Advanced File Systems

Thierry Sans

Advanced File Systems

How to improve the performances?

- **BSD Fast File System (FFS)**

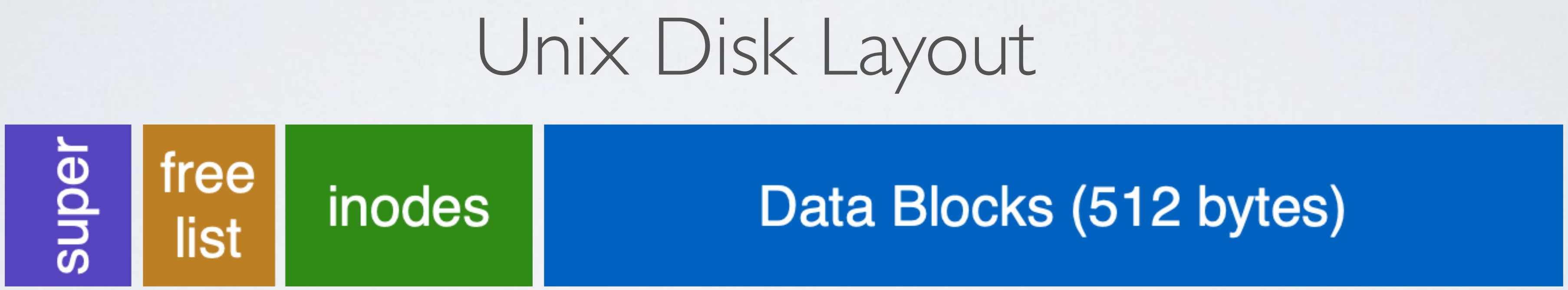
How to improve the reliability in case of a crash?

- **Log-Structured File system (LFS)**
- **Journaling File System (ext3)**

Improving Performances

with BSD Fast File System (FFS)

Original Unix FS



- It is slow on hard disk drive - only gets 2% of disk maximum (20Kb/sec) even for sequential disk transfers

Why so slow on hard disk drive?

Problem 1: in the original Unix File System, blocks were 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

