

Lecture 16:

Filesystem Implementations

CS343 – Operating Systems
Branden Ghena – Fall 2024

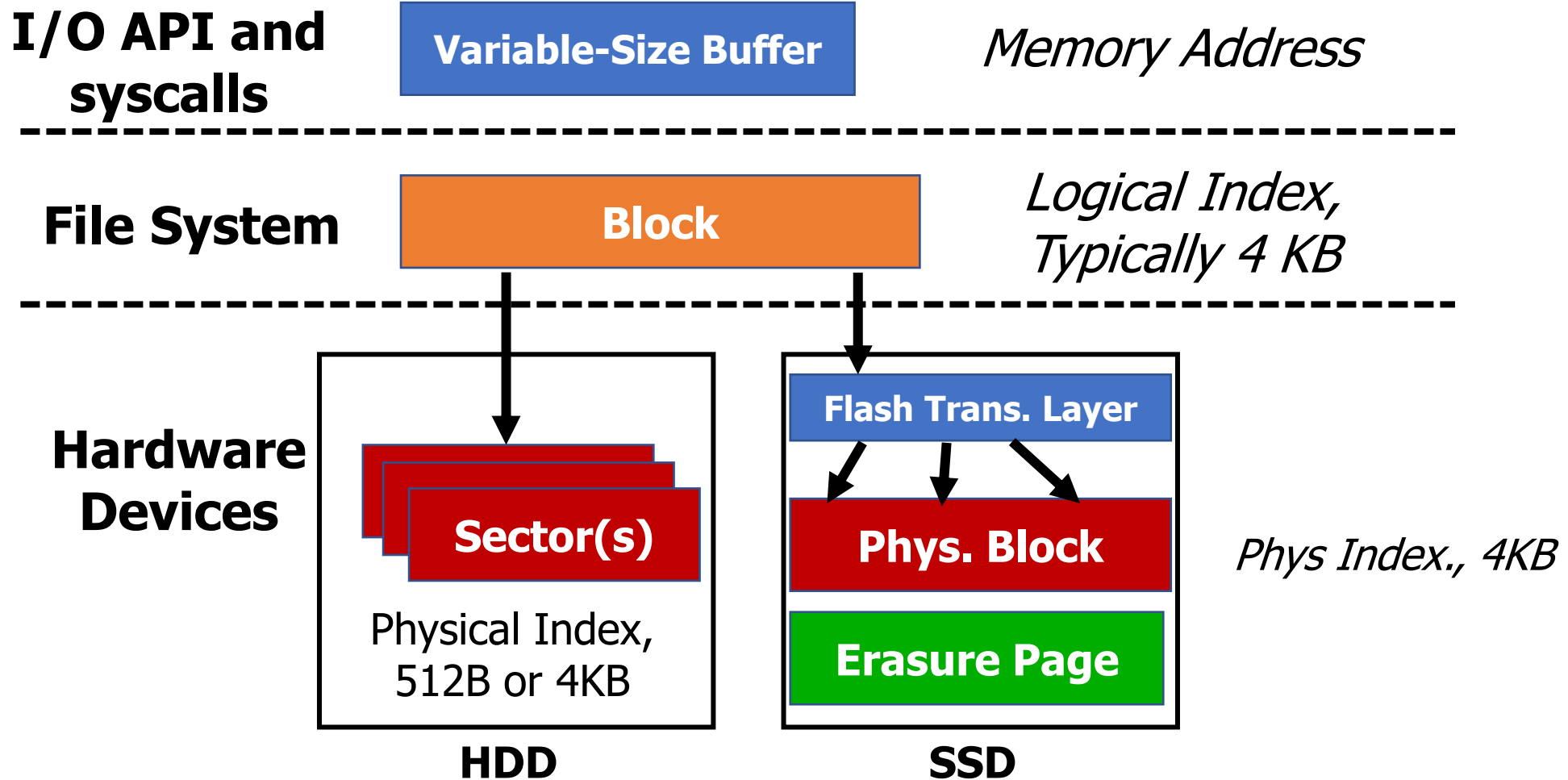
Some slides borrowed from:

Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), Ed Lazowska (Washington), and UC Berkeley CS162

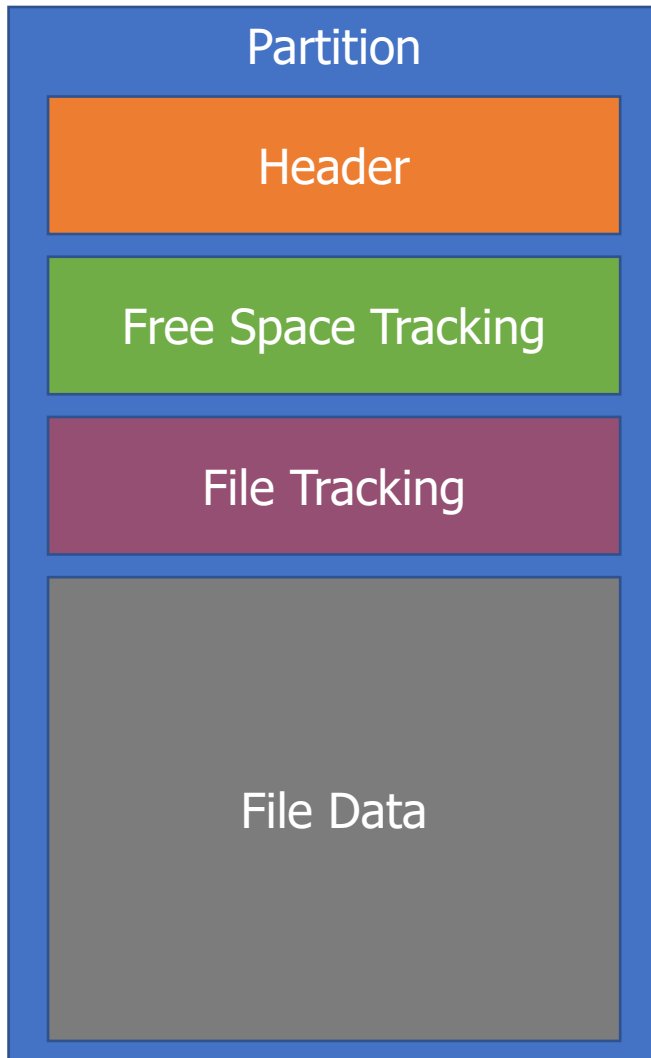
Today's Goals

- Understand about additional filesystem features
 - Performance: disk caching
 - Reliability: checking, journaling, and copy-on-write
- Explore real-world filesystem designs
 - FAT, FFS, ext3/ext4, NTFS, ZFS

File systems abstractions



What goes within a partition?

