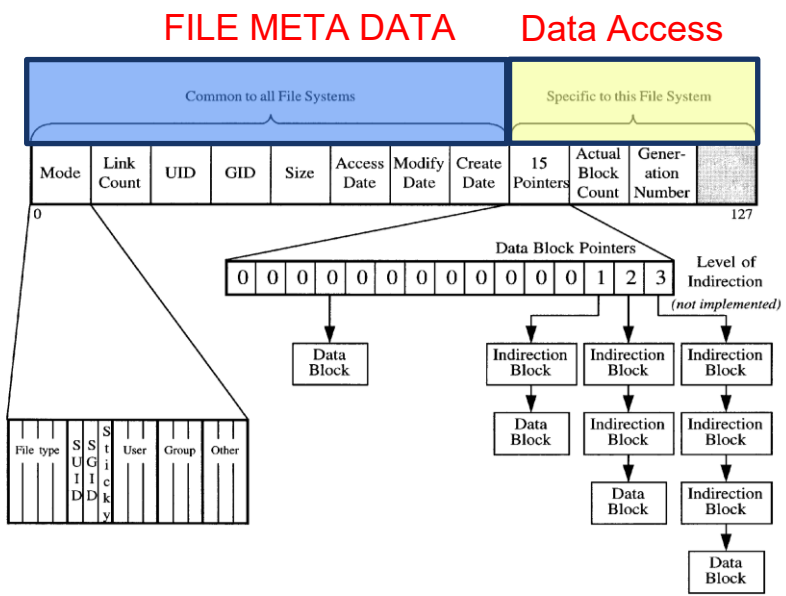
Hard link

- 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

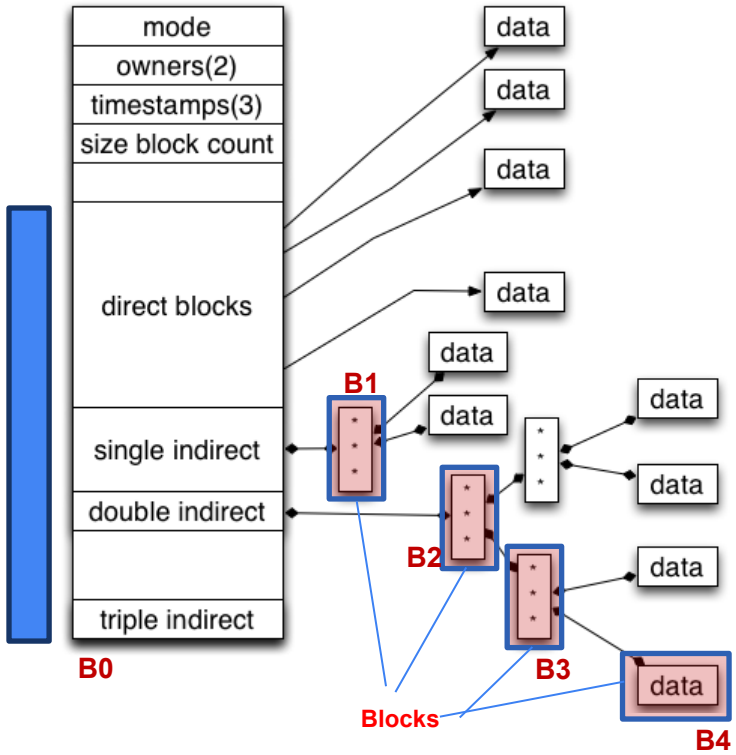
Implementing Files: I-nodes

- A data structure associated with each file (and directory)
- Lists the attributes and disk addresses of the file blocks
- Need only be in memory when the corresponding file is open
- (typically 128 bytes)





Implementing Files: i-nodes (ext2)



Properties:

- Small file access fast
- Everything a block
- Huge files can be presented
- Overhead (access, space) proportional to file size

To get to this data we must read
1) inode (B0)
2) double indirect block (B2)
and (B3)
3) data block (B4)

→ if uncached, 4 disk block reads

Example for File access with fixed size blocks

- Assume blksz=1K (2^10) and block number a 4 byte integer
  so the data is stored in 1 KB blocks and 256 (==2^8) block id's can fit in an indirection block

How much data range is covered at each level ??

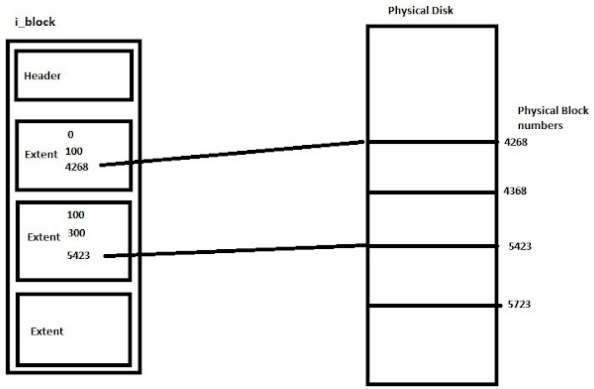
- `Direct Access (12 entries) cover 12* 2^10 = 12KB`
- `Indirect: 2^8 * 2^10 = 2^18 = 256KB`
- `Double Indirect: 2^8 * 2^8 * 2^10 = 2^26 = 64MB`
- `Triple Indirect: 2^8 * 2^8 * 2^8 * 2^10 = 2^34 = 16GB`
- `Quad Indirection: 2^8 * 2^8 * 2^8 * 2^8 * 2^10 = 2^42 = 4TB`

- Then: fileofs=260KB is covered by Indirect as it covers 12KB – 268KB

- Note: the math is different if you go to quad pointers (only 11 direct then) or if blksz is different.

Extent based Filesystems (ext4)

- Inodes, but instead of fixed size blocks, each entry (direct,indicrect,..) is an extent: [file-offset, phys-block, length]
- The extent is phys contiguous
- Significantly drives down fragmentation and disk
- Example: ext4




```
root@lnx2:~# filefrag -v /boot/vmlinuz-5.15.0-69-generic
Filesystem type is: ef53
File size of /boot/vmlinuz-5.15.0-69-generic is 11570216 (2825 blocks of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..    2047:     927744..    929791:   2048:
   1:     2048..    2824:     933888..    934664:    777:     929792: last,eof
/boot/vmlinuz-5.15.0-69-generic: 2 extents found
```

syscalls to retrieve meta data

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

`ls -ls` translates to

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Listing content of a directory

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
```

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