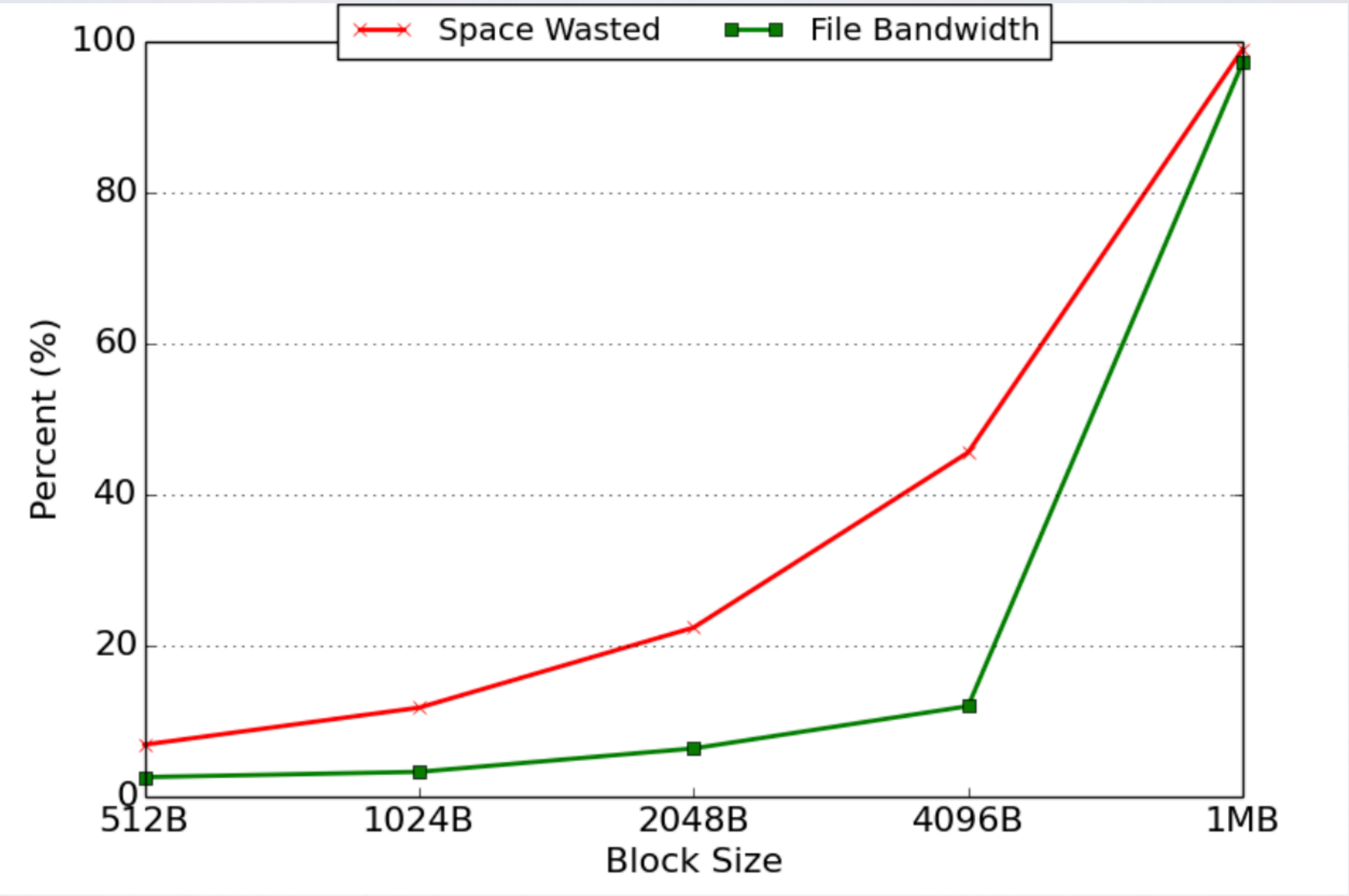
Problem 3: poor locality

- inodes far from data blocks
- inodes for directory not close together - poor enumeration performance
  e.g., "ls", "grep foo *.c"

FFS - Fast File System

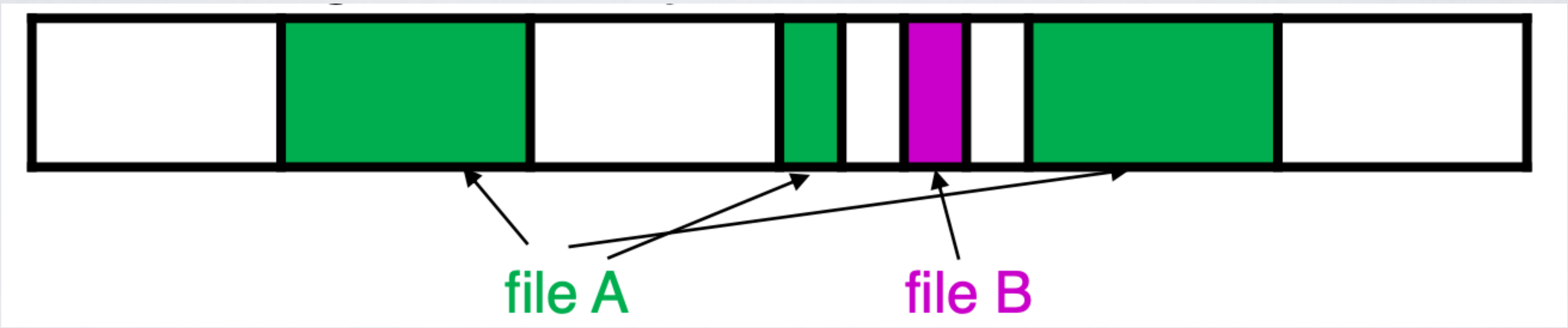
- ➡ Design FS structures and allocation polices to be "disk aware"
  Designed by a Berkeley research group for the BSD UNIX

Problem 1 - blocks are too small



- ✓ Bigger block increases bandwidth
- ◉ but increases internal fragmentation as well