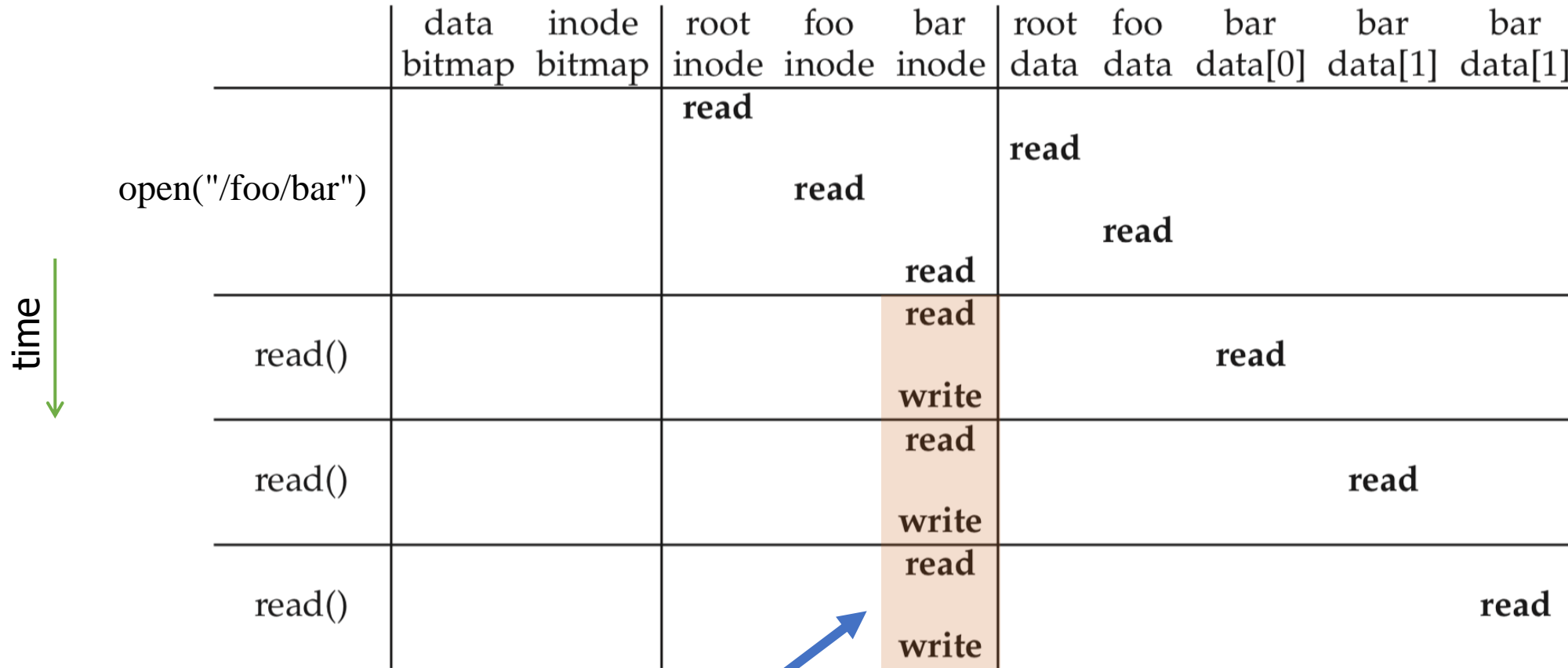


- Header (Superblock)
 - Details about which filesystem this is
 - Metadata about the filesystem
- Free Space Tracking
 - Likely a bitmap of whether blocks are used/free
- File Tracking
 - Either allocation table or inodes
- File Data

Outline

- **Disk Caching**
- Classical Filesystems
 - FAT
 - FFS
- Improving Reliability
 - FSCK
 - Journaling
- Journaling Filesystems
 - ext3/ext4
 - NTFS
- Copy-On-Write
 - ZFS

Many disk interactions should be hitting memory instead

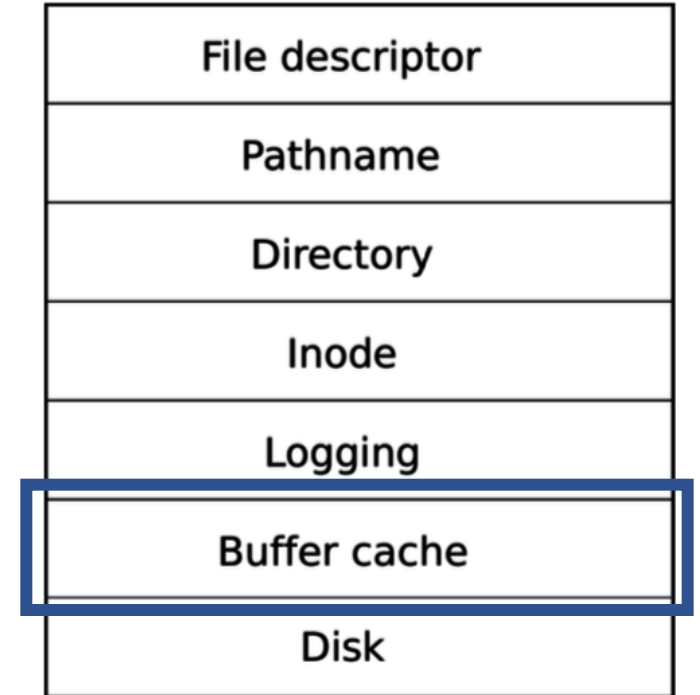


| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|------------------|----------------|-----------------|---------------|--------------|--------------|--------------|-------------|----------------|----------------|----------------|
| open("/foo/bar") | | | read | read | read | read | read | | | |
| read() | | | | | read | | read | | | |
| read() | | | | | write | | | | | |
| read() | | | | | read | | | | read | |
| read() | | | | | write | | | | | |
| read() | | | | | read | | | | | read |
| | | | | | write | | | | | |

inode reads/writes
occur in memory

Filesystem caching

- File I/O can be a significant bottleneck
- So keep useful parts of disk in RAM!
 - Improves performance
- OS kernel does this automatically
 - Using unused RAM to hold disk blocks



Goals for filesystem caching

1. Cache popular blocks so the disk can be accessed less frequently.
 - Recall that disk has $10,000\times$ greater delay than RAM.
 - ***Reads*** are faster if the disk block is already in memory from a recent access.
 - ***Writes*** can be aggregated.
 - If a thread writes three times briefly to the same file, these can likely be reduced to one write to disk if the writes are delayed.
 - If a thread creates a new file and quickly deletes it, these writes can be skipped altogether.
 - Eventually, changes must be flushed to disk, but there is no rush.
2. Must be careful to prevent two threads from accessing different unsynchronized copies of the disk block.
 - i.e., make the cache **coherent** and avoid race conditions

