

CS 318 Principles of Operating Systems

Fall 2019

Lecture 16: Advanced File Systems

Prof. Ryan Huang



JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

File Systems Examples

- **BSD Fast File System (FFS)**
 - What were the problems with the original Unix FS?
 - How did FFS solve these problems?
- **Log-Structured File system (LFS)**
 - What was the motivation of LFS?
 - How did LFS work?

Original Unix FS

- From Bell Labs by Ken Thompson
- Simple and elegant:

Unix disk layout



- **Components**
 - Data blocks
 - Inodes (directories represented as files)
 - Free list
 - Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
- **Problem: slow**
 - Only gets **2% of disk maximum** (20Kb/sec) even for sequential disk transfers!

Why So Slow?

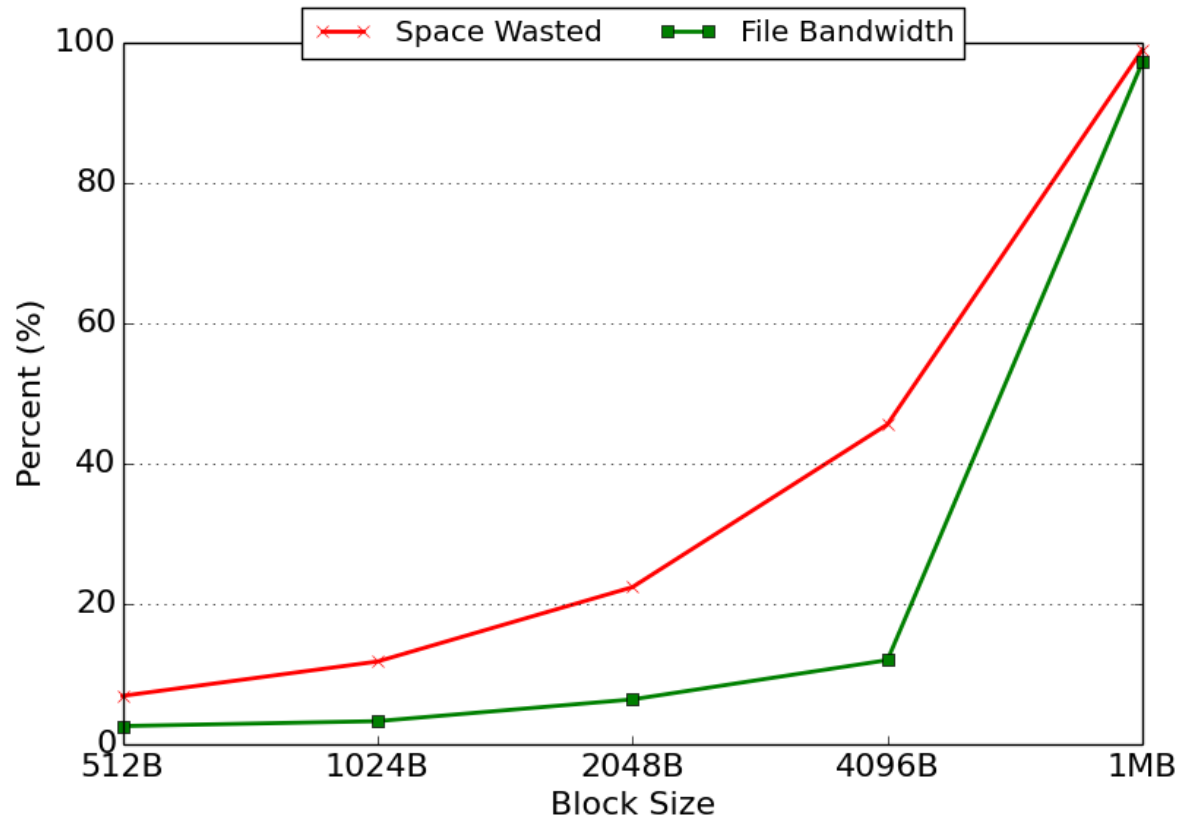
- **Problem 1: blocks too small (512 bytes)**
 - File index too large
 - Require more indirect blocks
 - Transfer rate low (get one block at time)
- **Problem 2: unorganized freelist**
 - Consecutive file blocks not close together
 - Pay seek cost for even sequential access
 - Aging: becomes fragmented over time
- **Problem 3: poor locality**
 - inodes far from data blocks
 - inodes for directory not close together
 - poor enumeration performance: e.g., “ls”, “grep foo *.c”

FFS: Fast File System

- **Designed by a Berkeley research group for the BSD UNIX**
 - A classic file systems paper to read: [[McKusic](#)]
- **Approach:**
 - **measure** an state of the art systems
 - identify and understand the fundamental problems
 - **The original FS treats disks like random-access memory!**
 - get an idea and **build** a better systems
- **Idea: design FS structures and allocation polices to be “disk aware”**
- **Next: how FFS fixes the performance problems (to a degree)**

Problem 1: Blocks Too Small

Measurement:

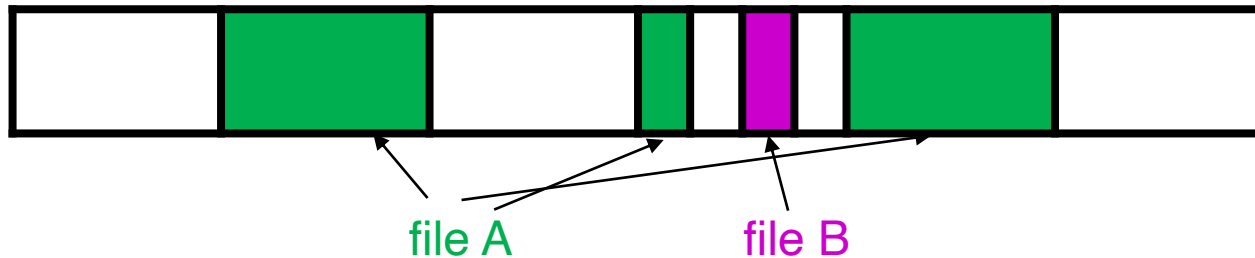


- **Bigger block increases bandwidth, but how to deal with wastage (“**internal fragmentation**”)?**
 - Use idea from malloc: split unused portion

Solution: Fragments

- **BSD FFS:**

- Has large block size (4096B or 8192B)
- Allow large blocks to be chopped into small ones called “fragments”
- Ensure fragments only used for little files or ends of files



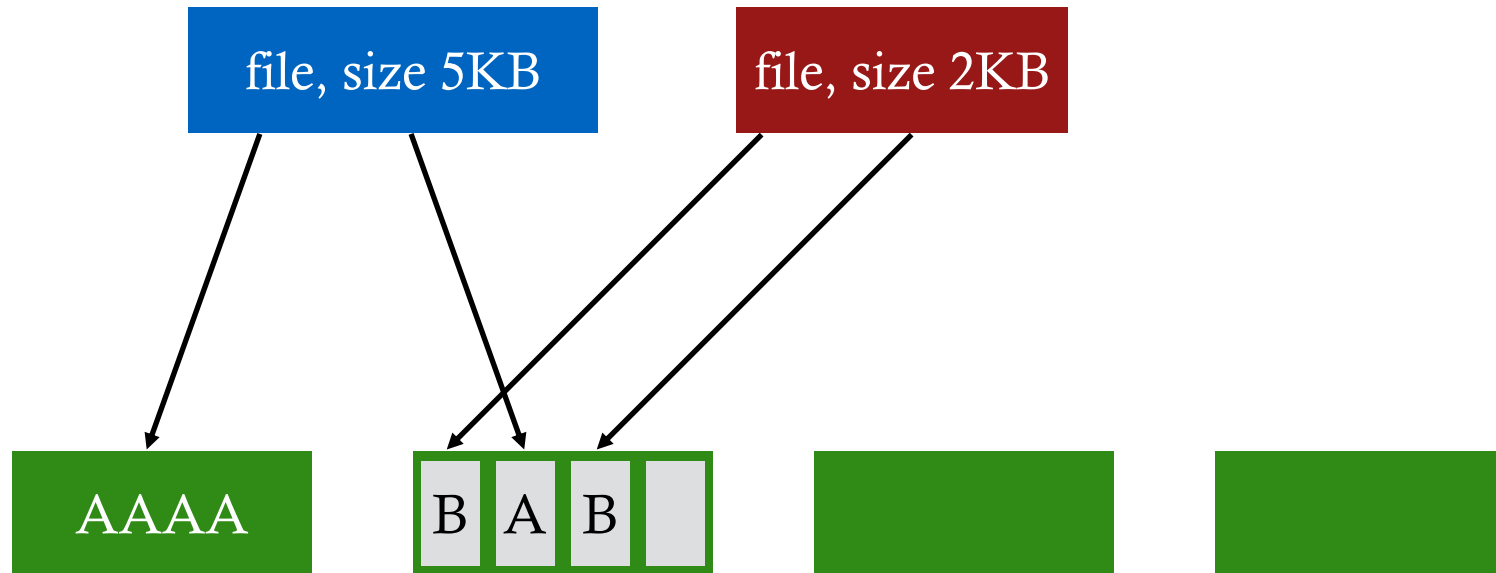
- Fragment size specified at the time that the file system is created
- Limit number of fragments per block to 2, 4, or 8