```
main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("  %s\n", entry->d_name);
    closedir(dir);
  }
}
```

These are the fundamentals of the /bin/ls command

When you combine with the stat() syscall

LFS: Log-structured File Systems (1)

- Disk seek time does not improve as fast as CPU speed, disk capacity, and memory capacity.
- File Data Caching a key strategy to keep reduce the overhead.
- Disk caches can satisfy most requests (see buffer cache or page cache (before))

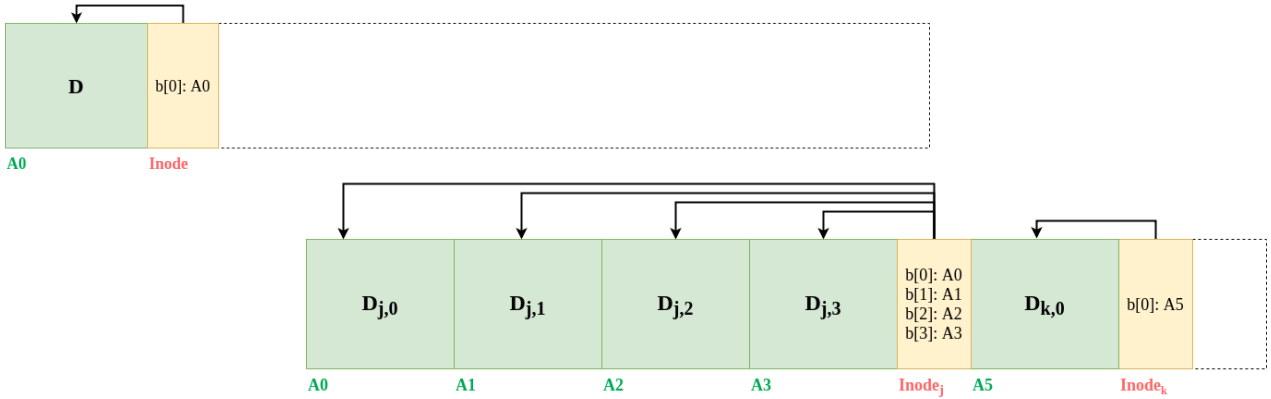
➔ The theory then:
in the future most disk accesses will be writes

➔ LSF optimizes for writes !

LFS: Log-structured File Systems (2)

- Basic idea: Buffer all writes (data + metadata) using an in-memory segment; once the segment is full, write the segment to a log (aka checkpoint)
- The segment write is one sequential write, so its fast
- We write one large segment (e.g., 1 or 4 MB) instead of a bunch of block-sized chunks to ensure that, at worst, we pay only one seek and then no rotational latencies (instead of one seek and possibly many rotational latencies)
- There are no in-place writes as in previous discussed filesystem implementation
- Reads still require random seeks, but physical RAM is plentiful, so buffer/page cache hit rate should be high (more on that later)

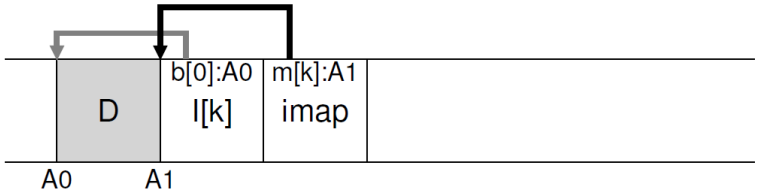
LFS: Segments



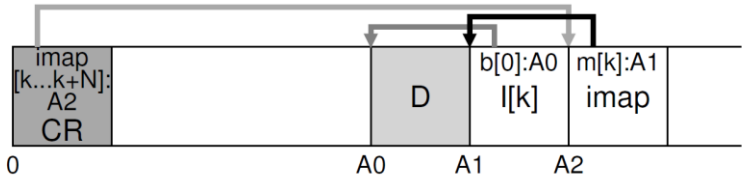
- In order to determine whether a block is stale, you need to know its identity.
- This is stored in an additional part of the segment called the **segment table**.
- The segment table contains an entry for each block in the segment, which identifies which file (inode number) the block is part of, and which part it is (e.g. "direct block 37", or "the inode").

Log-Structured File Systems (1)

- i-nodes now scattered over the disk
- Part of the i-node map is written with inode/block to log



- An i-node map, indexed by i-number, is maintained.
- The map is kept on disk at fixed location and is also cached.



- Checkpoint region (CR) only update periodically (30secs)

Log-Structured File Systems (2)

- Disks are not infinite, hence at some point no further segment can be written to the log

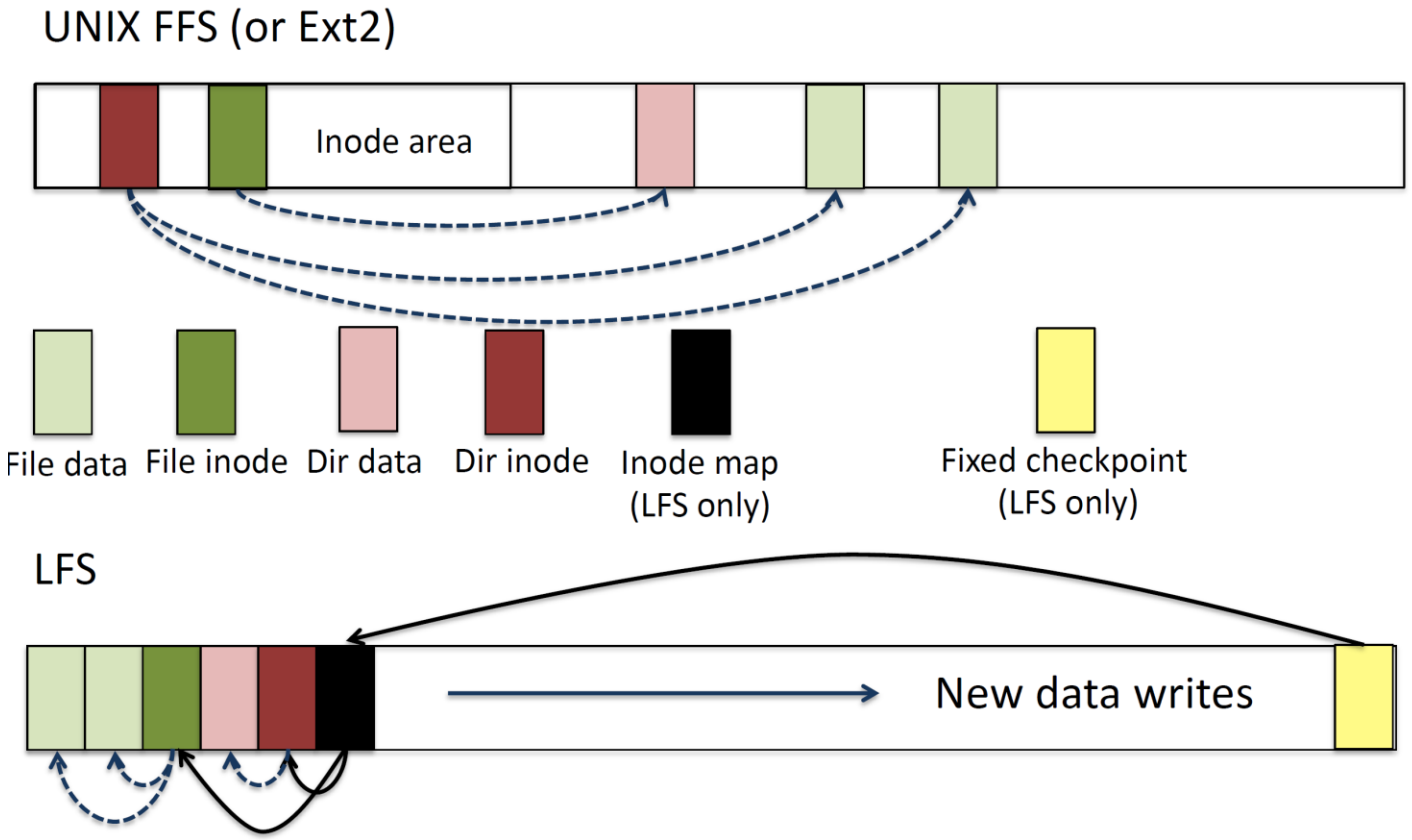