Unified Page Cache

- Page replacement policy can simultaneously consider both pages from Virtual Memory and pages cached from disk
 - May choose to evict either if needed
- Priority:
 1. Unwritten disk files or unmodified anonymous memory pages
 - Situational which is more important, but neither requires writeback
 2. Written disk files
 - Going to have to be written to disk eventually anyways
 3. Modified memory pages
 - Must go to swap space to be later read again

Prefetching

- Any cache can “prefetch”, loading memory *before* it’s needed
- Base idea: read multiple blocks from disk sequentially from each access
- Advanced: load specific files based on usage patterns
 - The user always opens Powerpoint, so load its data onto disk at boot
- Need to balance prefetching requests with other disk access
 - Don’t want to slow down real accesses with possibly needed prefetching

Short break + Question

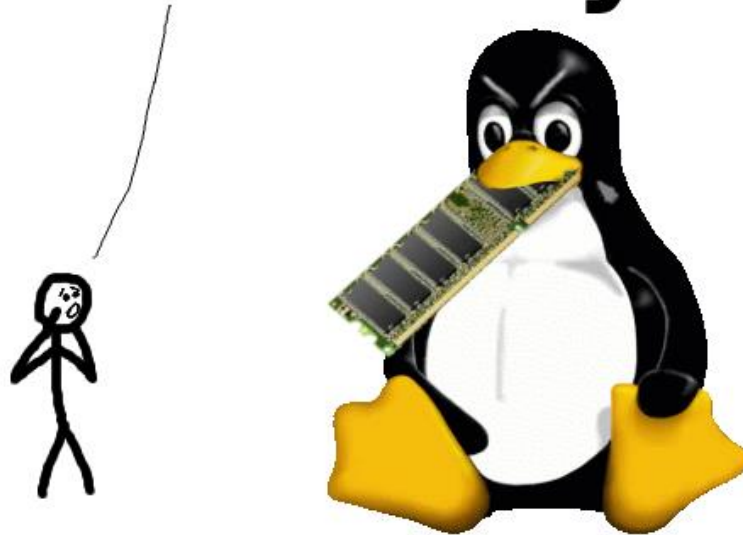
- What percentage of memory should an OS fill with disk pages?

Short break + Question

- What percentage of memory should an OS fill with disk pages?
 - As long as it can do it in the background, as much as possible!
 - There's no particular downside:
 - As long as the page wasn't written to, the RAM can be repurposed later if needed without requiring additional writes to disk
 - (Maybe energy use is a downside?)

Real OSes aggressively cache disk in unused RAM

Linux ate my ram!



Don't Panic!
Your ram is fine!