- **Pros**

- High transfer speed for larger files
- Low wasted space for small files or ends of files

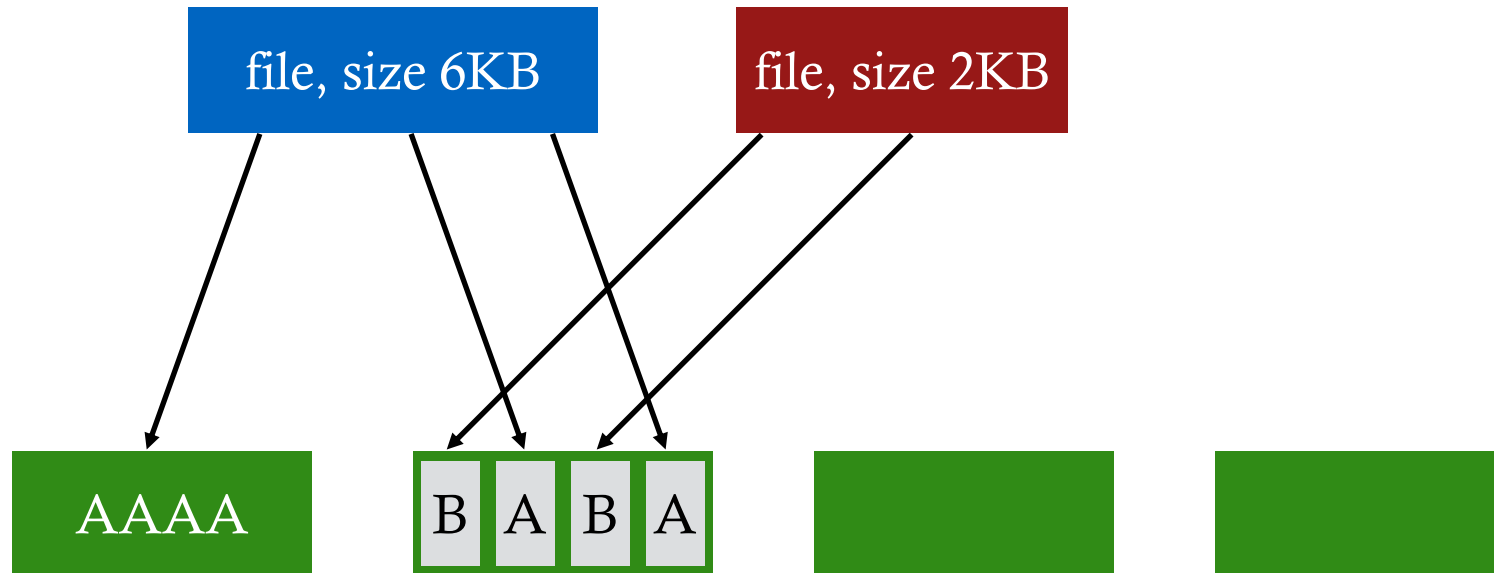
Fragment Example

Block size: 4096 B
Fragment size: 1024 B



Fragment Example

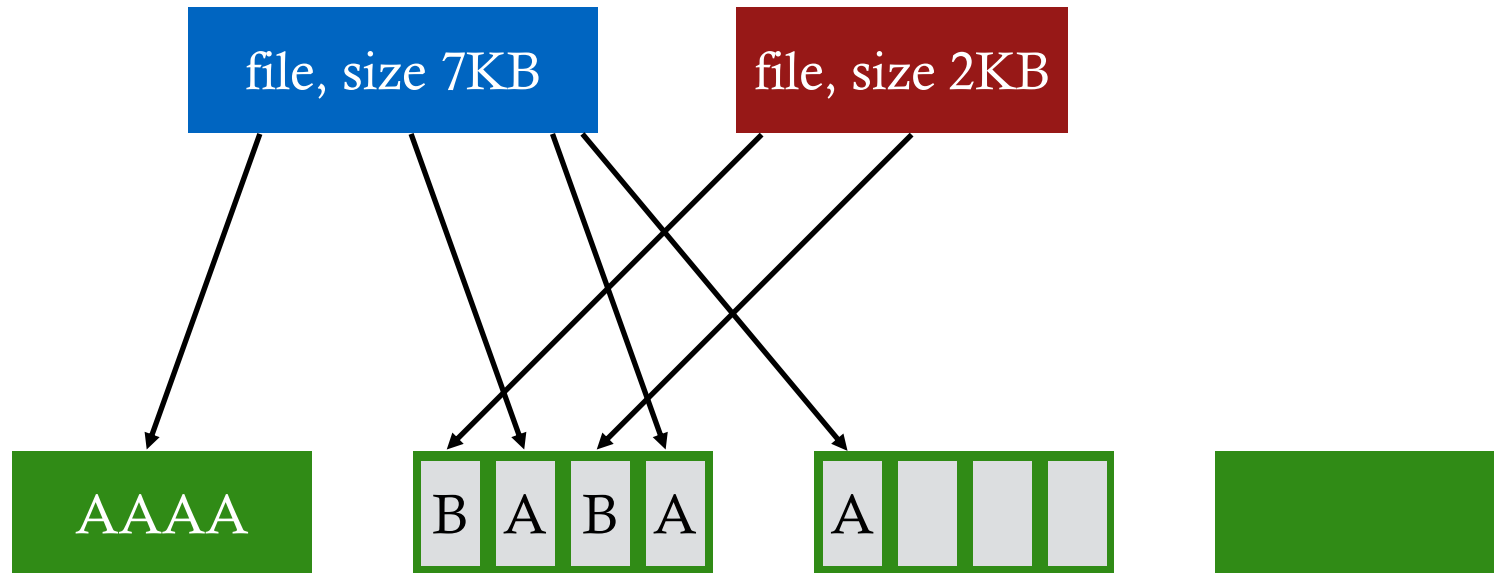
Block size: 4096 B
Fragment size: 1024 B



```
write(fd1, "A"); // append A to first file
```