- But many segments contains blocks that are no longer needed.
  - Block overwritten with a new block (now in a different segment)
  - E.g. file created then deleted (think compile)

➔ Cleaner Thread (kernel)

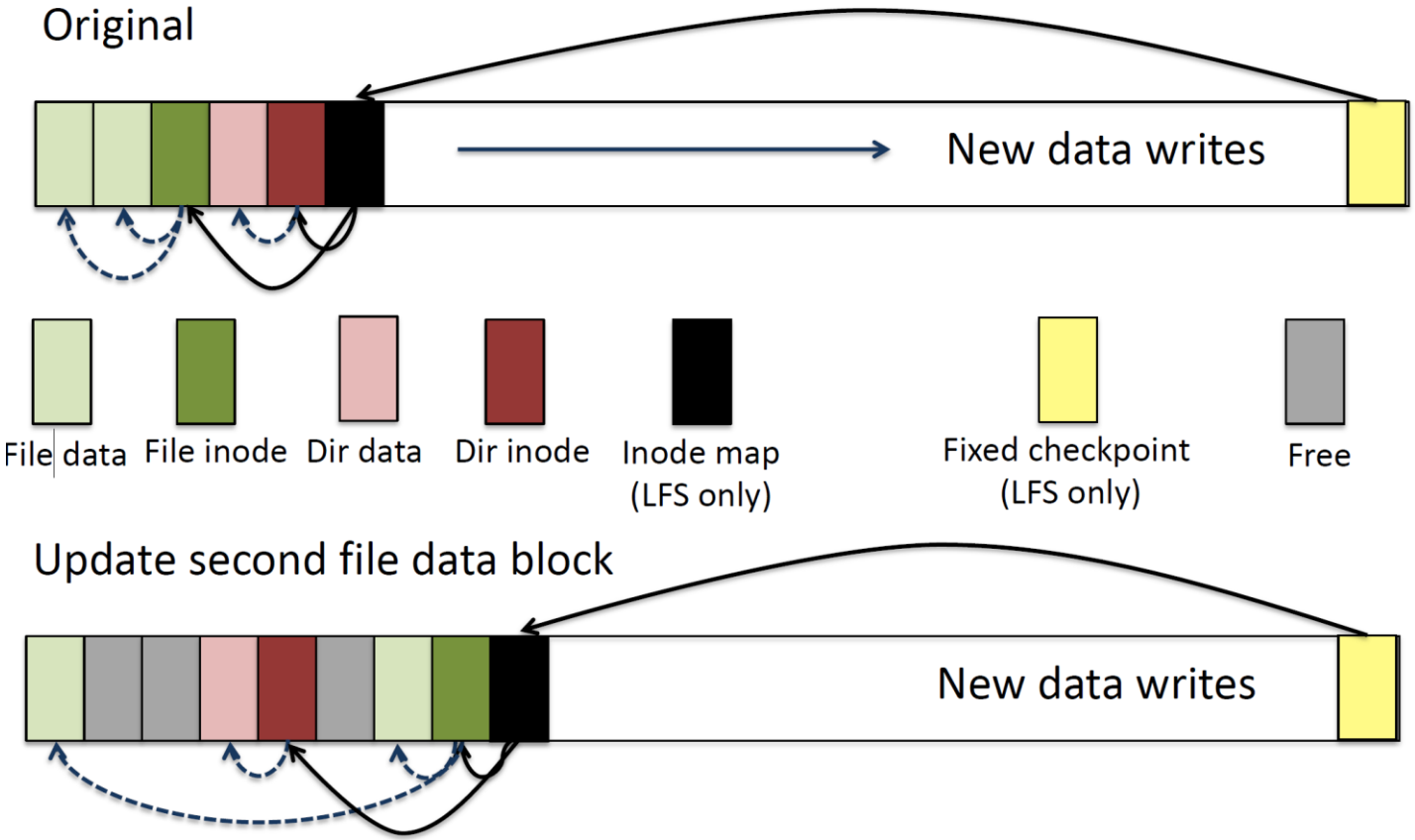
LFS (Cleaner Thread)

- Scan the log circularly to compact it
- Reads in the summary of the next segment to identify inodes and blocks
- Looks up inode map to check whether
  - inode is still in the current inode-map (deleted ?)
  - inode is still current (current pending write , so will be overwritten)
- Inodes and blocks still in use will go into memory and written to next segment
- Original Segment (and now compacted) is marked free
- This way disk is a big circular buffer
  - Writer Thread adds new segments to the front
  - Cleaner Thread removing old ones from the back
- Crash in period between checkpoints results in data loss or inconsistency

LFS: Efficient Reads

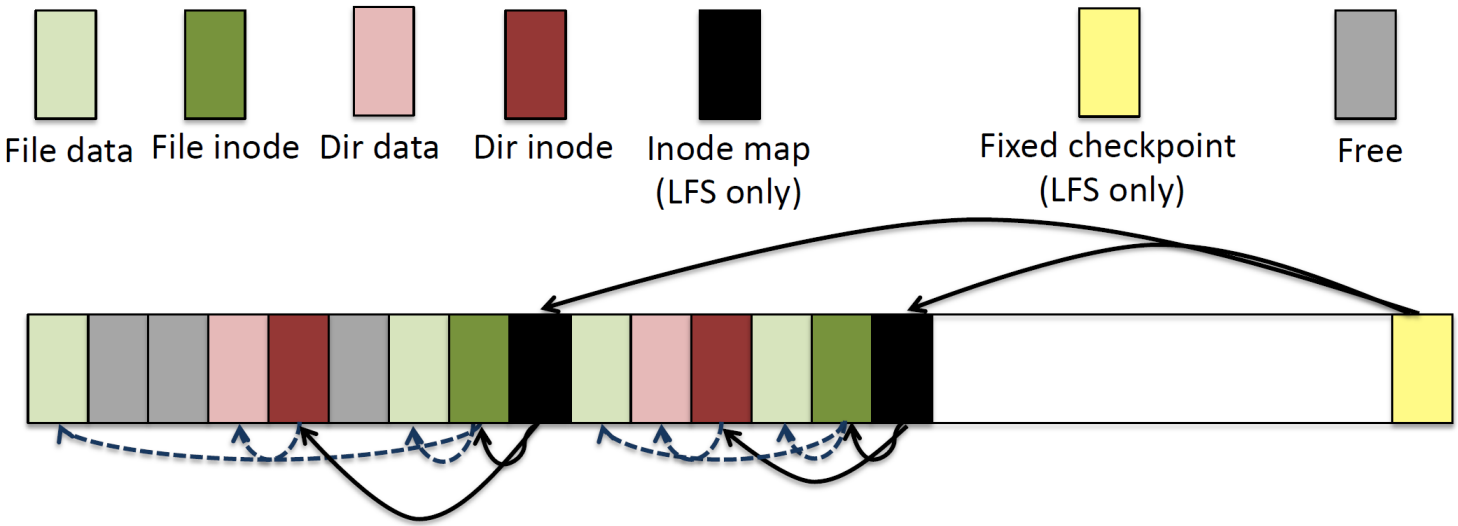


Writes: Copy-on-Write



Disk Cleaning

- When disk runs low on free space
  - Run a disk cleaning process
  - Compacts live information to contiguous blocks of disk



In reality, too expensive to clean contiguously.
FS is split into moderately large segments (e.g., 1MB or more).

Improving strict LFS

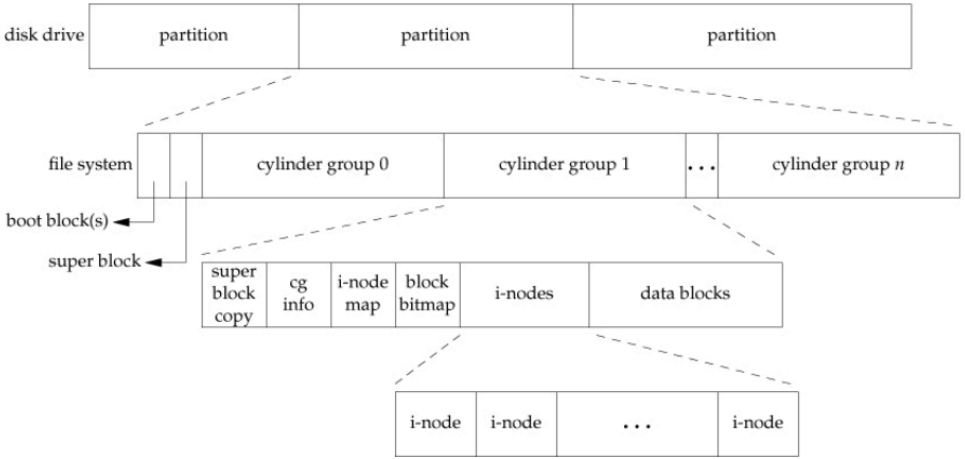
- This sort of garbage collection of stale inodes and data blocks can lead to:
  - Increased I/O load and write amplification
- Problem: long-lived data repeatedly copied over time
  - Solution: Group older files into same segment
  - Old segments won't have many changes. Skip.
- To reduce the overhead, most implementations avoid a strict circular log, but maintain it as a list of segments with an order to inspect, where the head of the log
  - Advances to a (non-adjacent) segment that is already free
  - Inspect the least-full segment first
- Improvement is increasingly ineffective as the file system fills up and approaches full capacity.

What about consistency?

Writes require several steps:

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

What if the system crashes?



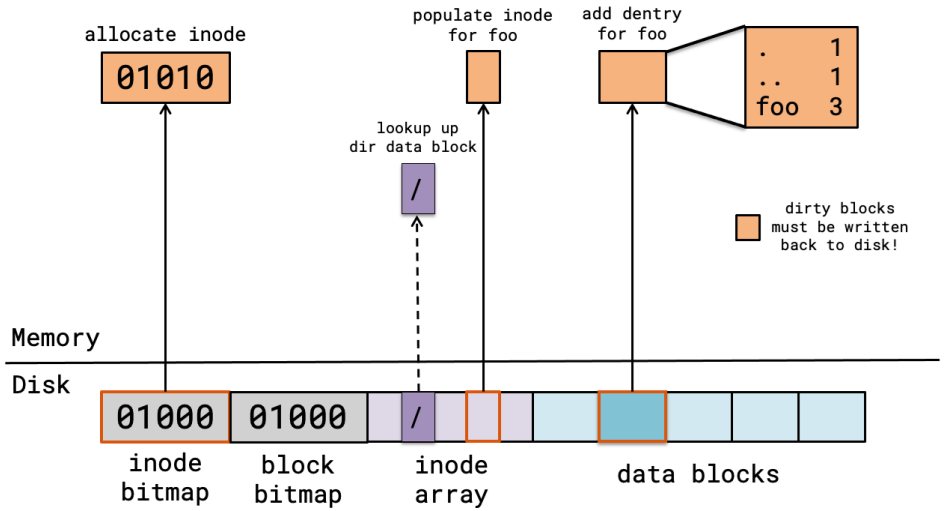
Example: ext2 empty `foo` file creation

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)
```