Solution - use 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

✓ High transfer speed for larger files

✓ Low wasted space for small files or ends of files

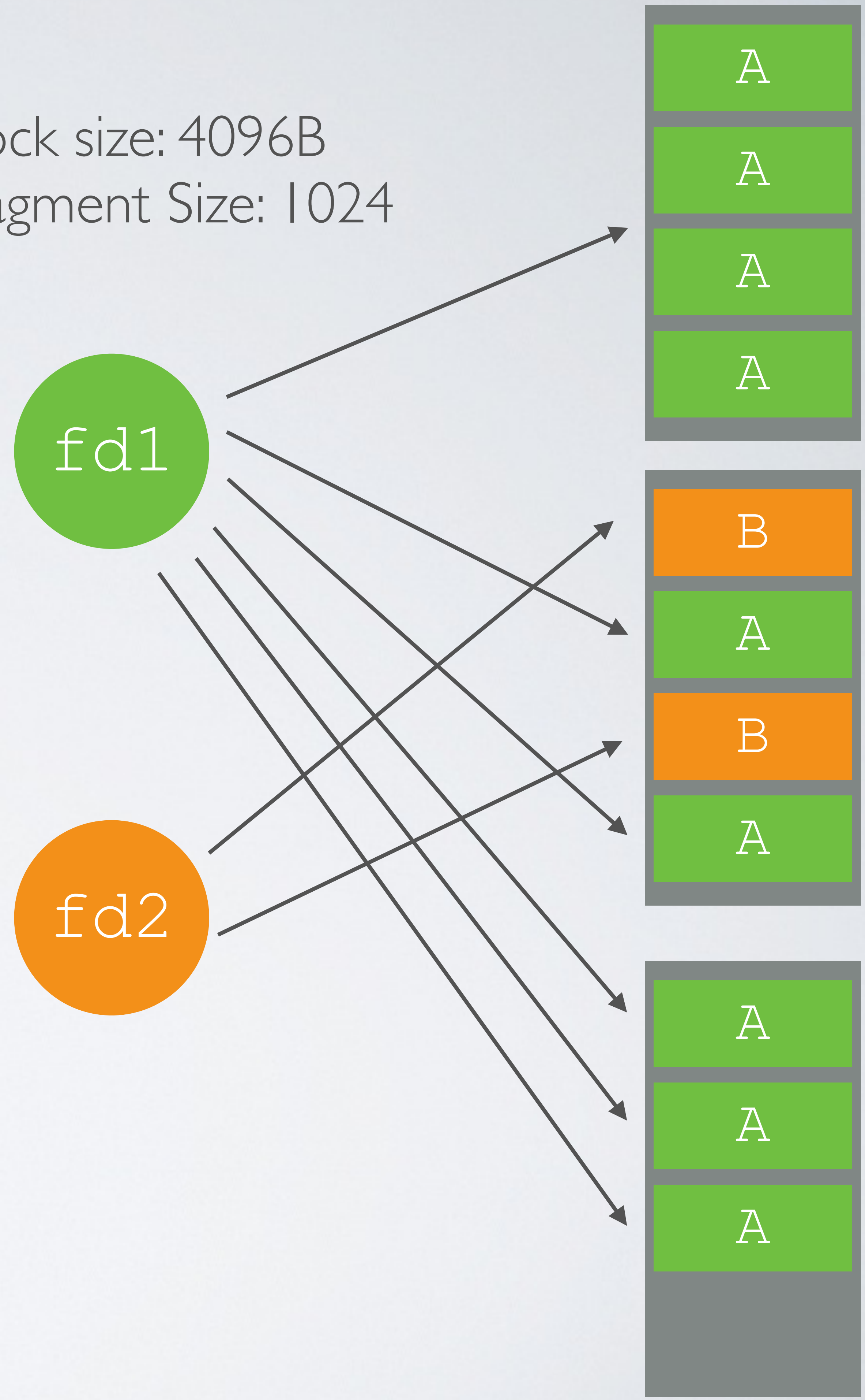
Fragment Example

Block size: 4096B
Fragment Size: 1024

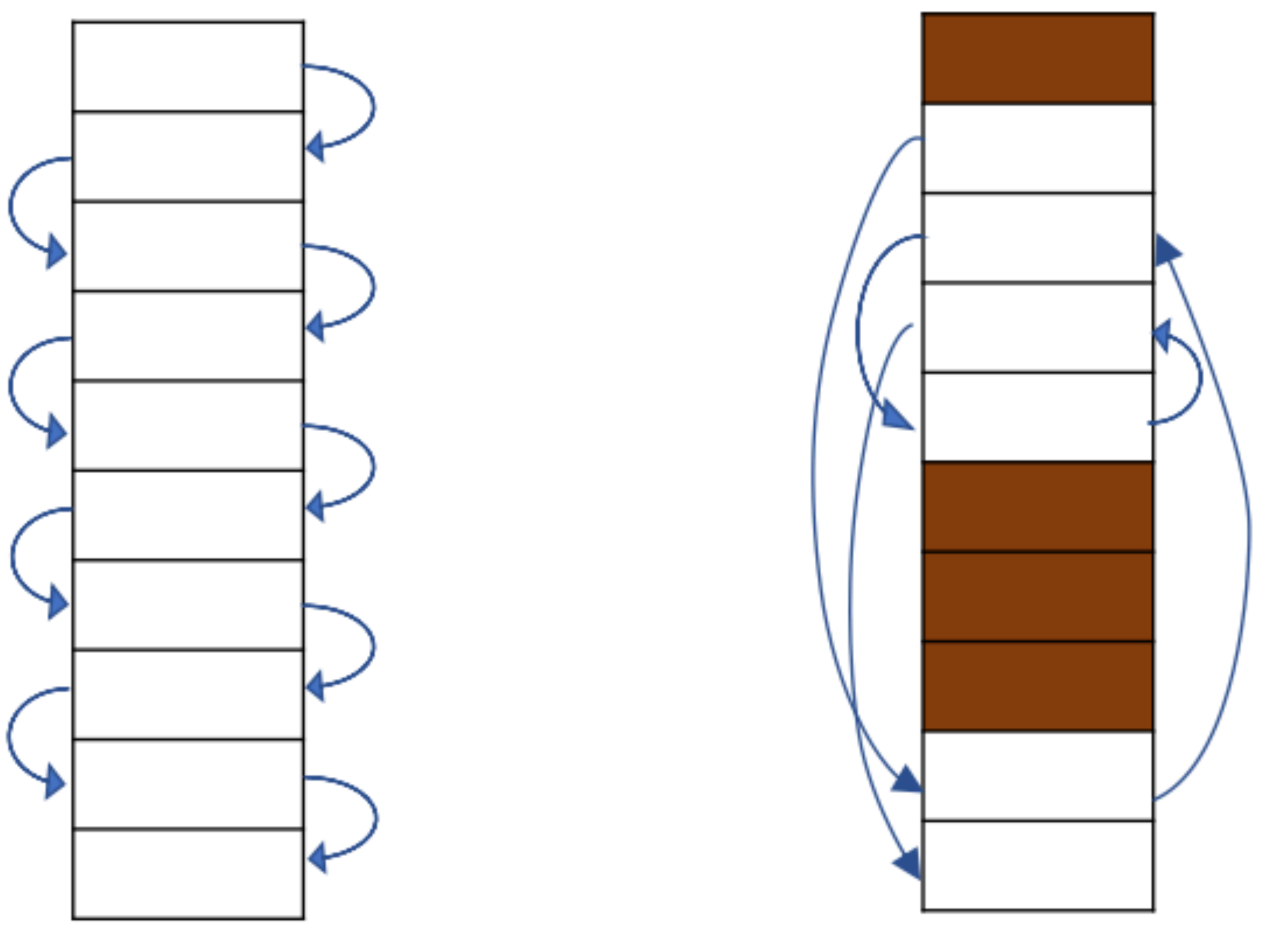
1. At first `fd1` is 5 KB and `fd2` is 2 KB