Fragment Example

Block size: 4096 B
Fragment size: 1024 B



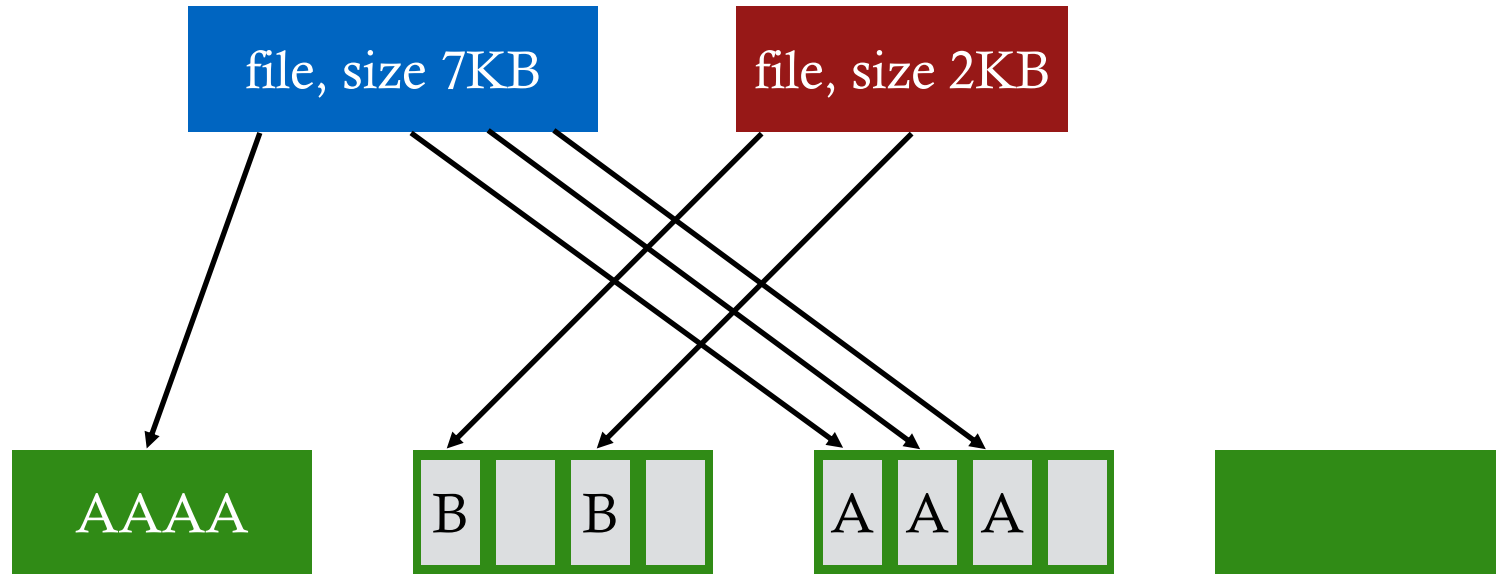
```
write(fd1, "A"); // append A to first file
```

Not allowed to use fragments across multiple blocks!

What to do instead?

Fragment Example

Block size: 4096 B
Fragment size: 1024 B

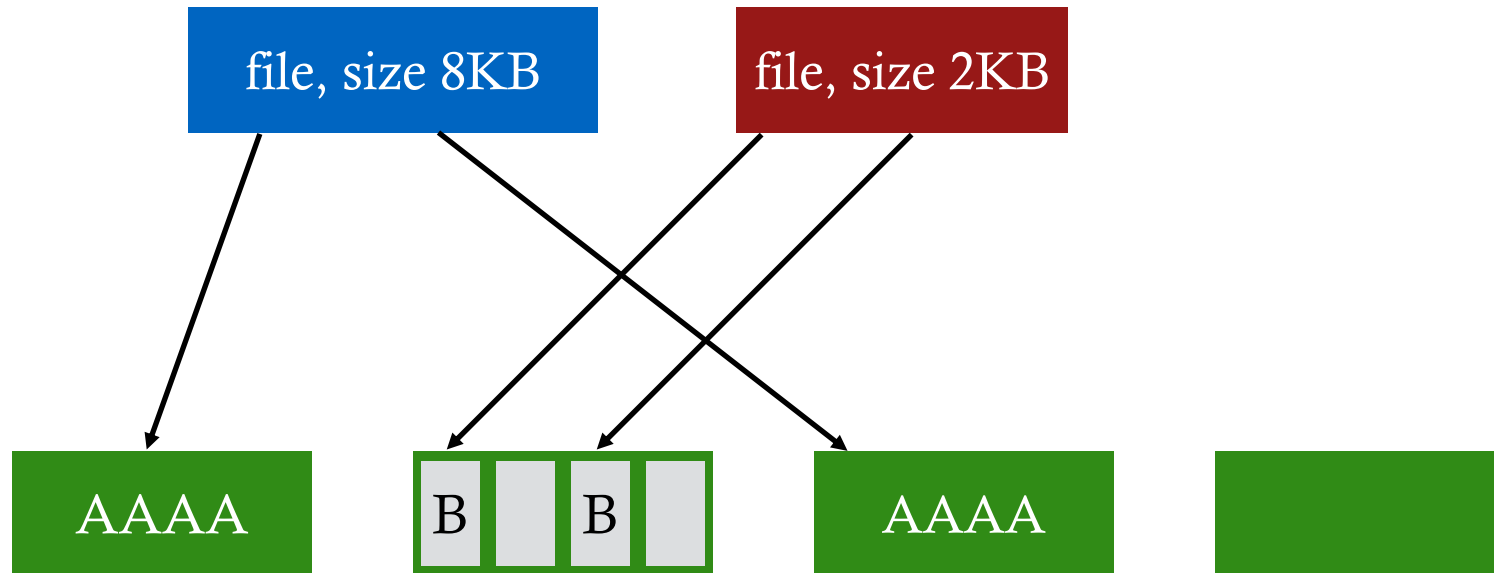


```
write(fd1, "A"); // append A to first file
```

copy old fragments to new block
new data use remaining fragments

Fragment Example

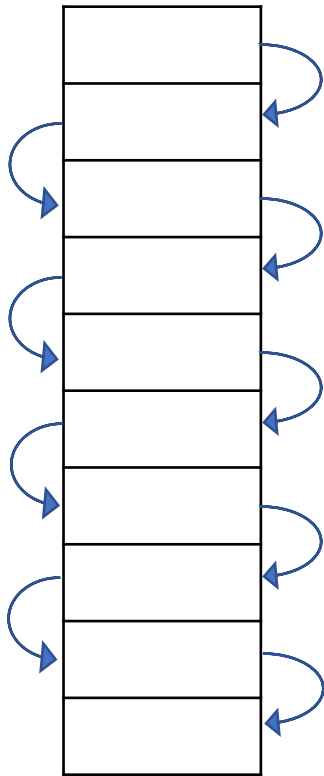
Block size: 4096 B
Fragment size: 1024 B



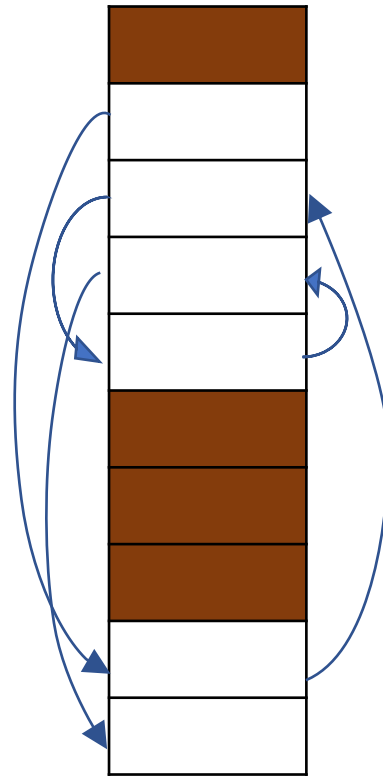
```
write(fd1, "A"); // append A to first file
```

Problem 2: Unorganized Freelist

- **Leads to random allocation of sequential file blocks overtime**



Initial performance good



Get worse over time

Measurement:

- New FS: **17.5%** of disk bandwidth
- Few weeks old: **3%** of disk bandwidth

Fixing the Unorganized Freelist

- **Periodical compact/defragment disk**
 - Cons: locks up disk bandwidth during operation
- **Keep adjacent free blocks together on freelist**
 - Cons: costly to maintain
- **FFS: bitmap of free blocks**
 - Each bit indicates whether block is free
 - E.g., 1010101111111000001111111000101100
 - Easier to find contiguous blocks
 - Small, so usually keep entire thing in memory
 - Time to find free blocks increases if fewer free blocks
 - What about fragments in a block?