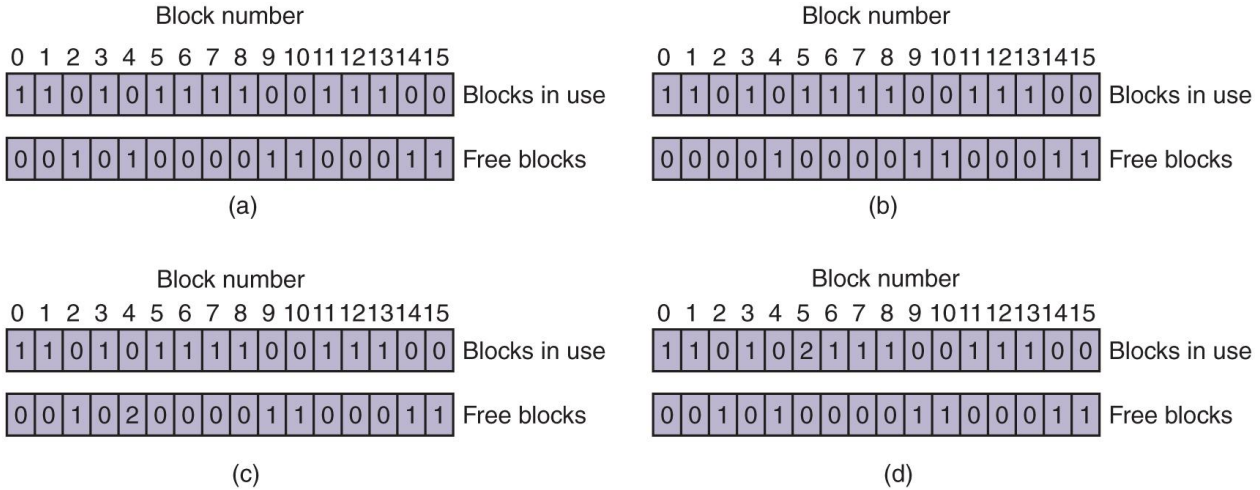
fsck: file system consistency check

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

Disadvantages:

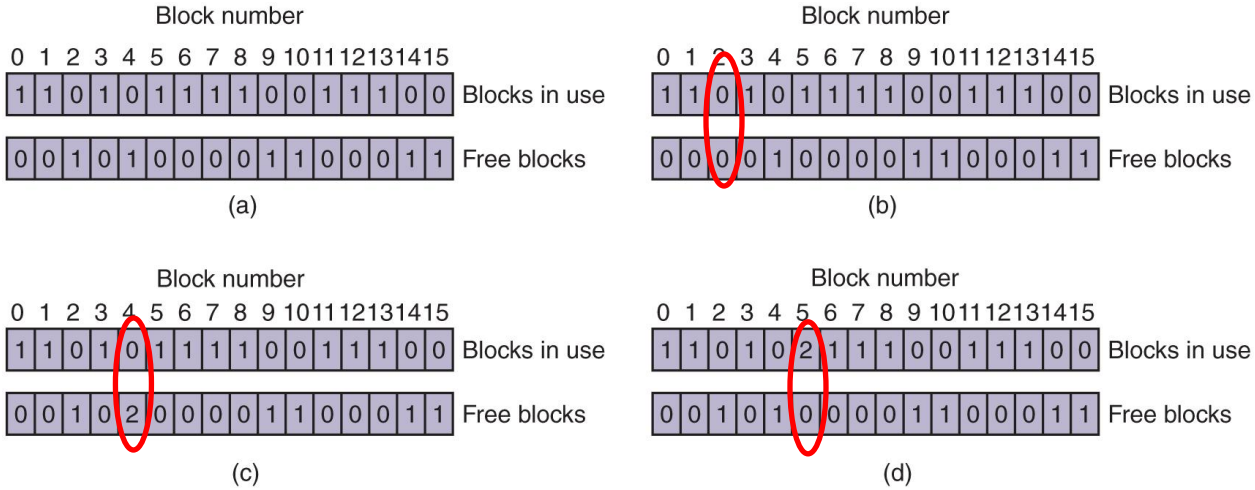
- slow to scan large disk
- cannot correctly fix all crash scenarios, e.g., `B' I D'`
- no well-defined consistency, e.g., what do we do for `B I D'`?

File System Consistency: Blocks (1)



File-system states:
(a) Consistent.
(b) Missing block.
(c) Duplicate block in free list.
(d) Duplicate data block.

File System Consistency: Blocks (2)



Add the block to the free list

Rebuild the free list

Allocate a free block, make a copy of that block and give it to the other file.

Solution: Journaling

Keep a write-ahead log

Persistently write intent to log/journal, then update filesystem

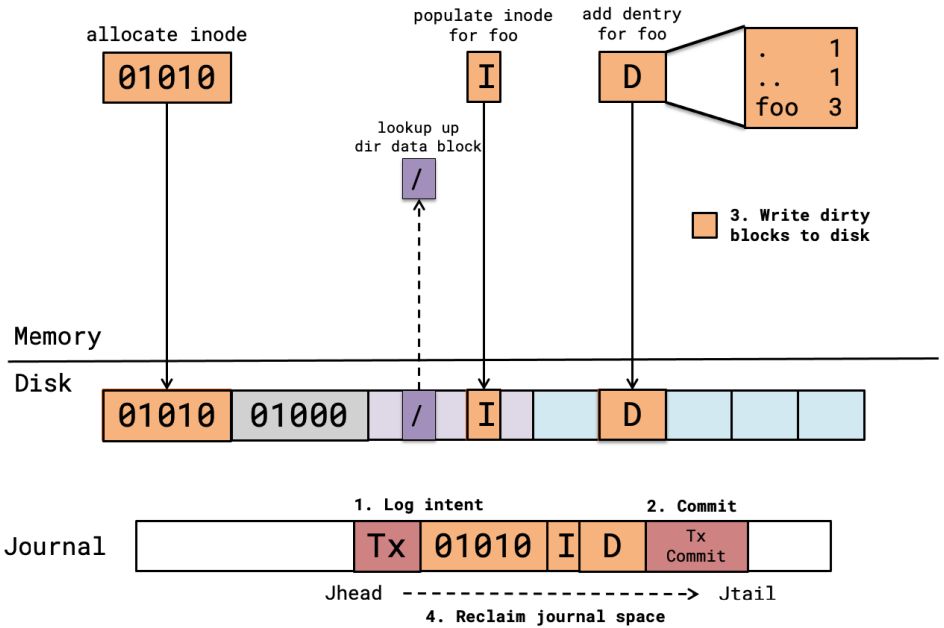
- crash before intent is committed: noop
- crash after intent is committed: replay op