linuxatemyram.com

Real OSes aggressively cache disk in unused RAM

```
top - 10:25:45 up 7 days, 48 min, 3 users, load average: 0.04, 0.06, 0.09
Tasks: 650 total, 1 running, 649 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.0%sy, 0.0%ni, 99.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 132144848k total, 129331984k used, 2812864k free, 37895660k buffers
Swap: 16383996k total, 436k used, 16383560k free, 45074412k cached
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|-----------|----|----|-------|------|------|---|------|------|----------|-----------------|
| 9213 | mysql | 20 | 0 | 1263m | 156m | 14m | S | 0.0 | 0.1 | 3:57.24 | mysqld |
| 10001 | root | 20 | 0 | 5748m | 219m | 14m | S | 0.3 | 0.2 | 15:02.22 | dsm_om_connsvc |
| 9382 | root | 20 | 0 | 337m | 18m | 11m | S | 0.0 | 0.0 | 0:10.67 | httpd |
| 8304 | apache | 20 | 0 | 352m | 19m | 10m | S | 0.0 | 0.0 | 0:00.29 | httpd |
| 8302 | apache | 20 | 0 | 339m | 14m | 7144 | S | 0.0 | 0.0 | 0:00.16 | httpd |
| 8298 | apache | 20 | 0 | 339m | 14m | 7140 | S | 0.0 | 0.0 | 0:00.12 | httpd |
| 8299 | apache | 20 | 0 | 339m | 14m | 7136 | S | 0.0 | 0.0 | 0:00.17 | httpd |
| 8303 | apache | 20 | 0 | 339m | 14m | 7136 | S | 0.0 | 0.0 | 0:00.17 | httpd |
| 8300 | apache | 20 | 0 | 339m | 14m | 7120 | S | 0.0 | 0.0 | 0:00.13 | httpd |
| 8301 | apache | 20 | 0 | 339m | 14m | 7120 | S | 0.0 | 0.0 | 0:00.16 | httpd |
| 8305 | apache | 20 | 0 | 339m | 14m | 7112 | S | 0.0 | 0.0 | 0:00.13 | httpd |
| 1386 | apache | 20 | 0 | 339m | 14m | 7096 | S | 0.0 | 0.0 | 0:00.06 | httpd |
| 1387 | apache | 20 | 0 | 339m | 14m | 7084 | S | 0.0 | 0.0 | 0:00.07 | httpd |
| 1122 | spt175 | 20 | 0 | 251m | 14m | 6484 | S | 0.0 | 0.0 | 0:00.26 | emacs |
| 2615 | root | 20 | 0 | 92996 | 6200 | 4816 | S | 0.0 | 0.0 | 0:00.93 | NetworkManager |
| 9865 | root | 20 | 0 | 1043m | 23m | 4680 | S | 0.3 | 0.0 | 9:44.98 | dsm_sa_datamgrd |
| 8737 | postgres | 20 | 0 | 219m | 5380 | 4588 | S | 0.0 | 0.0 | 0:01.00 | postmaster |
| 2786 | haldaemon | 20 | 0 | 45448 | 5528 | 4320 | S | 0.0 | 0.0 | 0:03.99 | halld |
| 9956 | root | 20 | 0 | 491m | 7268 | 3280 | S | 0.0 | 0.0 | 3:16.30 | dsm_sa_snmpd |
| 990 | root | 20 | 0 | 103m | 4188 | 3172 | S | 0.0 | 0.0 | 0:00.01 | sshd |
| 1014 | root | 20 | 0 | 103m | 4196 | 3172 | S | 0.0 | 0.0 | 0:00.02 | sshd |
| 19701 | root | 20 | 0 | 103m | 4244 | 3172 | S | 0.0 | 0.0 | 0:00.01 | sshd |

- *buffers* and *cached* both represent file data that is being stored in memory for improved performance
 - Still available for programs
 - Just being made useful for now by caching disk
- Might be a lot of RAM's use for big systems
 - Total RAM: 128 GB
 - Disk cache: 83 GB

Outline

- Disk Caching
- **Classical Filesystems**
 - **FAT**
 - **FFS**
- Improving Reliability
 - FSCK
 - Journaling
- Journaling Filesystems
 - ext3/ext4
 - NTFS
- Copy-On-Write
 - ZFS

FAT (FAT/FAT12/FAT16/FAT32)

- File Allocation Table
- FAT: Microsoft system from *before* MS-DOS (1977)
 - 8 MB max file size
 - 9 character file names
 - No subdirectories
- FAT32: Windows 2000 (introduced 1996)
 - 2 GB max file size
 - 255 character file names
 - Supports up to 16 TB partitions

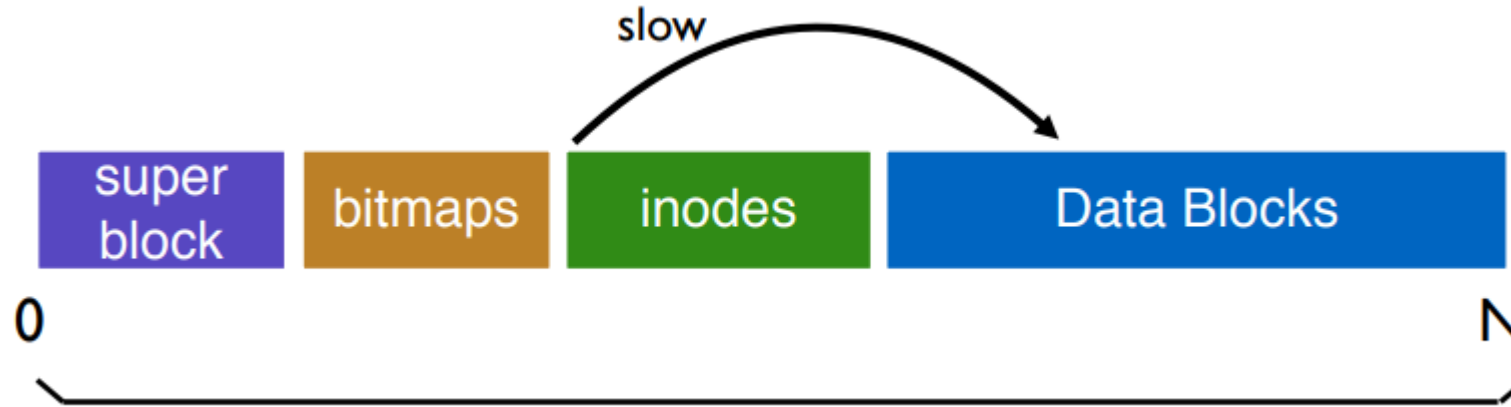
FAT design choices

- Allocation table for tracking data blocks
 - Requires four bytes per block in the disk
 - File attributes need to be kept in the directory data block
- Still in use for embedded systems
 - Simple to implement
 - Still compatible with modern general-purpose OSes
 - Works fine for relatively small disks with correspondingly small files
 - Think SD cards
 - Implements aggressive block caching

Fast File System (FFS)

- Unix FileSystem (FS) from 1970
 - inode-based design (combination of all the basic stuff covered last time)
 - Simple and slow
 - inodes are far from data blocks (reads tend to jump back-and-forth)