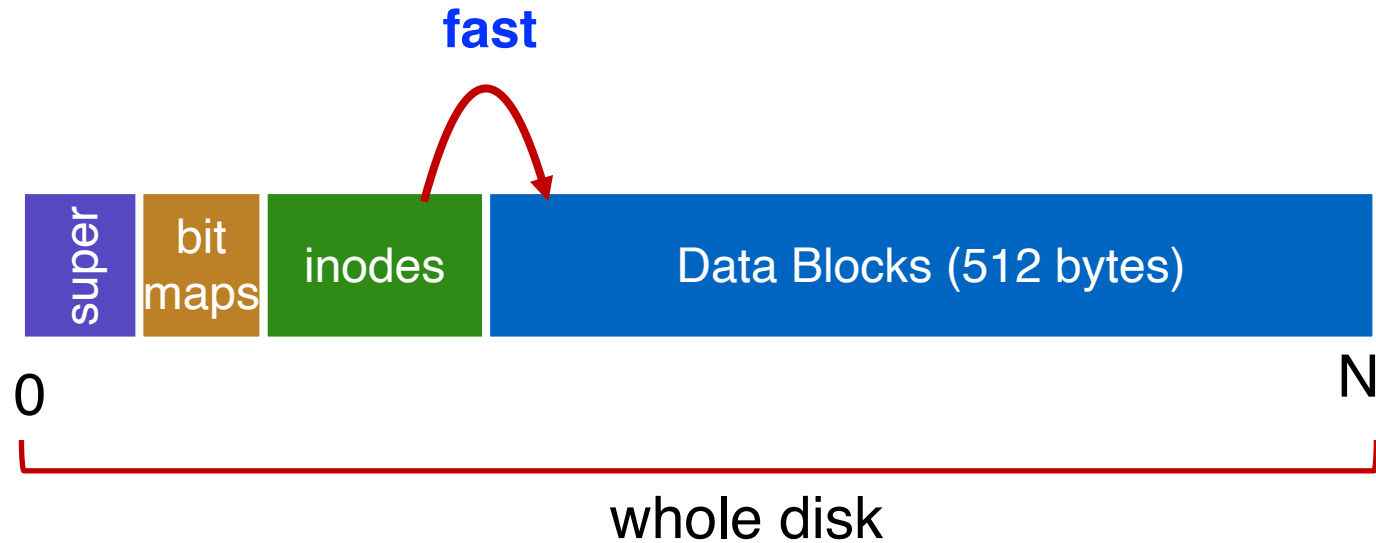
Bits in map	XXXX	XXOO	OOXX	O000
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Using a Bitmap

- **Usually keep entire bitmap in memory:**
 - 4G disk / 4K byte blocks. How big is map?
- **Allocate block close to block x?**
 - Check for blocks near $\text{bmap}[x/32]$
 - If disk almost empty, will likely find one near
 - As disk becomes full, search becomes more expensive and less effective
- **Trade space for time (search time, file access time)**