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 journaling metadata, then you end up with old file metadata and new file data, where the journal says everything is OK

E Pluribus Unum → Virtual File Systems
Supporting Many File System under Unix

- s

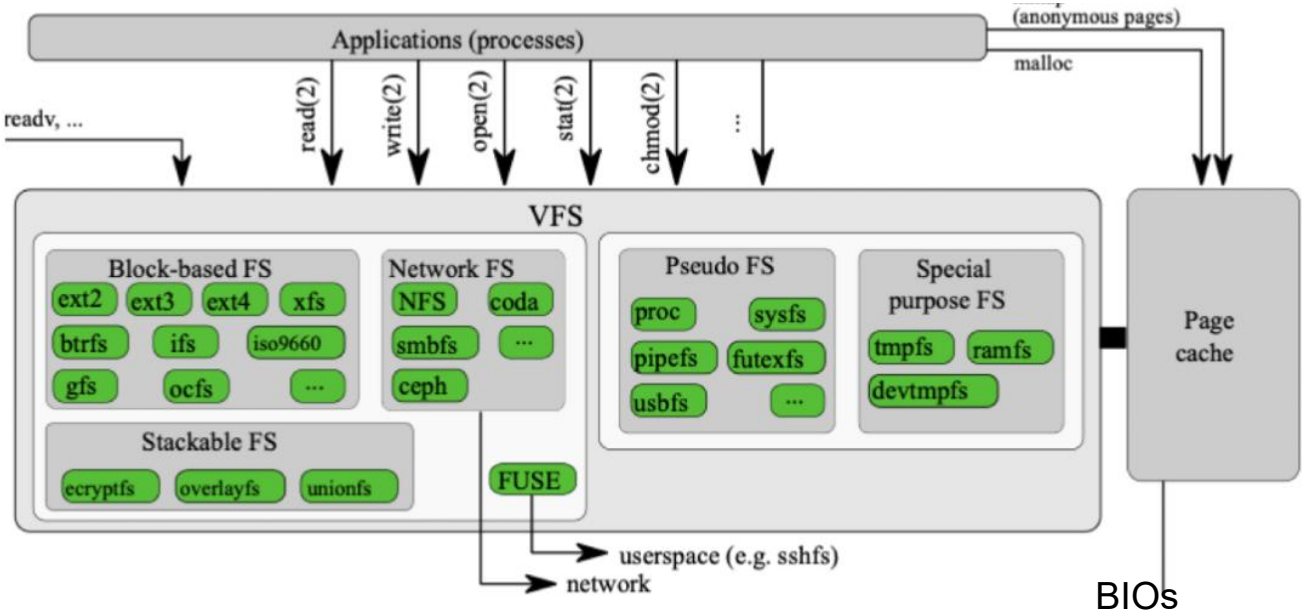


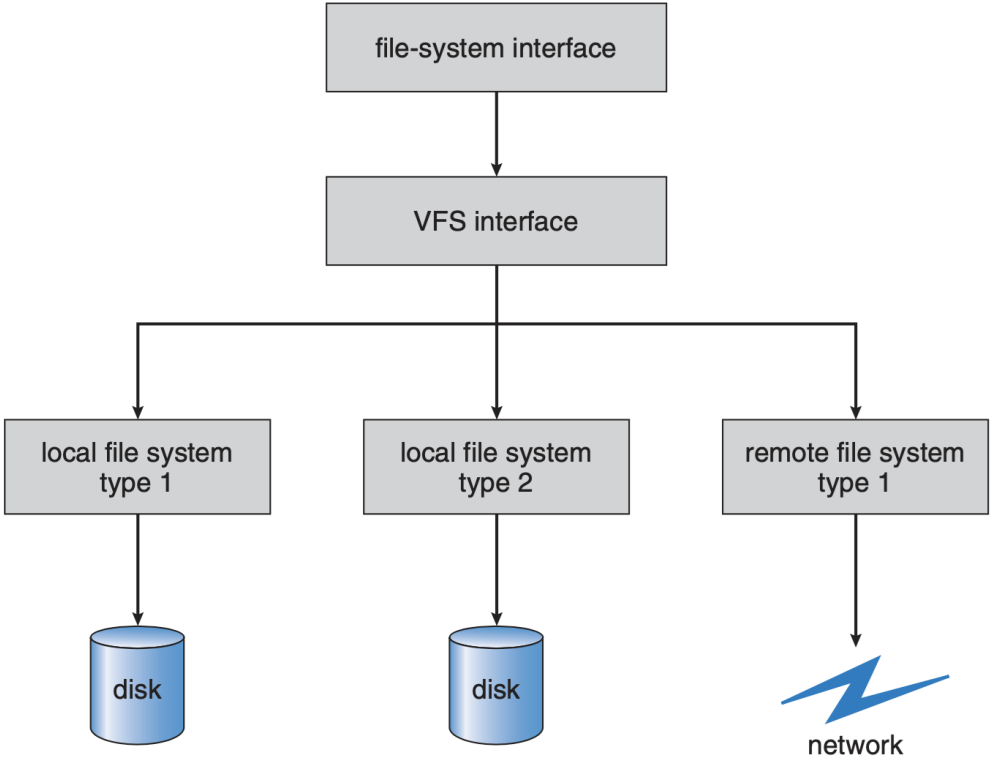
Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

Virtual File System (VFS)

Many file systems and device types can coexist on the same system.

Different levels of the stack have different interfaces:

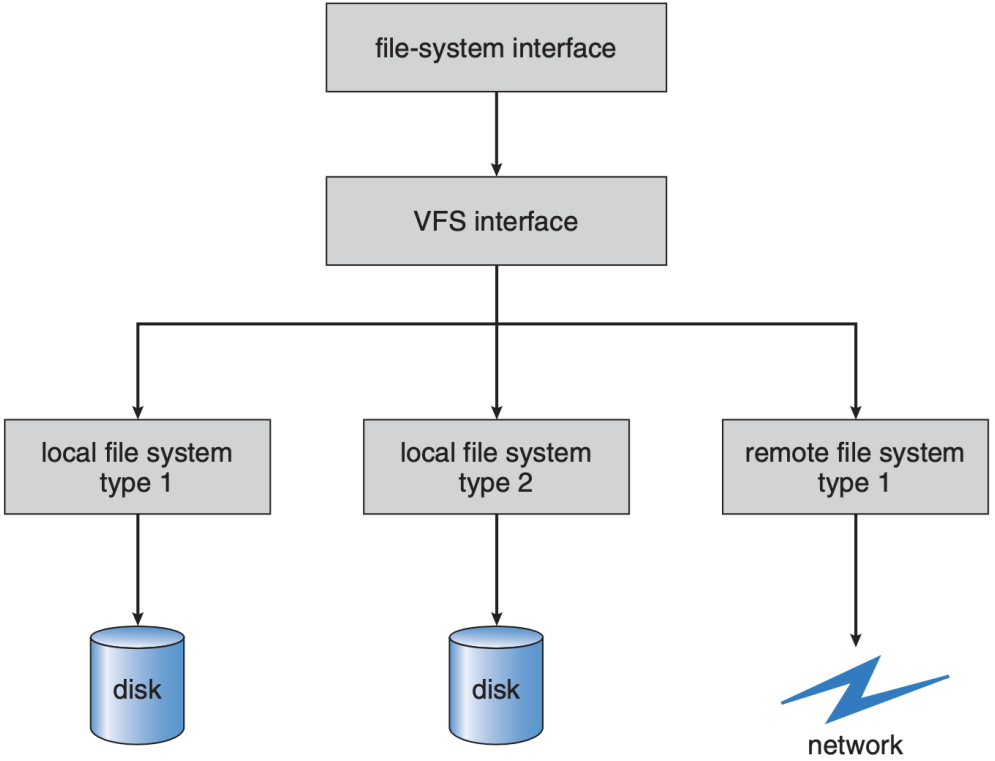
- File System Interface
- VFS Interface
- Storage Level