 - data blocks become fragmented over time
- BSD Fast File System (mid-1980s)
 - First “Disk aware file system”
 - Understands disk seek patterns and sequential access benefits

FFS groups



- Split disk space into a set of “cylinder groups”
 - Each group has its own bitmaps, inodes, and data
 - Keeps data and inodes closer together



FFS file placement strategy

- General theme: put related pieces of data near each other
- Rules
 1. Put directory data near directory inodes
 2. Put file inodes near directory data
 3. Put data blocks near file inodes
- Example
 - Each directory gets put in an empty group
 - Keep all files within a directory in that single group

FFS example

- Example:
 - Directories: **/**, **/a/**, and **/b/**
 - /a/ files: **c**, **d**, **e**
 - /b/ files: **f**

| group | inodes | data |
|-------|-----------|------------|
| 0 | /----- | /----- |
| 1 | acde----- | accddee--- |
| 2 | bf----- | bff----- |
| 3 | ----- | ----- |
| 4 | ----- | ----- |
| 5 | ----- | ----- |
| 6 | ----- | ----- |
| 7 | ----- | ----- |
| ... | | |

FFS large file problem

- A single large file can fill nearly all of a group
 - So remaining files would have to be placed in other groups

| group | inodes | data |
|-------|---------|---|
| 0 | /a----- | /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a----- |
| 1 | ----- | ----- |
| 2 | ----- | ----- |
| ... | | |

- Instead, limit filesize per group and place remaining blocks in other groups

| group | inodes | data |
|-------|---------|-------------|
| 0 | /a----- | /aaaaa----- |
| 1 | ----- | aaaaa----- |
| 2 | ----- | aaaaa----- |
| 3 | ----- | aaaaa----- |
| 4 | ----- | aaaaa----- |
| 5 | ----- | aaaaa----- |
| 6 | ----- | ----- |
| ... | | |

- Most files are small so prioritize them
- Rare, large files will have worse performance

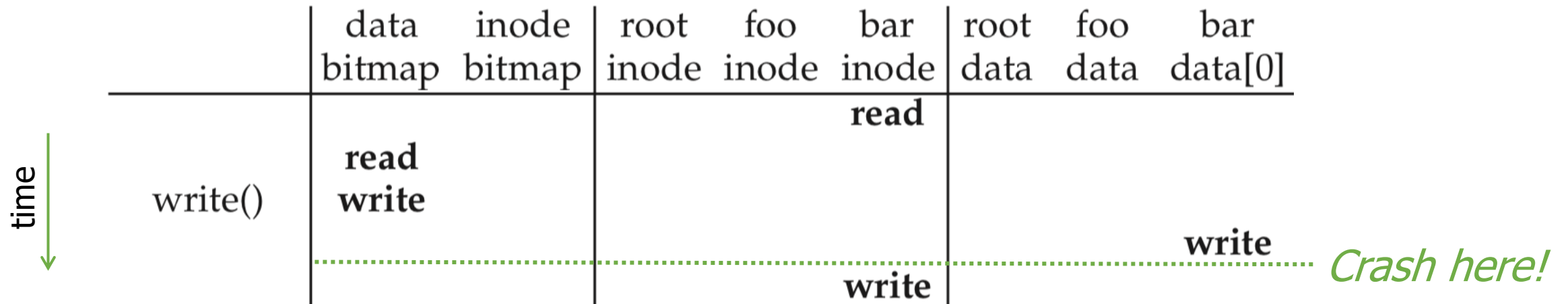
Outline

- Disk Caching
- Classical Filesystems
 - FAT
 - FFS
- **Improving Reliability**
 - **FSCK**
 - **Journaling**
- Journaling Filesystems
 - ext3/ext4
 - NTFS
- Copy-On-Write
 - ZFS

Crash tolerance

- Filesystems are persistent and store important data
- They *cannot* rely on a graceful shutdown
 - Power outages happen
 - Kernel might panic
 - USB plug might be yanked out
- File system data structure updates are *critical sections*
 - Not concerned about race conditions, but rather partial updates
 - Transactions should be performed atomically, “all or none”
- All reads and writes aren’t necessarily guaranteed
 - But system needs to stay **consistent**

Crash example (writing to /foo/bar)



- Crash before write to file's inode could leak a data block
 - Data bitmap was updated to reserve data block and data was written
 - But the data block is not pointed to by any inode
 - Block ends up wasted
- Other write orders would be even worse
 - Inode points to a block that hasn't been written and has garbage data
 - Or block is still marked as free in the bitmap, and another file will overwrite it!!

File system checker (FSCK)

- After a crash, scan entire disk for contradictions and “fix”
 - System pauses boot until FSCK completes
- Example: check data bitmap consistency
 - Read every valid inode
 - Any referenced data block should be marked as used
 - Any used blocks that are not referenced can be marked free
- Also check
 - Each inode should only be listed under one directory (without hard links)
 - Two inodes should not share a data block
 - All block addresses should be valid

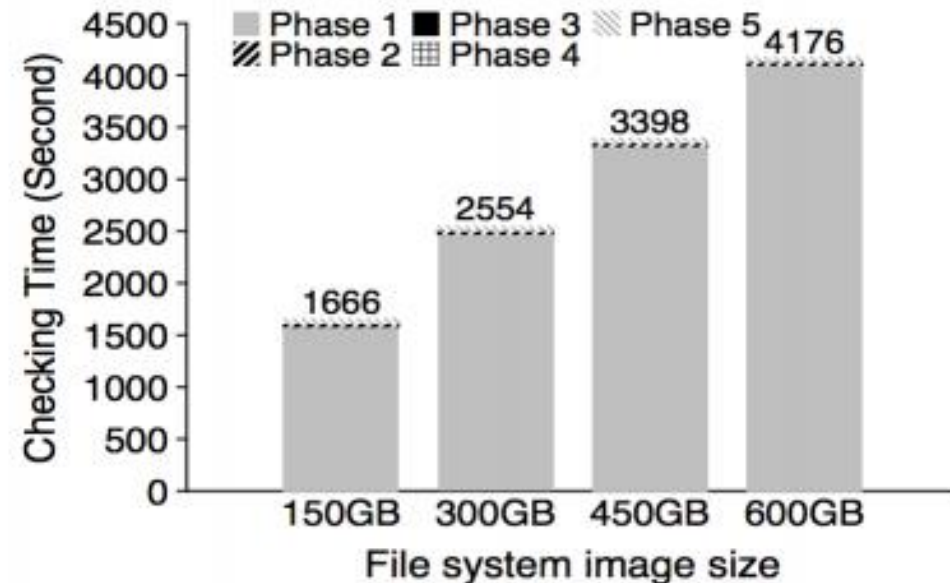
Problems with FSCK

1. FSCK makes disks *consistent*, not *correct*

- Not always obvious how best to fix file system image
- Trivial way to get consistency: reformat disk

2. FSCK is very slow

- Reading from disk is slow
- Reading ALL of disk takes a long time, especially as disks increase in size