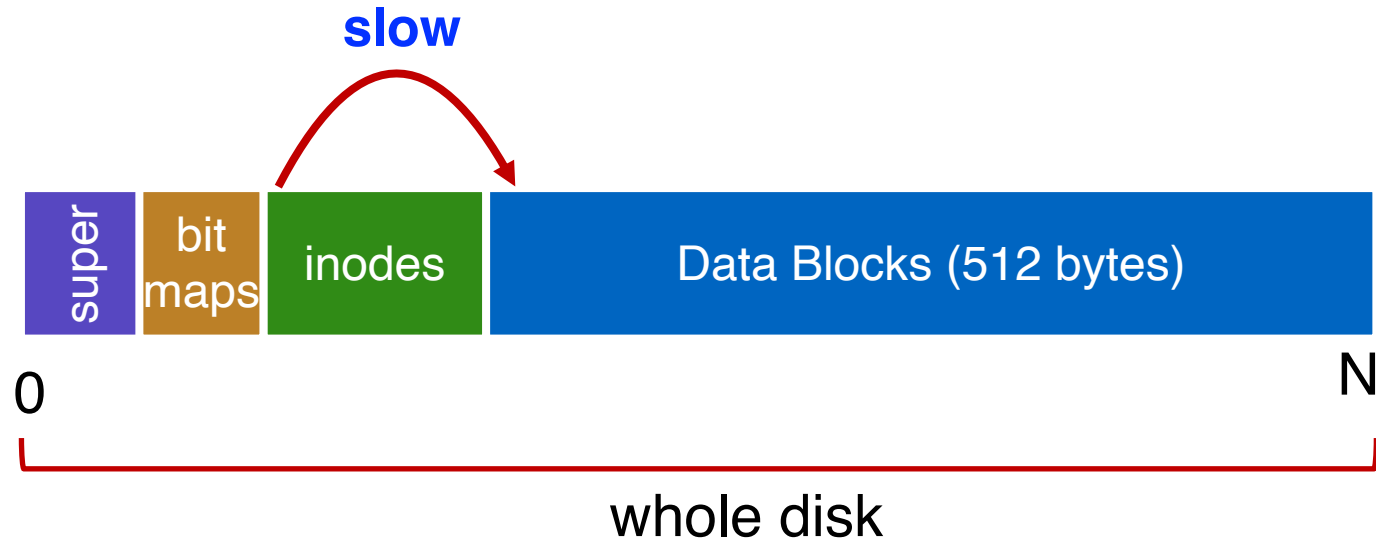


Problem 3: Poor Locality



- **How to keep inode close to data block?**

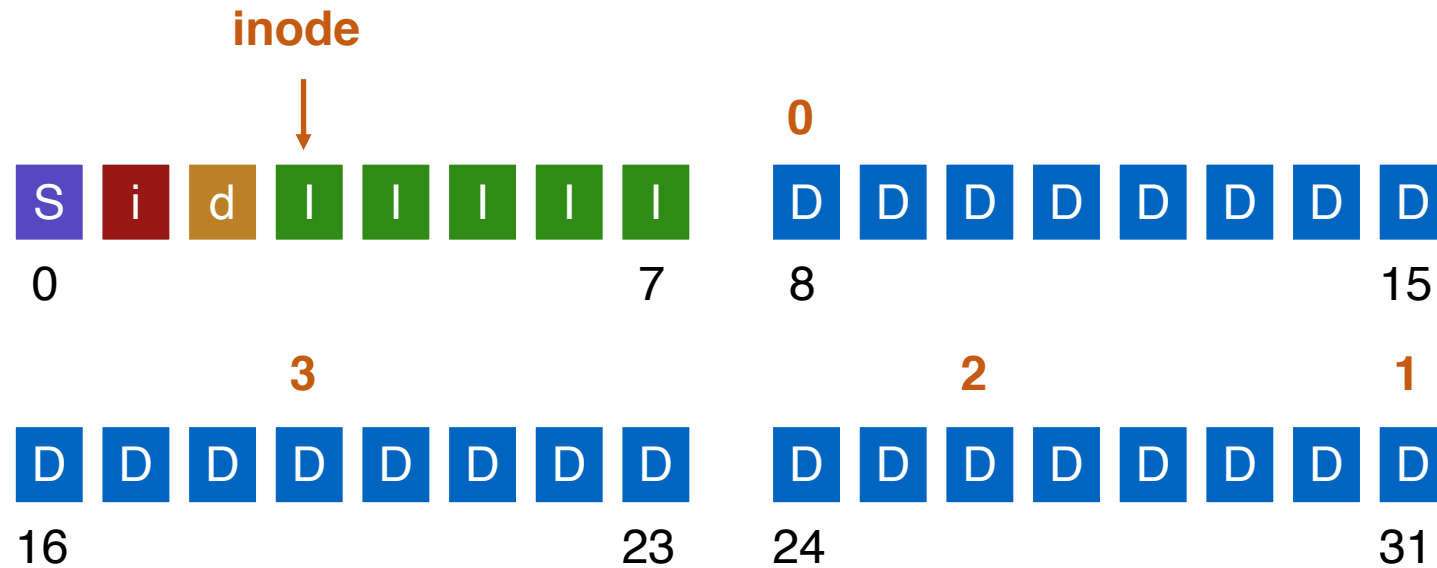
Problem 3: Poor Locality



- **How to keep inode close to data block?**

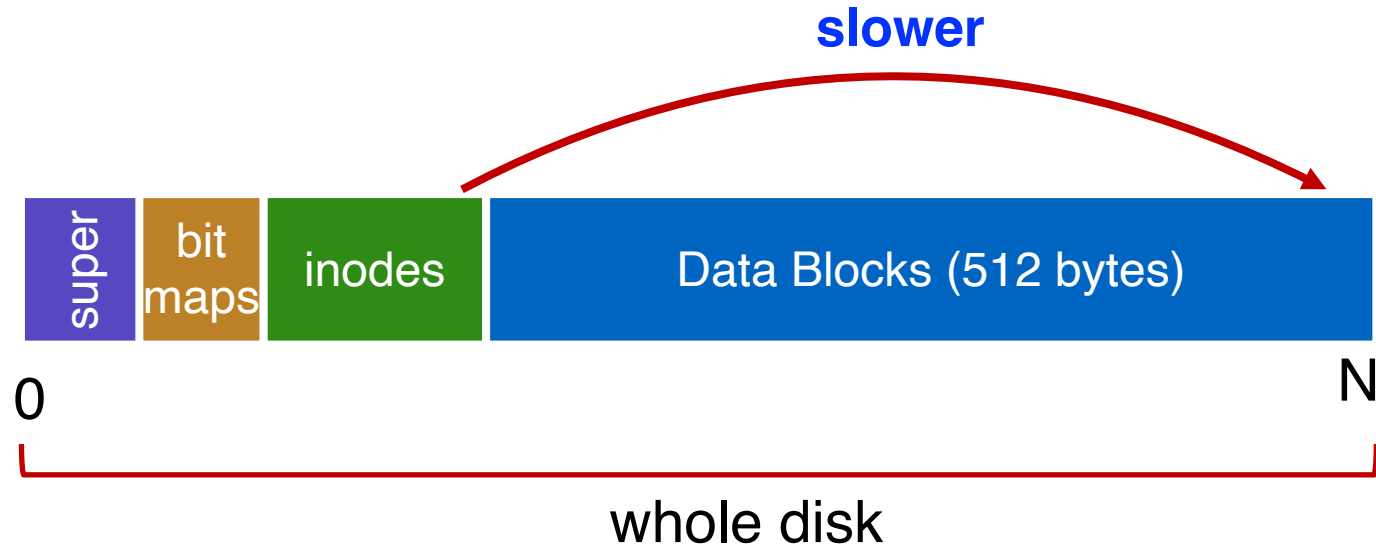
Problem 3: Poor Locality

- **Example bad layout:**



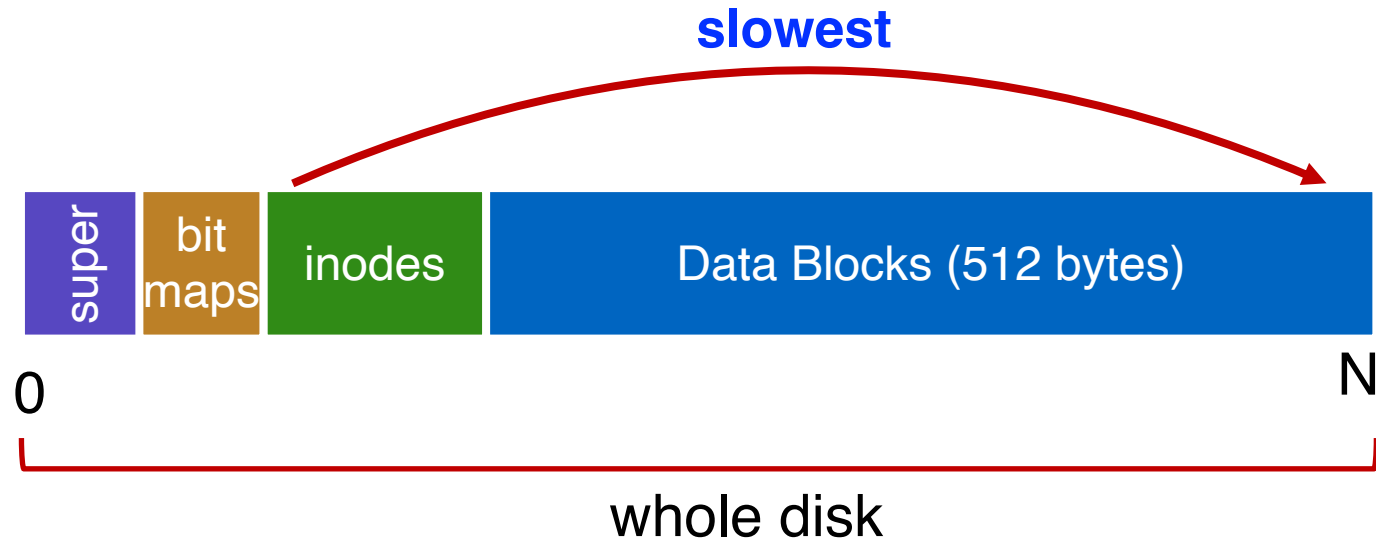
- **How to keep inode close to data block?**

Problem 3: Poor Locality



- **How to keep inode close to data block?**

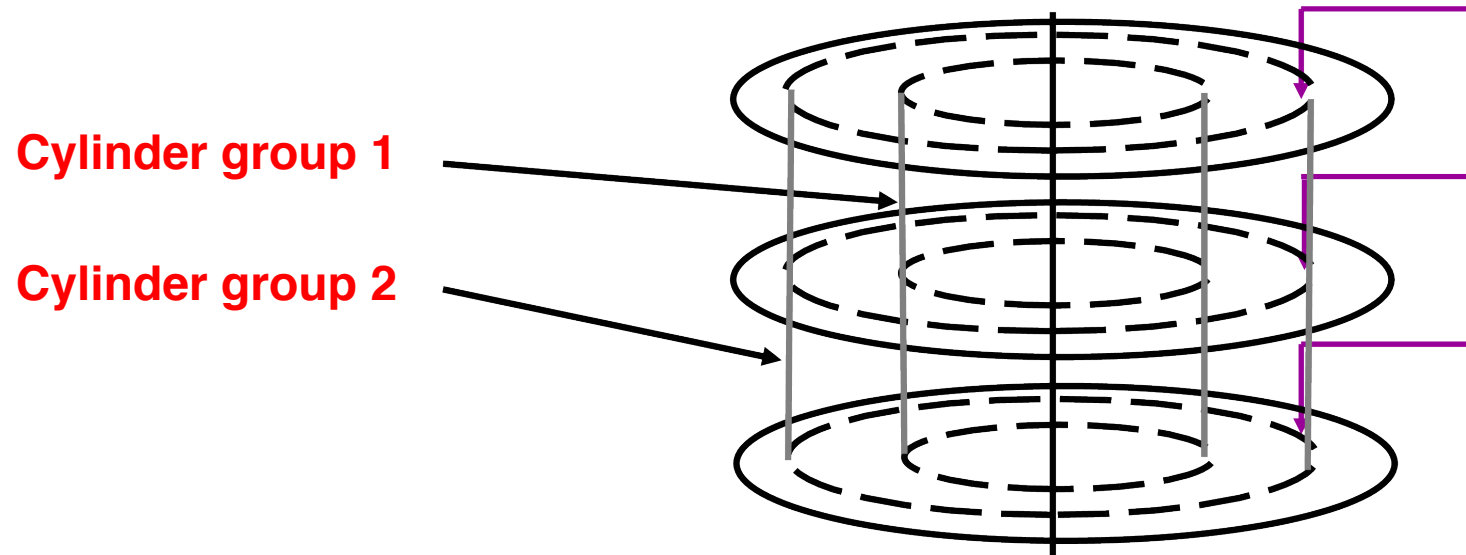
Problem 3: Poor Locality



- **How to keep inode close to data block?**

FFS Solution: Cylinder Group

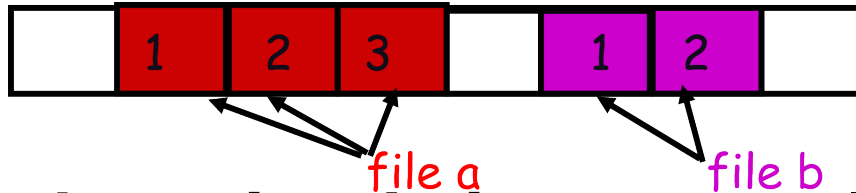
- Group sets of consecutive cylinders into “**cylinder groups**”