Virtual File System (VFS)

File System Interface:

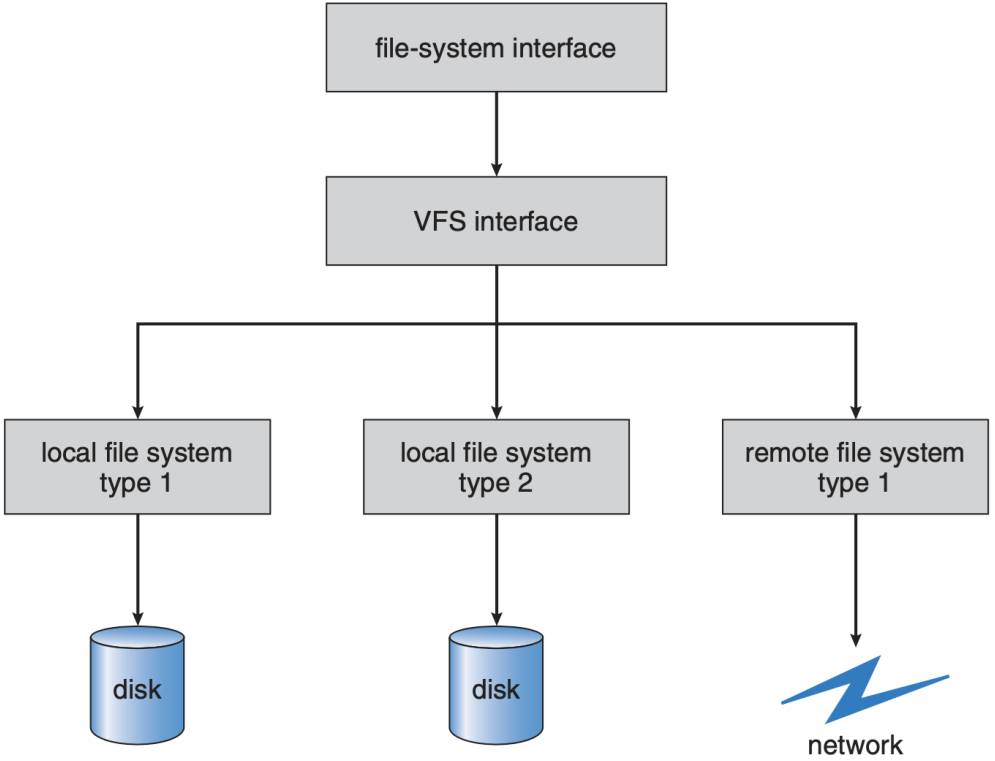
- API for userspace programs to interact with files
- `open()`, `close()`, `read()`, etc.
- Uses file descriptor to refer to a file
- Does not expose implementation details to the users



Virtual File System (VFS)

Storage Level:

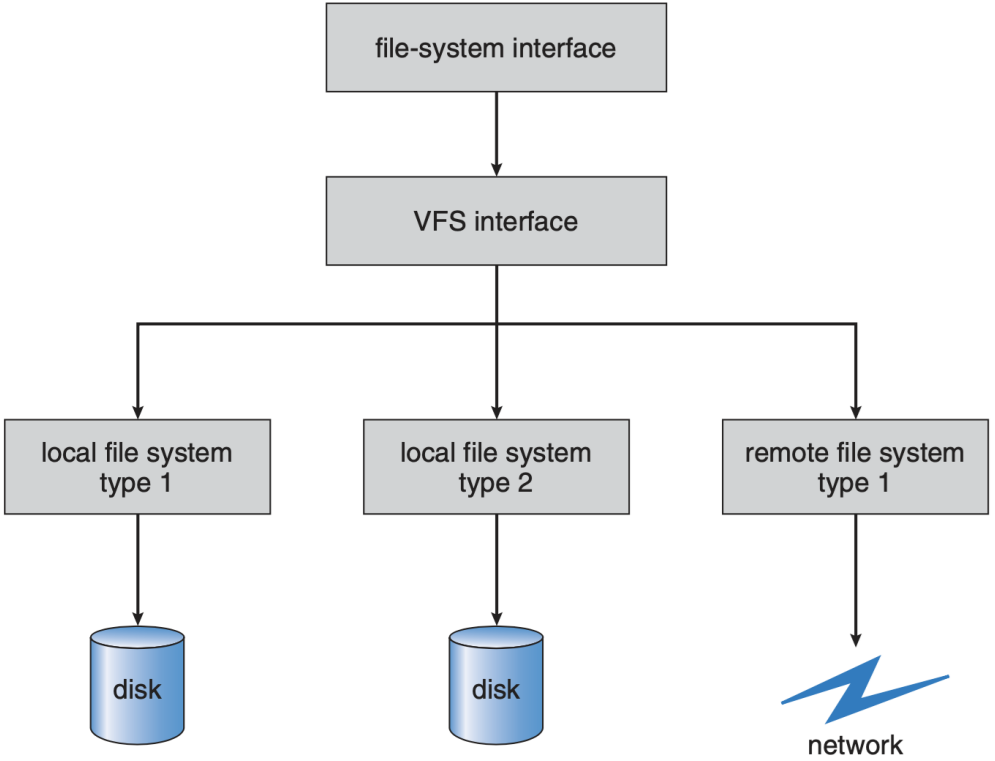
- Determines how data are stored in the disk
- Userspace programs are not burdened with these details
- Can even store data remotely, over the network



Virtual File System (VFS)

VFS Interface:

- Abstraction layer that can support multiple file systems
- Specifies an interface (similar to `struct sched_class`) that a given FS implements to hook into the kernel
- VFS dispatches operations to a specific FS using the interface, e.g.., `dir->inode_op->mkdir()`



Example: fs creation and mounting

- Create a new filesystem on a particular device:
  - Allocate inodes and blocks

```
# mkfs -t ext3 /dev/sda6
mke2fs 1.42 (29-Nov-2011)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
1120112 inodes, 4476416 blocks
223820 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=0
137 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

- Associating a device with a particular filesystem and providing an access point “mount point”

```
mount -t type device directory
```

```
~]# mount -t vfat /dev/sdc1 /media/flashdisk
```

Type	Description
ext2	The **ext2** file system.
ext3	The **ext3** file system.
ext4	The **ext4** file system.
iso9660	The **ISO 9660** file system. It is commonly used by optical media, typically CDs.
jfs	The **JFS** file system created by IBM.
nfs	The **NFS** file system. It is commonly used to access files over the network.
nfs4	The **NFSv4** file system. It is commonly used to access files over the network.
ntfs	The **NTFS** file system. It is commonly used on machines that are running the Windows operating system.
udf	The **UDF** file system. It is commonly used by optical media, typically DVDs.
vfat	The **FAT** file system. It is commonly used on machines that are running the Windows operating system, and on certain digital media such as USB flash drives or floppy disks.

“mount” Example