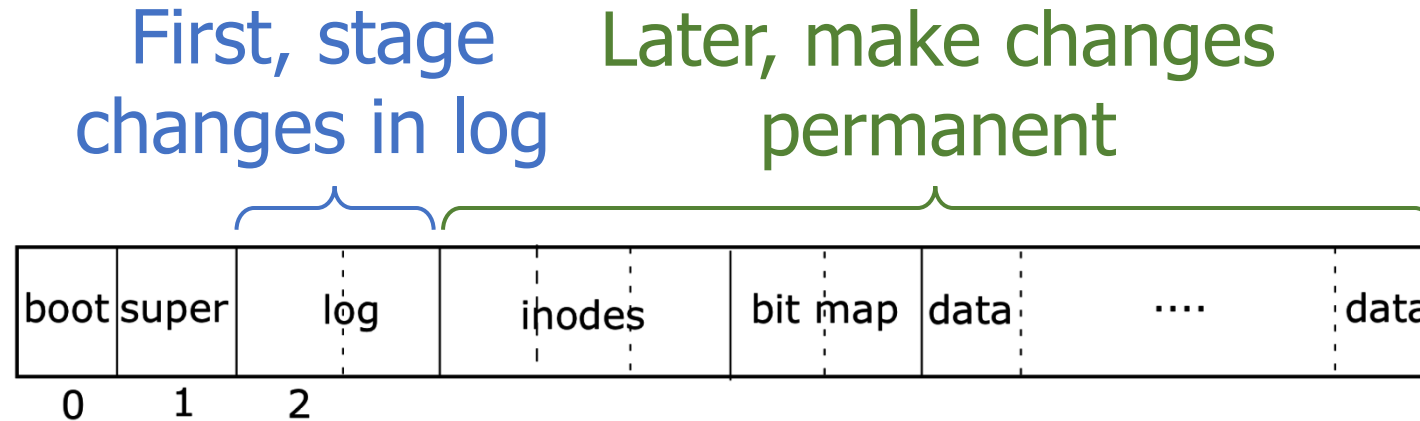
Checking a 600GB disk takes ~70 minutes

Filesystem transactions

- Goals
 - Move reliability mechanism to continuous operations during runtime
 - Some recovery after crash is fine, but not entire disk
 - Don't just make file system consistent
 - Guarantee correctness
- Solution: enforce atomic transactions
 - Each transaction must be performed in its entirety or not at all
 - Either all new data is visible
 - Or all old data is visible

Journaling Filesystems


- Write all transactions to *journal* instead of actual locations



1. Write the blocks to the log, a reserved part of the disk.
 - This makes a durable record of the transaction you plan to commit.
 - Continue putting all writes to the log, until ***commit*** is called.
2. On commit, write a commit message to the log, then start writing all of the logged writes where they belong on disk.
 - Clear the log after everything is written again.

Journaling example

Journal




| |
|---|
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | B | C | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Current contents of 8 blocks of disk and the journal
 - Note that the journal is also on disk
- Keeping this abstract
 - Blocks could be bitmaps, inodes, data, or anything

Journaling example

Journal




| Transaction Begin |
|-------------------|
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0 | 0 | B | C | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Write transaction start to journal

Journaling example

Journal




| |
|------------------------|
| Transaction Begin |
| Write Block 6, Data: Y |
| Write Block 7, Data: Z |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | B | C | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Write transaction start to journal
- Then actions for that transaction
 - Along with the data
 - Journal must be multiple blocks in size

Journaling example

Journal




| |
|--------------------------|
| Transaction Begin |
| Write Block 6, Data: Y |
| Write Block 7, Data: Z |
| Transaction End |
| 0 |
| 0 |

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0 | 0 | B | C | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Write transaction start to journal
- Then actions for that transaction
 - Along with the data
 - Journal must be multiple blocks in size
- “Commit” by writing transaction end

Journaling example

Journal




| |
|--------------------------|
| Transaction Begin |
| Write Block 6, Data: Y |
| Write Block 7, Data: Z |
| Transaction End |
| 0 |
| 0 |

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0 | 0 | B | C | 0 | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- Sometime after transaction is written, data can actually be recorded to disk

Journaling example

Journal



| |
|---|
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | B | C | 0 | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |


- Sometime after transaction is written, data can actually be recorded to disk
- And then journal can be cleared

Resolving crashes with journaling

- The next time the computer boots, OS resolves filesystem:
 1. No transactions happening when crash occurred
 - Journal is empty. Do nothing because there were no outstanding transactions.
 2. Crash occurred *before commit* (before Transaction End):
 - There is data in the journal, but no commit message.
 - Just clear the log to **roll back** the transaction.
 3. Crash occurred *after commit*, while writing data to main part of disk.
 - We don't know how much of the transaction was finished.
 - However, the journal tells us exactly what must be done!
 - **Replay** the transaction (from the beginning), then clear the journal.

Break + Check your understanding – resolve after crash

Journal




| |
|--------------------------|
| Transaction Begin |
| Write Block 0, Data: Q |