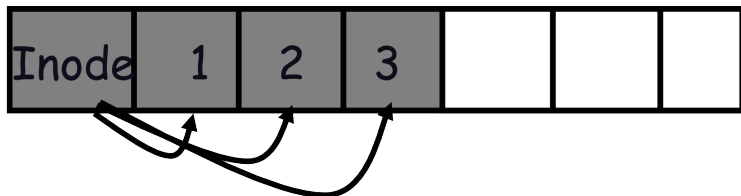
- Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
- Tries to put everything related in same cylinder group
- Tries to put everything not related in different group

Clustering in FFS

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

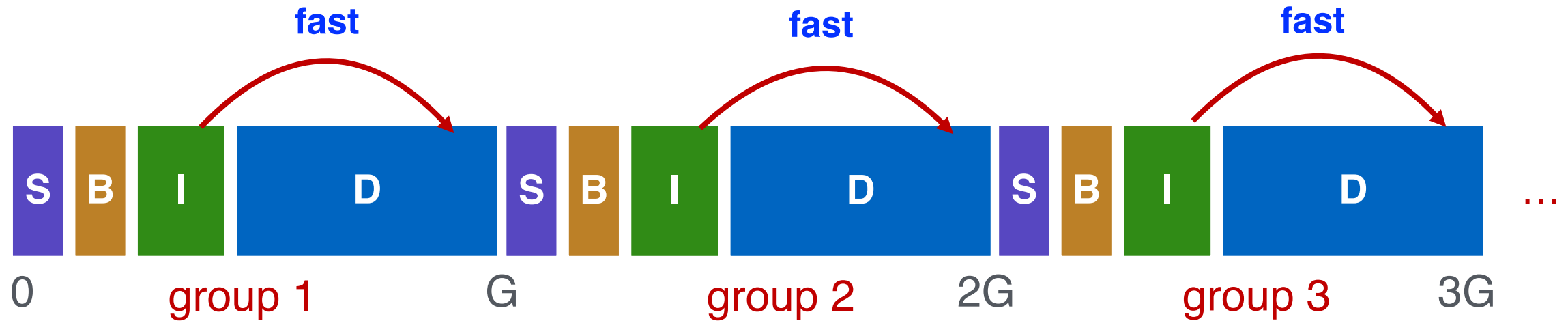


- **Tries to keep inode in same cylinder as file data:**
 - (If you look at inode, most likely will look at data too)



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

What Does Disk Layout Look Like Now?



- **How to keep inode close to data block?**
 - Answer: Use groups across disks
 - Strategy: allocate inodes and data blocks in same group
 - Each cylinder group basically a mini-Unix file system
- **Is it useful to have multiple super blocks?**
 - Yes, if some (but not all) fail

FFS Results

- **Performance improvements:**
 - Able to get 20-40% of disk bandwidth for large files
 - 10-20x original Unix file system!
 - Stable over FS lifetime
 - Better small file performance (why?)
- **Other enhancements**
 - Long file names
 - Parameterization
 - Free space reserve (10%) that only admin can allocate blocks from

LFS: Log-structured File System

- **Motivation**

- Faster CPUs: I/O becomes more and more of a bottleneck
- More memory: file cache is effective for reads
- Implication: writes compose most of disk traffic

- **Problems with previous FS**

- Perform many small writes
 - Good performance on large, sequential writes, but many writes are still small, random
- Synchronous operation to avoid data loss
- Depends upon knowledge of disk geometry

- **An influential work designed by Mendel Rosenblum (VMWare co-founder) and John Ousterhout**

LFS Idea

- **Insight: treat disk like a tape-drive**
 - Best performance from disk for sequential access
- **File system buffers writes in main memory until “enough” data**
 - How much is enough?
 - Enough to get good sequential bandwidth from disk (MB)
 - Unit called a “segment”
- **Write buffered data to new segment on disk in a sequential log**
 - Transfer all updates into a series of sequential writes
 - **Do not overwrite old data on disk: old copies left behind**
 - Write both data and metadata in one operation

Pros And Cons

- **Pros**

- Always large sequential writes → good performance
- No knowledge of disk geometry
 - Assume sequential better than random

- **Potential problems**

- How do you find data to read?
- What happens to metadata during write?
- What happens when you fill up the disk?

Read in LFS

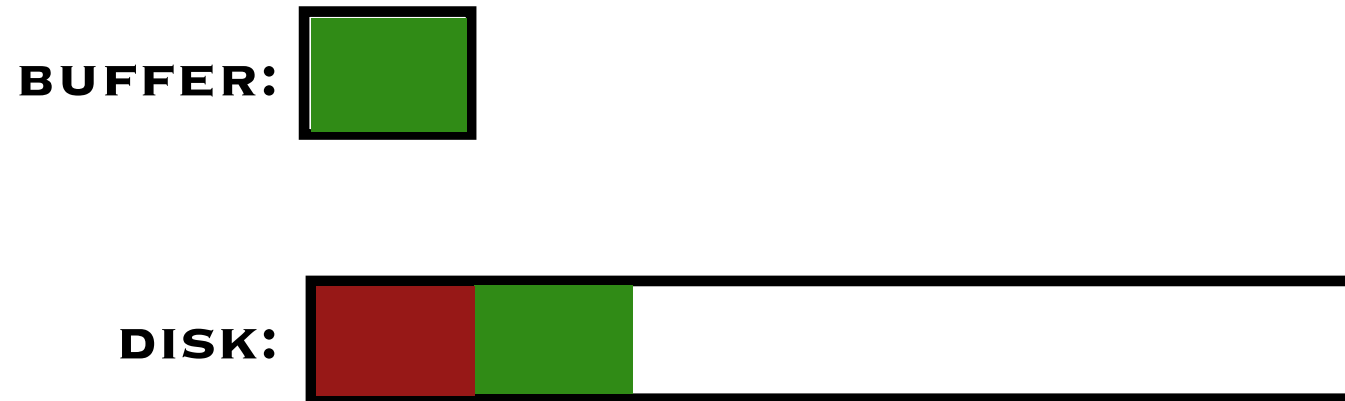
- **Same basic structures as Unix**
 - Directories, inodes, indirect blocks, data blocks
 - Reading data block implies finding the file's inode
 - Unix FS: inodes kept in array
 - LFS: inodes spread around on disk
- **Solution: inode map indicates where each inode is stored**
 - Can keep cached copy in memory
 - inode map written to log with everything else
 - Periodically written to known checkpoint location on disk for crash recovery

Write in LFS

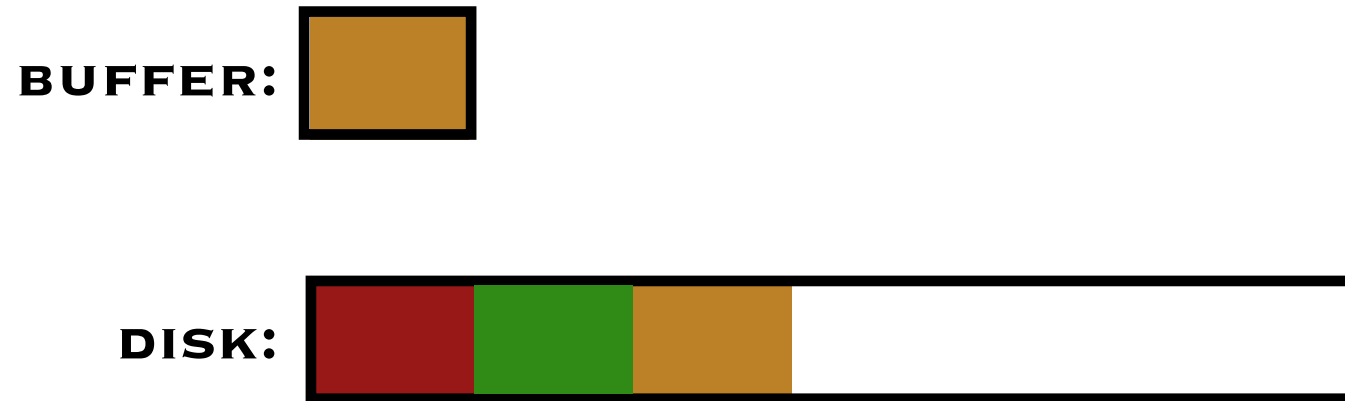


- **Why do we buffer the write?**
 - Sequential write alone is not enough
 - Disk is constantly rotating!
 - Must issue a large number of **contiguous** writes