```
root@lnx1:~# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1986852k,nr_inodes=496713,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=403920k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
```

- Note the top entry “/” is mapped to disk
- At various mount points different filesystems are “attached”, e.g. all pseudo files systems or network file systems.

VFS Data Structures

`struct file`: Represents an instance of an open file

- Pointed to by per-process fdtable entry, allows for open file sharing by copying the pointer
- Stores flags, current position, etc.
- Refers to dentry via `struct path f_path` (which refers to the inode)

VFS interface: `struct file_operations *f_op`

- read, write, seek, etc.

VFS Data Structures

`struct dentry`: Basically a “hard link”: contains name of link and inode number

Break up an absolute path into dentries, one per component, e.g., `/home/frankeh/foo` has `/`, `home` , `frankeh`, `foo`

Path resolution is expensive to open `/home/frankeh/foo` you need to:

- consult the dentry for `/` to find the root inode
- find the root data block, iterate through it to find dentry for `home`
- consult the dentry for `home` to find the inode
- find the corresponding data block, iterate through it to find dentry for `frankeh`
- consult the dentry `frankeh` to find the inode
- find the corresponding data block, iterate through to find the dentry for `foo`
- consult the dentry `foo` to find the inode
- find the corresponding data block, and finally read the file contents!

VFS interface: `const struct dentry_operations *d_op`

- manage dentries through dentry cache (create/remove/hash/etc), more on this later

VFS Data Structures

`struct inode`: Unique descriptor of a file or directory

`i_ino`: inode # unique per “mounted” file system

Can refer to fs-specific data via `i_private` (will be used for HW-7)

VFS interface: `const struct inode_operations *i_op`

- read, write, seek, etc.

VFS Data Structures

`struct super_block`: Descriptor of a mounted filesystem.

VFS interface: `const struct super_operations *s_op`

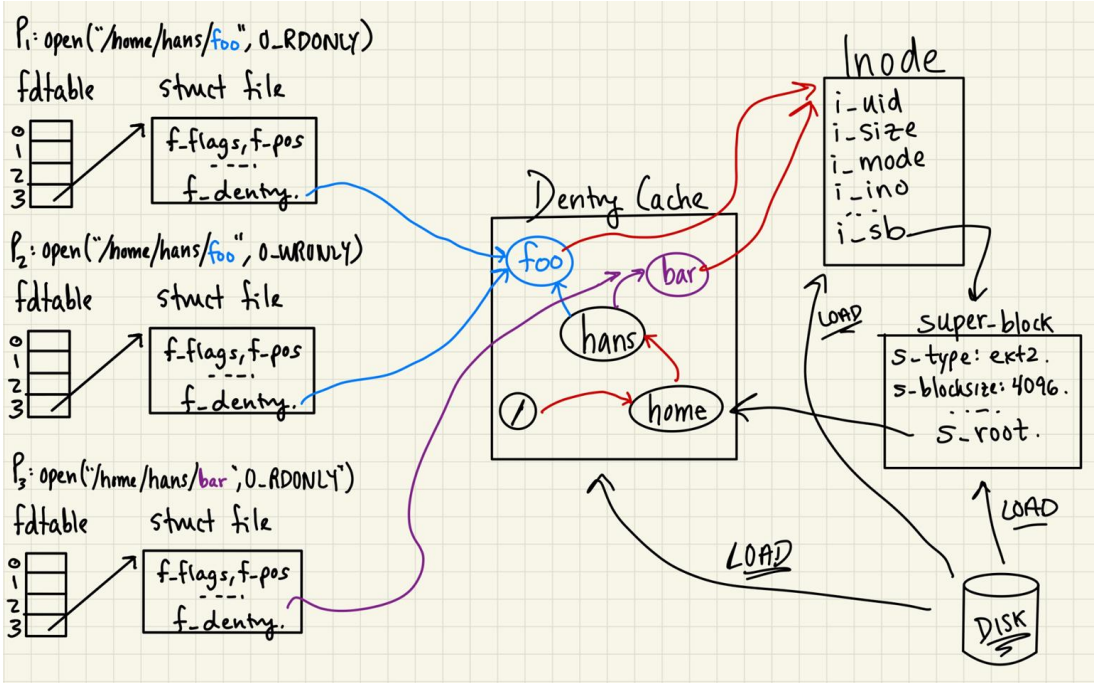
- inode management, journaling, syncing metadata

Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

1. Mount an instance of ext2 at `/home`

`s_root` field of `super_block` refers to the root dentry of the mount



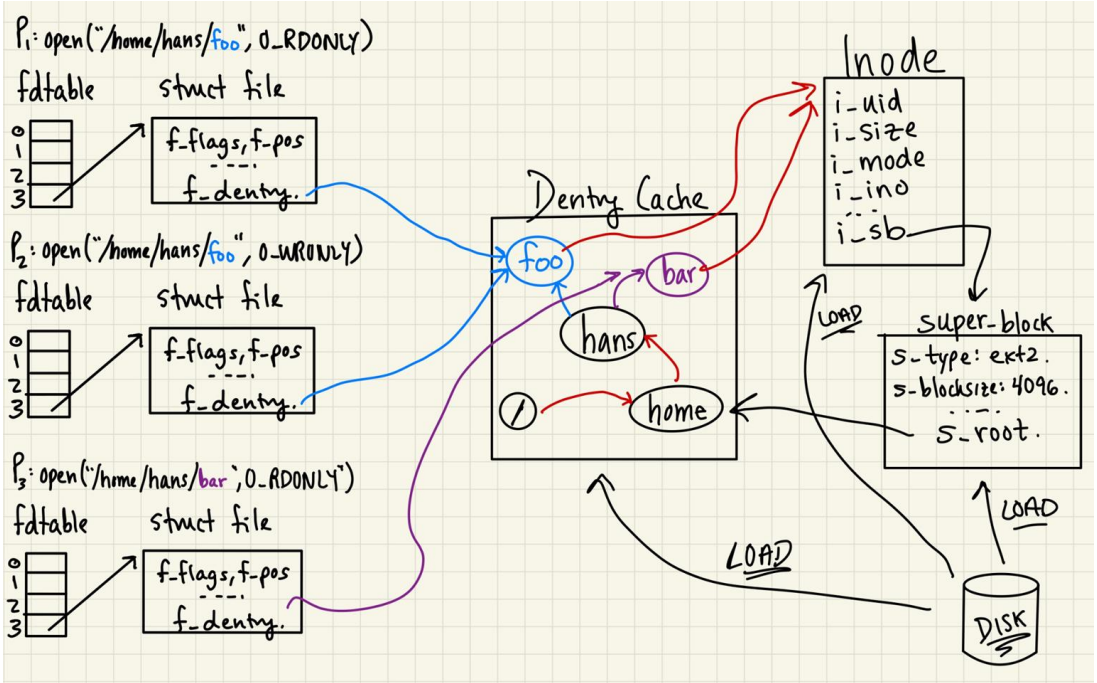
Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

2. P1 opens `/home/hans/foo` for reading

Need to read several inodes/dentries from disk

Along the way, cache them in the dcache



Dentry Cache

Linux kernel makes path resolution efficient by employing a dentry cache (dcache)

3. P3 opens `/home/hans/bar`

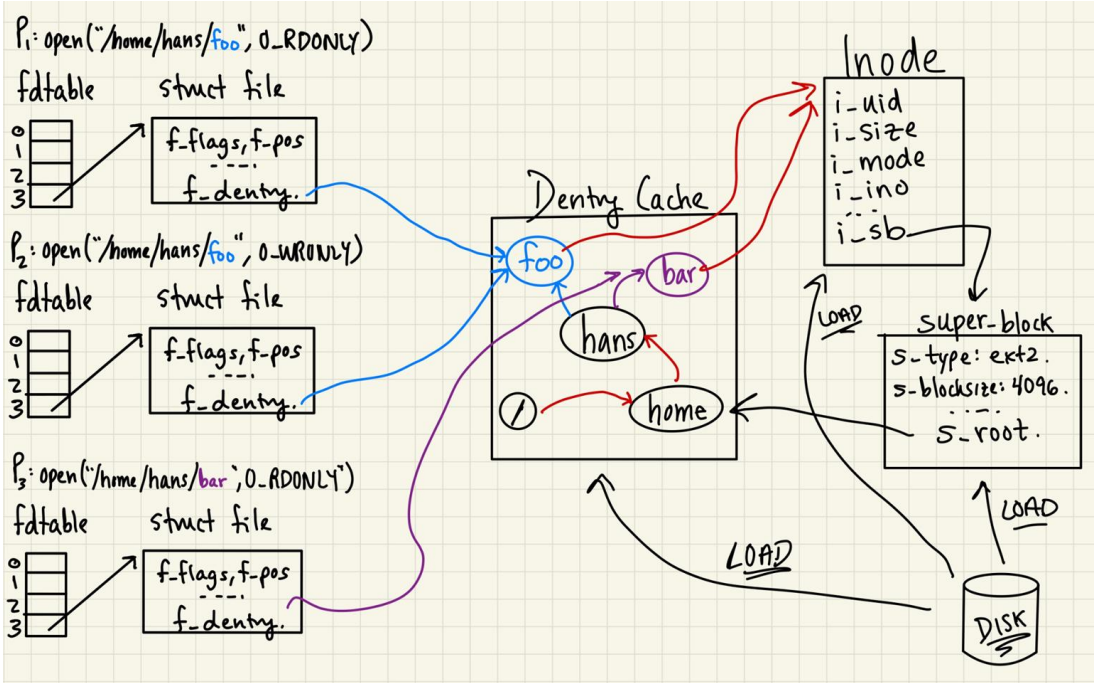
Different file than P1 and P2

`/home/hans/` path resolution cached in dcache

Need to read in `hans/` directory data block to find dentry for `bar`

…only to find it refers to the same inode as `foo`

`bar` and `foo` are hard links to the same inode!



Network File System

- Files are located on a different server
  - CIFS, NFS, ….
- Requests are sent over to server with name and block requests
- Data can be cached, but be aware of sharing

Pseudo FileSystem

- /proc or /sys
- Means to access/manipulate OS internals
  (state, parameters, algos, settings)
- Allows scripting (vs. a special syscall interface)
- The hierarchy is created dynamically and on access, there is no real disk !

Proc fs (inspecting state)

- Inspecting a single process or system statistics
- Registers read/write functions per “node”