| Write Block 1, Data: R |
| Write Block 2, Data: S |
| Transaction End |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q | R | 0 | B | C | 0 | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- When did this crash occur?
- What steps should be taken?

Break + Check your understanding – resolve after crash

Journal




| |
|--------------------------|
| Transaction Begin |
| Write Block 0, Data: Q |
| Write Block 1, Data: R |
| Write Block 2, Data: S |
| Transaction End |
| 0 |

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| Q | R | S | B | C | O | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- When did this crash occur?
 - After commit
 - Some data may have even been written (impossible to know)
Note: only look at the journal
- What steps should be taken?
 - Replay transaction and perform the writes

Break + Check your understanding – resolve after crash again

Journal




| |
|------------------------|
| Transaction Begin |
| Write Block 3, Data: B |
| Write Block 4, Data: C |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q | R | 0 | B | F | 0 | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- When did this crash occur?
- What steps should be taken?

Break + Check your understanding – resolve after crash again

Journal



| |
|---|
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q | R | 0 | B | F | 0 | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

- When did this crash occur?
 - Before transaction committed
- What steps should be taken?
 - Delete partial transaction from journal
 - No need to edit disk blocks

Journaling performance

- Transactions only need to be written to the journal for writes
- Interactions with disk can still be cached as before
 - Would be lost in a crash, but no consistency problems
 - Several writes can be combined into one transaction
- Can avoid writing all disk blocks twice by only tracking metadata
 - Writes to bitmaps, inodes, and directories are journaled
 - Writes to file data blocks just happen whenever
 - File could still be corrupted! But the *filesystem* is safe
 - Likely only corrupted in units of whole blocks

Outline

- Disk Caching
- Classical Filesystems
 - FFS
 - FAT
- Improving Reliability
 - FSCK
 - Journaling
- **Journaling Filesystems**
 - **ext3/ext4**
 - **NTFS**
- Copy-On-Write
 - ZFS

ext2/ext3/ext4

- extended filesystem – default for Linux
- ext2 (1993)
 - “Block groups” rather than cylinder groups, of arbitrary size
- ext3 (2001)
 - Adds journaling
 - Configuration options choose to journal either everything or metadata-only
- ext4 (2006)
 - Extents, encryption
 - Used on modern-day linux systems

Extents reduce number of pointers to data blocks

- Extents
 - Instead of raw block addresses
 - Store starting block address and length
 - Greatly compacts sequentially stored data pointers in inodes
- ext4 uses extents
 - 4 extents per file
 - Large, fragmented files use hierarchical system like original inodes

Other ext4 advances

- Encryption
 - Encrypts a directory and all of its contents
 - File names *and* file data
 - AES encrypt/decrypt is performed on data blocks during read/write
- Directory data structure
 - Htree (specialized B-tree)
 - Enables large subdirectory chains and many files with good seek time

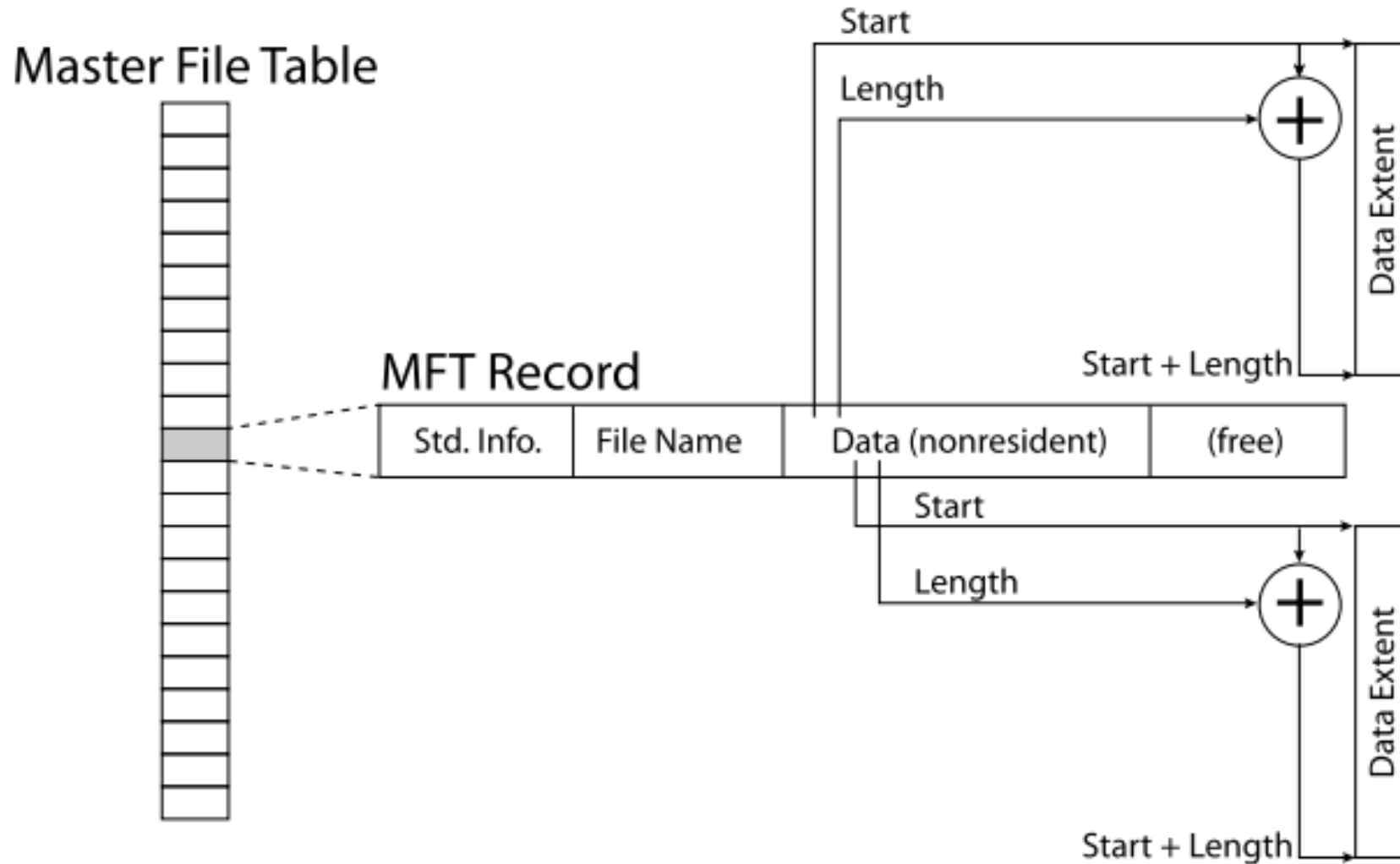
NTFS

- NT File System – modern Windows filesystem (1993)
 - Designed for Windows NT (Windows 2000 and up)
 - Uses Master File Table rather than Allocation Table
- Has grown to include many features we've seen
 - Journaling
 - Extents
 - Encryption
 - Directories using B-Trees
- Adds compression

NTFS Master File Table

- Master File Table
 - Similar in practice to an array of inodes
 - Usually one MFT record per file
 - But large files can include pointers to additional MFT records
- Each MFT Record contains