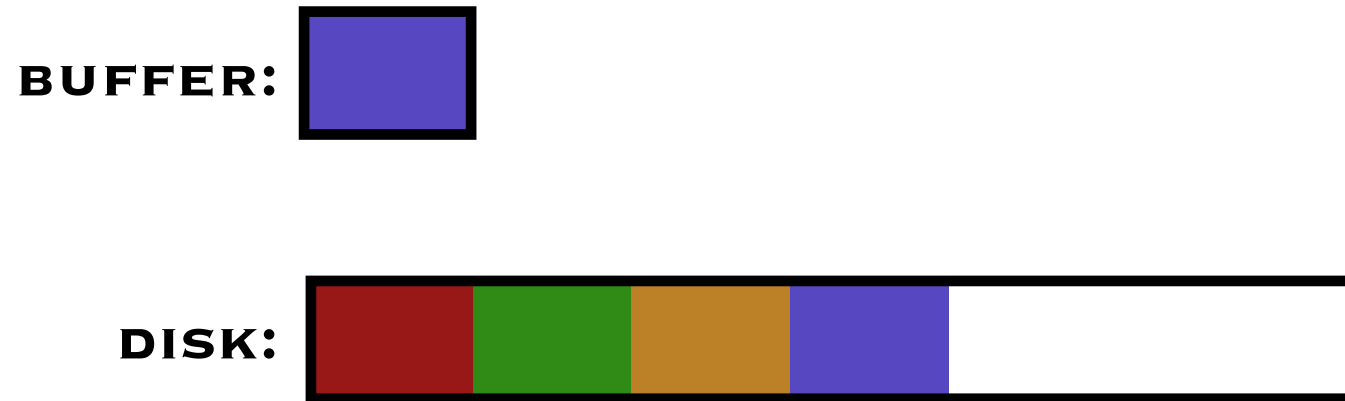
Write in LFS



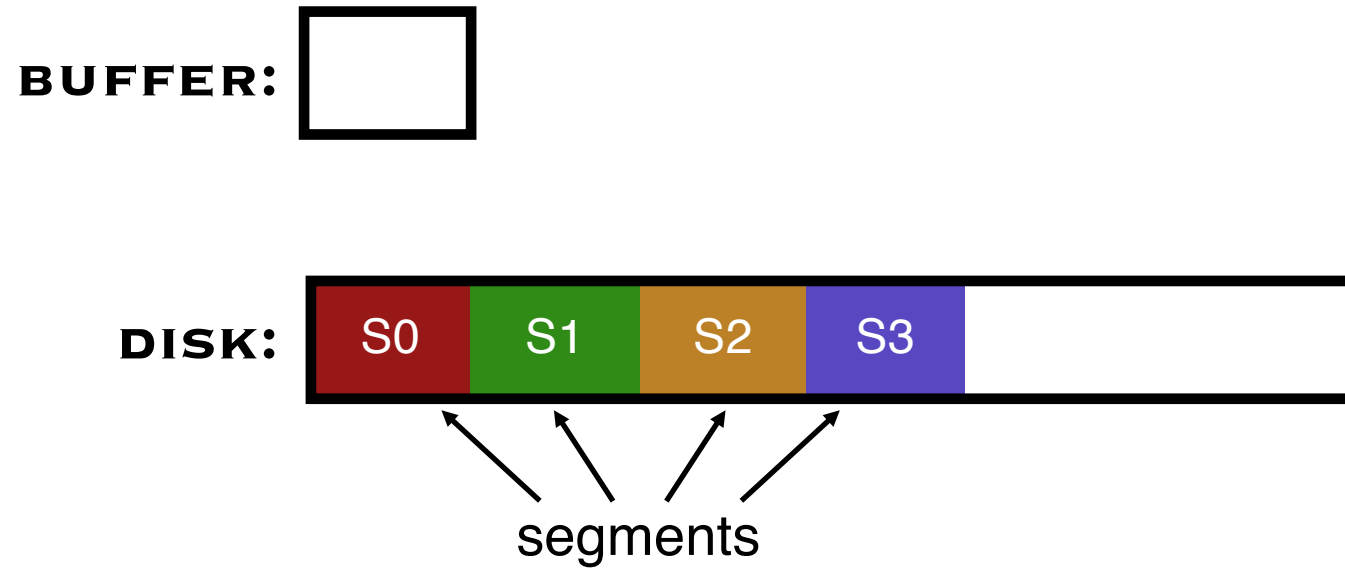
Write in LFS



Write in LFS



Write in LFS



Data Structures for LFS (attempt 1)



What data structures from FFS can LFS remove?

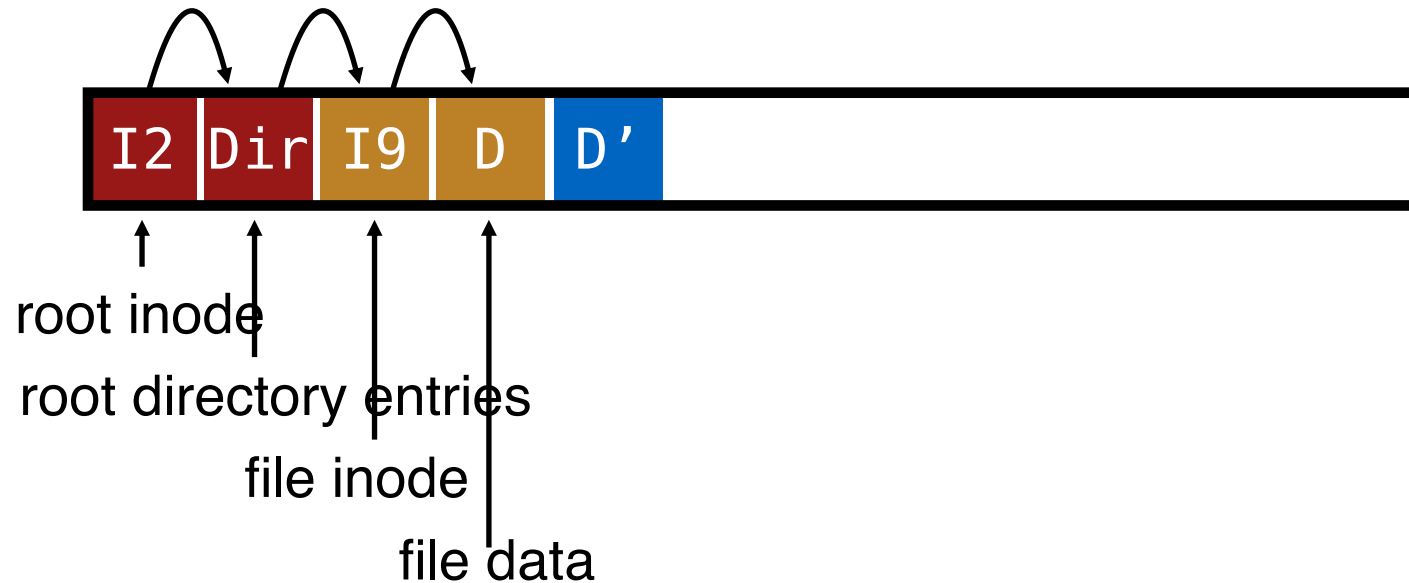
- allocation structs: data + inode bitmaps

What type of name is much more complicated?