```
frankeh@lnx1:~$ ps -edf | grep 1692
frankeh   1692  1509  0 Apr27 ?        00:00:19 x-terminal-emulator
```

```
frankeh@lnx1:~$ ls /proc/1692/
attr        coredump_filter  gid_map    mountinfo   oom_score      sched         stat           uid_map
autogroup   cpuset           io         mounts      oom_score_adj  schedstat     statm          wchan
auxv        cwd              limits     mountstats  pagemap        sessionid     status
cgroup      environ          loginuid   net         patch_state    setgroups     syscall
clear_refs  exe              map_files  ns          personality    smaps         task
cmdline     fd               maps       numa_maps   projid_map     smaps_rollup  timers
comm        fdinfo           mem        oom_adj     root           stack         timerslack_ns
```

```
frankeh@lnx1:/proc/1692$ cat stat
1692 (x-terminal-emul) S 1509 1397 1397 0 -1 4194304 13687 13059 12 49 1472 604 61 132 20 0 3 0 3444 524214272 10142
18446744073709551615 94007262363648 94007262436208 140727902463216 0 0 0 0 4096 65536 0 0 0 17 2 0 0 4 0 0 9400726453
5656 94007264542176 94007273050112 140727902464728 140727902464748 140727902464748 140727902466011 0
frankeh@lnx1:/proc/1692$ cat statm
127982 10142 5828 18 0 12981 0
```

```
frankeh@lnx1:~$ cat /proc/1692/maps | head
557fc57ac000-557fc57be000 r-xp 00000000 08:01 1194341                    /usr/bin/lxterminal
557fc59be000-557fc59bf000 r--p 00012000 08:01 1194341                    /usr/bin/lxterminal
557fc59bf000-557fc59c0000 rw-p 00013000 08:01 1194341                    /usr/bin/lxterminal
557fc61dd000-557fc72da000 rw-p 00000000 00:00 0                          [heap]
7fc988000000-7fc988021000 rw-p 00000000 00:00 0
7fc988021000-7fc98c000000 ---p 00000000 00:00 0
7fc98c000000-7fc98c021000 rw-p 00000000 00:00 0
```

Sys-fs

- Read and modify system status and behavior

```
frankeh@lnx1:/sys$ ls
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module  power
frankeh@lnx1:/sys$ cd bus
frankeh@lnx1:/sys/bus$ ls
ac97         container     gpio  iscsi_flashnode  mipi-dsi  nvmem        platform  sdio    usb        xen
acpi         cpu           hid   machinecheck     mmc       pci          pnp       serial  virtio     xen-backend
clockevents  edac          i2c   mdio_bus         nd        pci-epf      rapidio   serio   vme
clocksource  event_source  isa   memory           node      pci_express  scsi      spi     workqueue
frankeh@lnx1:/sys/bus$ cd pci
frankeh@lnx1:/sys/bus/pci$ ls
devices  drivers  drivers_autoprobe  drivers_probe  rescan  resource_alignment  slots  uevent
frankeh@lnx1:/sys/bus/pci$ ls devices/
0000:00:00.0  0000:00:01.1  0000:00:03.0  0000:00:05.0  0000:00:07.0  0000:00:0d.0
0000:00:01.0  0000:00:02.0  0000:00:04.0  0000:00:06.0  0000:00:08.0
```

- Example: change the scheduling algorithm for a particular disk device (/dev/sda):

```
root@lnx1:~# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
root@lnx1:~# echo noop > /sys/block/sda/queue/scheduler
root@lnx1:~# cat /sys/block/sda/queue/scheduler
[noop] deadline cfq
```

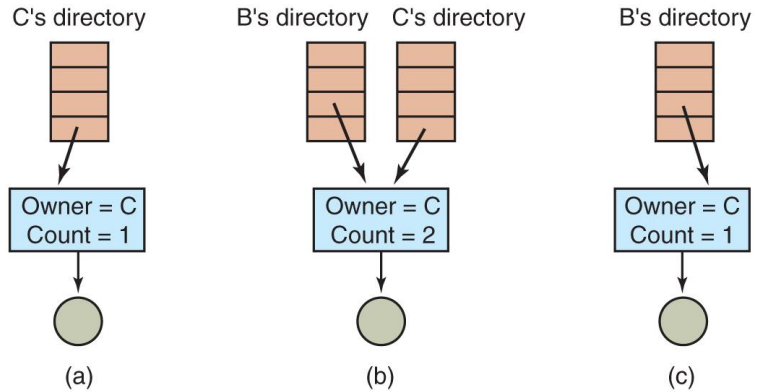
Special Purpose Filesystem

- RamFS (builds a ram disk)
- Backed by memory not disk
- Instead of issuing block I/O requests you memcpy (4K) to/from memory
- Obviously
  - significantly faster than going to any real device
  - But no persistence
  - Reduces available RAM on your system
- Linux : tmpfs

Shared Files: Method 1

- Disk blocks are not listed in directories but in a data structure associated with the file itself (e.g. i-nodes in UNIX).
- Directories just point to that data structure.
- This approach is called: static linking
- "`ln <origfile> <newfilename>`" or "`ln -P`"

Problematic Scenario



Shared Files: Method 2

- Have the system create a new file (of type LINK). This new file contains the path name of the file to which it is linked.
- This approach is called: symbolic linking
- The main drawback is the extra overhead.
- "`ln -s <origfile> <newfilename>`"

* DEMO *