 - Standard attributes
 - Name and pointer to parent directory
 - Storage space
 - Can hold extents to point to series of data blocks
 - Can hold pointers to additional MFT records (for more data blocks)
 - Can hold file data itself!! (if small enough)

NTFS with medium-sized, mostly non-fragmented file



NTFS with a small file

Master File Table

| |
|--|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |

MFT Record (small file)

| | | | |
|------------|-----------|-----------------|--------|
| Std. Info. | File Name | Data (resident) | (free) |
|------------|-----------|-----------------|--------|

NTFS can automatically compress files

- Before write to disk, compress file data blocks
 - Only write smaller compressed data
- After read from disk, decompress file data blocks
- Interesting tradeoff
 - Read less total blocks from disk
 - Spend more CPU time manipulating blocks

Break + Extend Thinking

- In Windows 10, a service compresses infrequently used files
 - What files will this work on and what won't this help with?

Break + Extend Thinking

- In Windows 10, a service compresses infrequently used files
 - What files will this work on and what won't this help with?
- Text files are super compressible!!
- Code binaries are maybe compressible
- Unfortunately, can't compress *already compressed* files
 - Particularly: videos and music

Outline

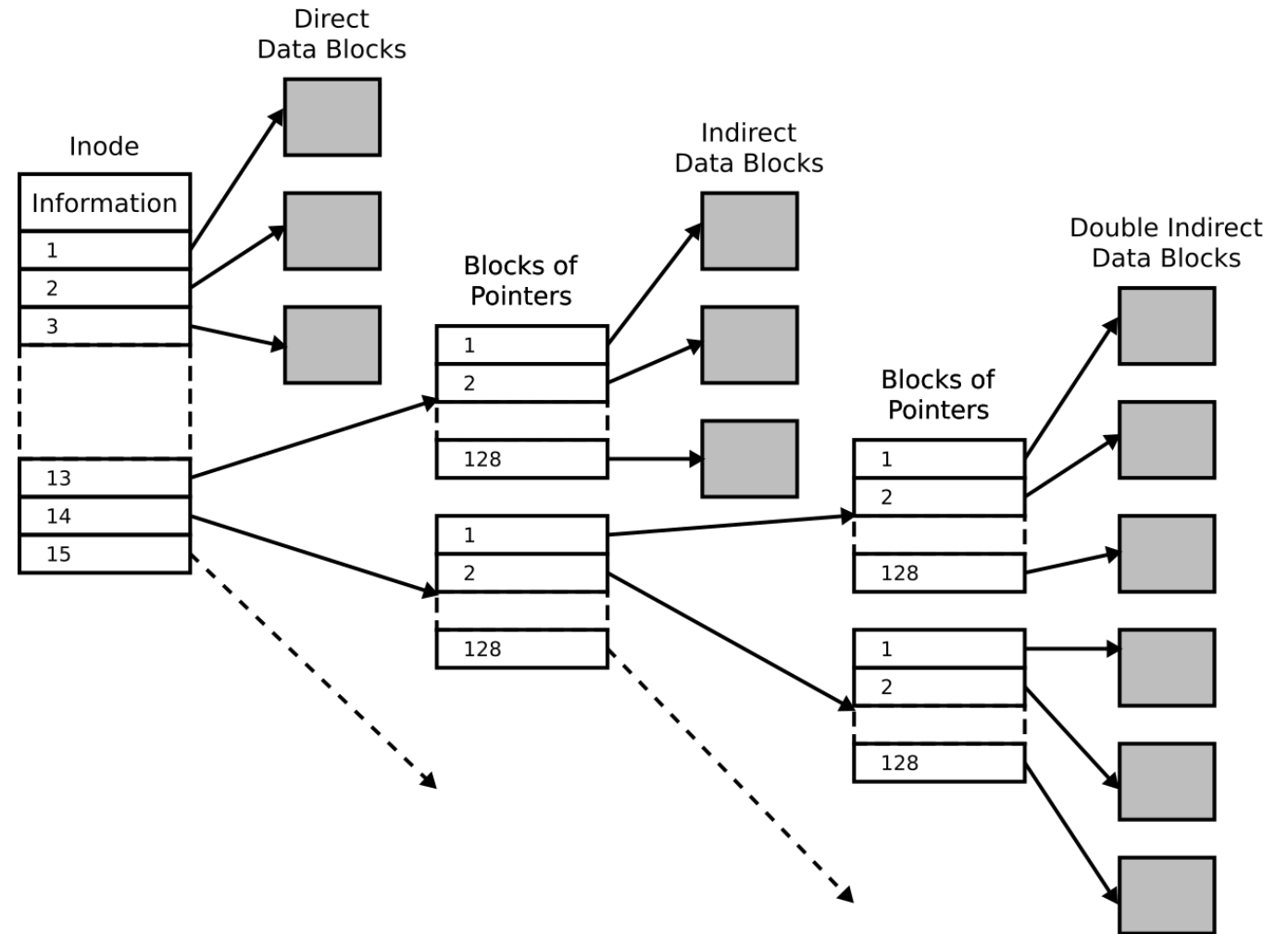
- Disk Caching
- Classical Filesystems
 - FFS
 - FAT
- Improving Reliability
 - FSCK
 - Journaling
- Journaling Filesystems
 - ext3/ext4
 - NTFS
- **Copy-On-Write**
 - **ZFS**

Adding file versioning through copy-on-write

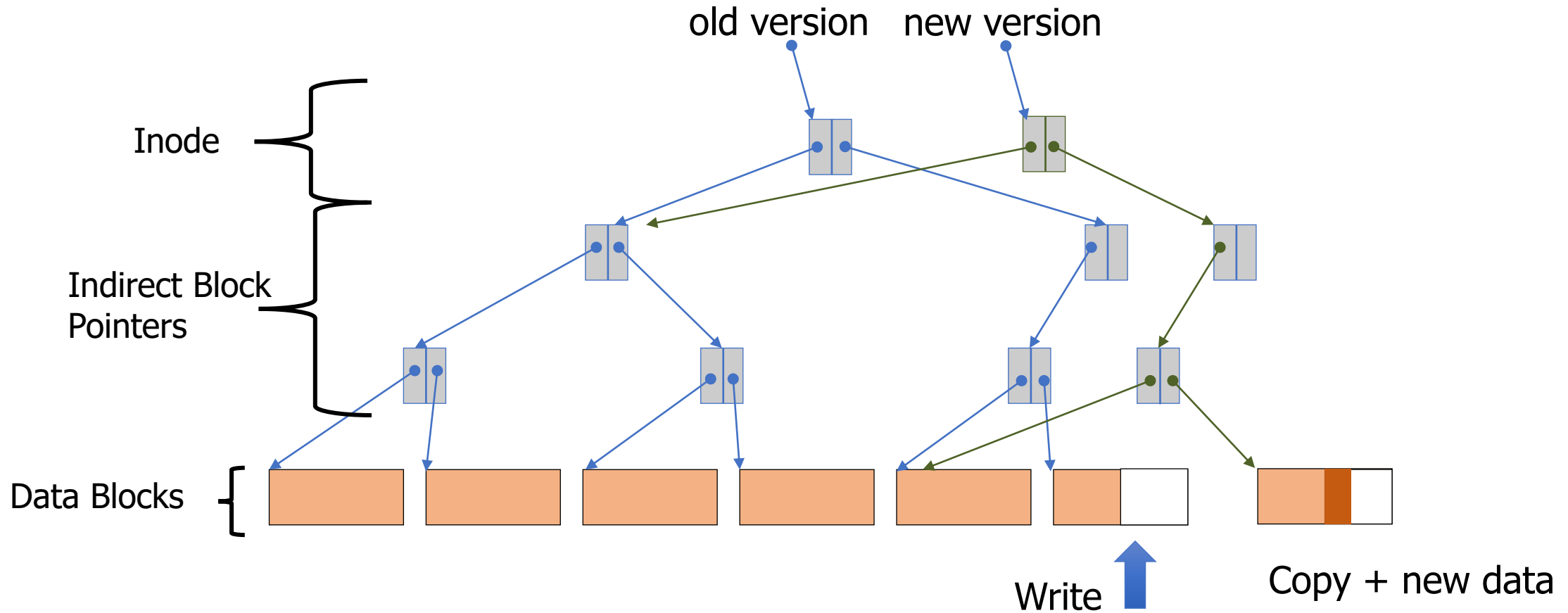
- Correctness could also come with a bonus: ability to version files
 - File could be rolled back to an older version from a prior point in time
- Method: instead of over-writing existing data block
 - Write update to a brand new data block
 - Create a new inode for the file that points to the new data block
 - And still points to original data for the other unmodified blocks
 - New inode now points to new version of file
 - Old inode now points to old version of file
- No longer needs journal for correctness

Reminder: hierarchical inodes

- Some bit in each entry specifies whether it points at:
 1. A data block
 2. A block with additional data pointers
- This system can recurse multiple layers deep
 - Allows for really large files



Copy-on-write example



ZFS

- Developed by Sun Microsystems, now Oracle (2006)
- Uses Copy-on-Write transactions
- Snapshots
 - Enabled by copy-on-write
 - Points in time for the filesystem can be “snapshot”
 - Files can be returned to prior versions from the snapshot

Pooled file system

- ZFS (and other filesystems) use a concept of pools of storage
 - Flips around disk-filesystem relationship
 - Instead of one filesystem per partition and multiple partitions per disk
 - One filesystem manages multiple disks
- Replaces need for RAID by allowing filesystem to make choices
- Common design pattern in computer systems
 - Abstractions make systems easy to use
 - Breaking abstractions allows for improved performance

Log-Structured File Systems

- Can go further along copy-on-write path
 - Entire disk is just a log of updates to files and inodes
- No longer doing small writes all over disk
 - Jumping between inodes and data blocks
 - Small, random writes are bad for HDD seek
- Instead, treat disk as a circular buffer that updates are written to
 - Write new data, then new inode after it, then next new data
 - All writes end up occurring sequentially
 - Garbage collect old file versions eventually when space gets low

Outline

- Disk Caching
- Classical Filesystems
 - FFS
 - FAT
- Improving Reliability
 - FSCK
 - Journaling
- Journaling Filesystems
 - ext3/ext4
 - NTFS
- Copy-On-Write
 - ZFS