- Inodes are no longer at fixed offset
- Use **current offset on disk** instead of table index for name
- Note: **when update inode, inode number changes!!**

Overwrite Data in LFS – Attempt 1

- **Overwrite data in /file.txt**



- **How to update Inode 9 to point to new D' ???**

Overwrite Data in LFS – Attempt 1

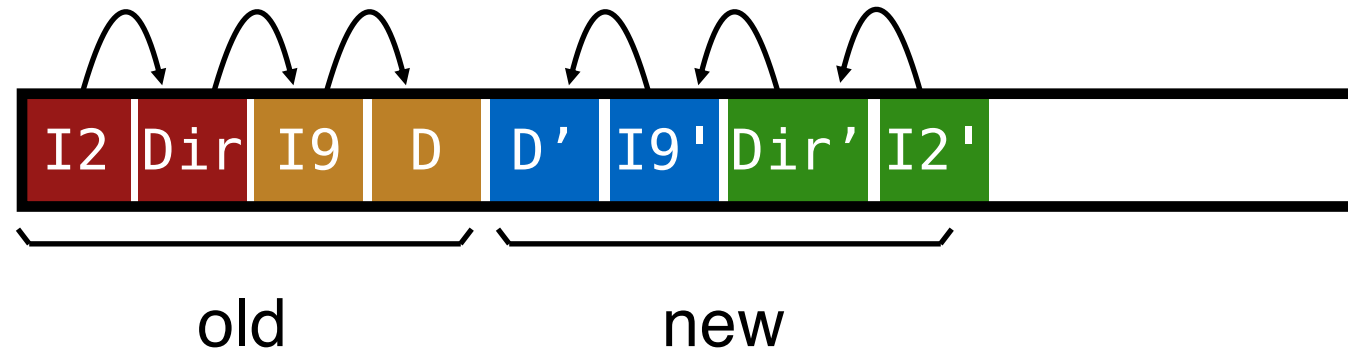
- Overwrite data in /file.txt



- **Can LFS update Inode 9 to point to new D'?**
 - NO! This would be a random write

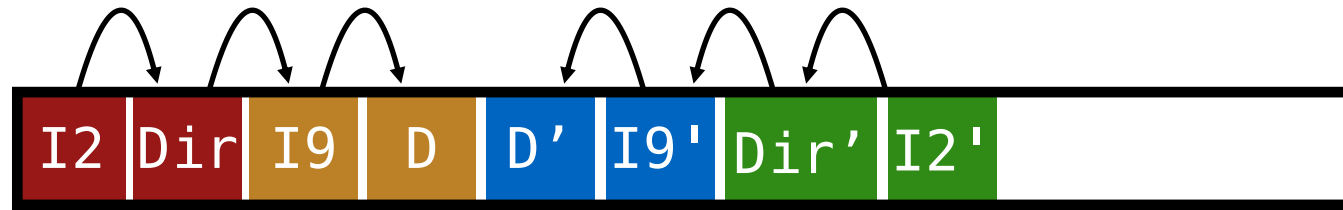
Overwrite Data in LFS – Attempt 1

- **Overwrite data in /file.txt**



- **Must update all structures in sequential order to log**

Attempt 1: Problem w/ Inode Numbers



- **Problem:**
 - For every data update, must propagate updates all the way up directory tree to root
- **Why?**
 - When inode copied, its location (inode number) changes
- **Solution:**
 - Keep inode numbers constant; don't base name on offset
- **FFS found inodes with math. How now?**

Data Structures for LFS (attempt 2)

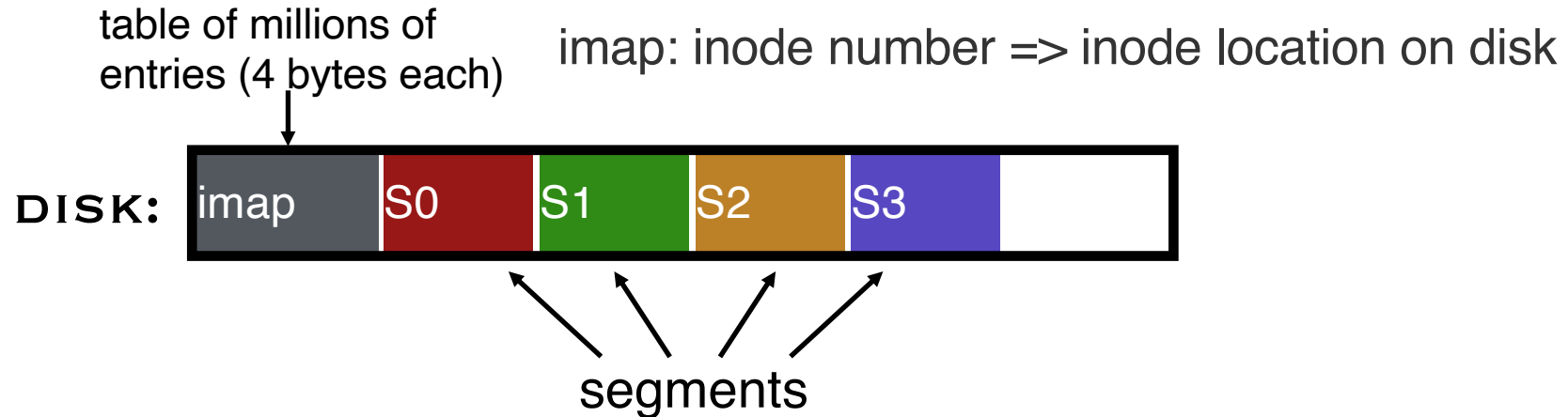
What data structures from FFS can LFS remove?

- allocation structs: data + inode bitmaps

What type of name is much more complicated?

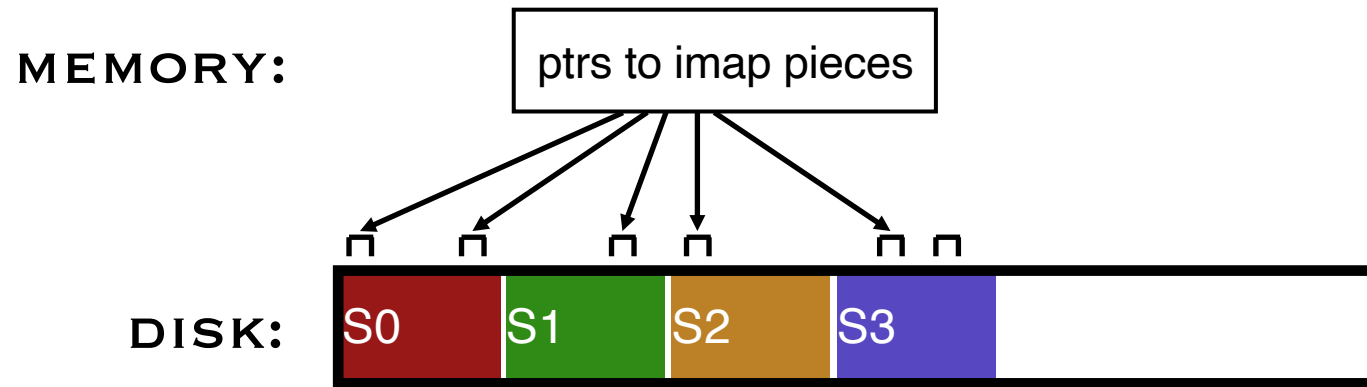
- Inodes are no longer at fixed offset
- Use **imap** structure to map:
 - inode number => **most recent** inode location on disk

Where to keep Imap?



- **Where can imap be stored? Dilemma:**
 1. imap too large to keep in memory
 2. don't want to perform random writes for imap
- **Solution: Write imap in segments**
 - Keep pointers to pieces of imap in memory

Solution: Imap in Segments



- **Solution:**

- Write imap in segments
- Keep pointers to pieces of imap in memory
- Keep recent accesses to imap cached in memory

Disk Cleaning

- **When disk runs low on free space**
 - Run a disk cleaning process
 - Compacts live information to contiguous blocks of disk
- **Problem: long-lived data repeatedly copied over time**
 - Solution: partition disk in to segments
 - Group older files into same segment
 - Do not clean segments with old files
- **Try to run cleaner when disk is not being used**

Next Time...

- **Read Chapter 42**