2. Append A to fd1
   `write(fd1, "A");`
   Then `fd1` is 6 KB

3. `Append A to fd1` again
   `write(fd1, "A");`
   - Not allowed to use fragments across multiple blocks
   - ✓ Copy old fragments to new block
   - ➡ Then `fd1` is 7 KB



Problem 2 - Unorganized Freelist



Initial performance good

Get worse over time

Measurement:

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

- Leads to random allocation of sequential file blocks overtime

Solution - Bitmaps

Periodical compact/defragment disk

- locks up disk bandwidth during operation

Keep adjacent free blocks together on freelist

- costly to maintain

➡ FFS - bitmap of free blocks (same idea as Indexed File System)

- Each bit indicates whether block is free
  e.g., 101010111111110000011111111000101100
- Easier to find contiguous blocks
- Small, so usually keep entire thing in memory
- Time to find free blocks increases if fewer free blocks

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

Using a Bitmap

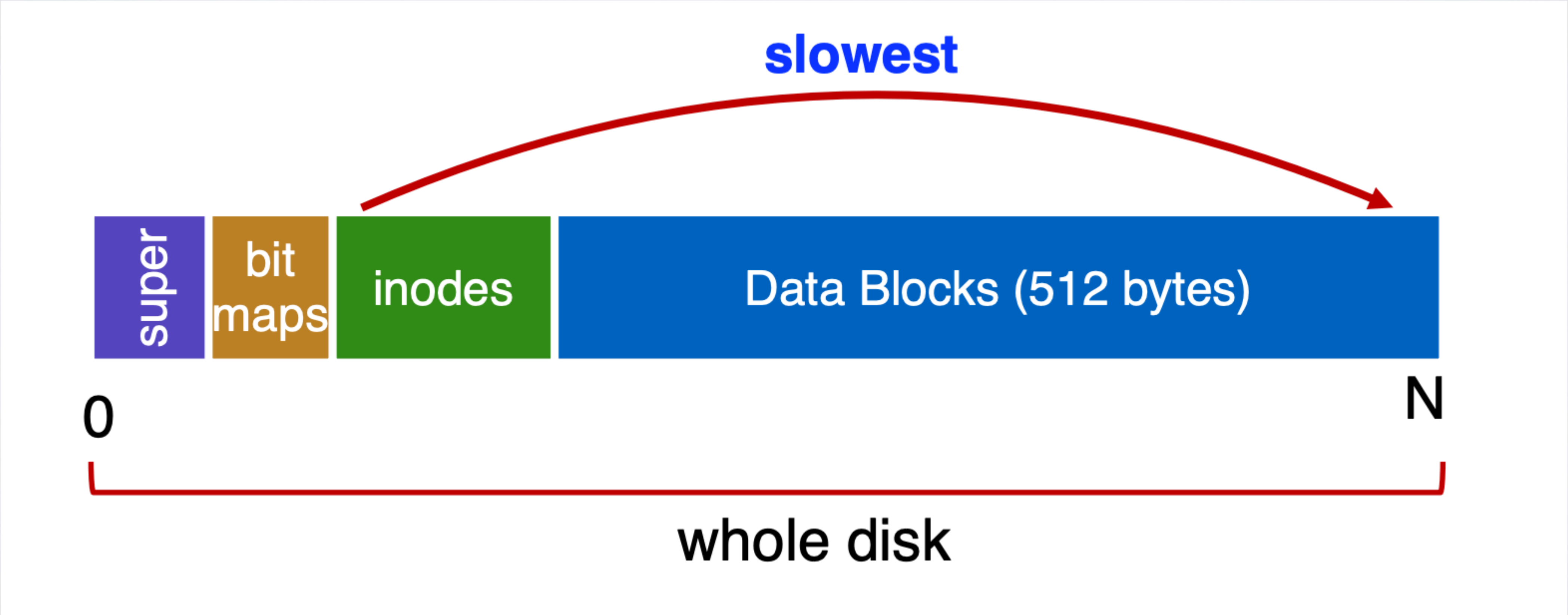
Allocate block close to block x

- Check for blocks near 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 (for hard disk drive)

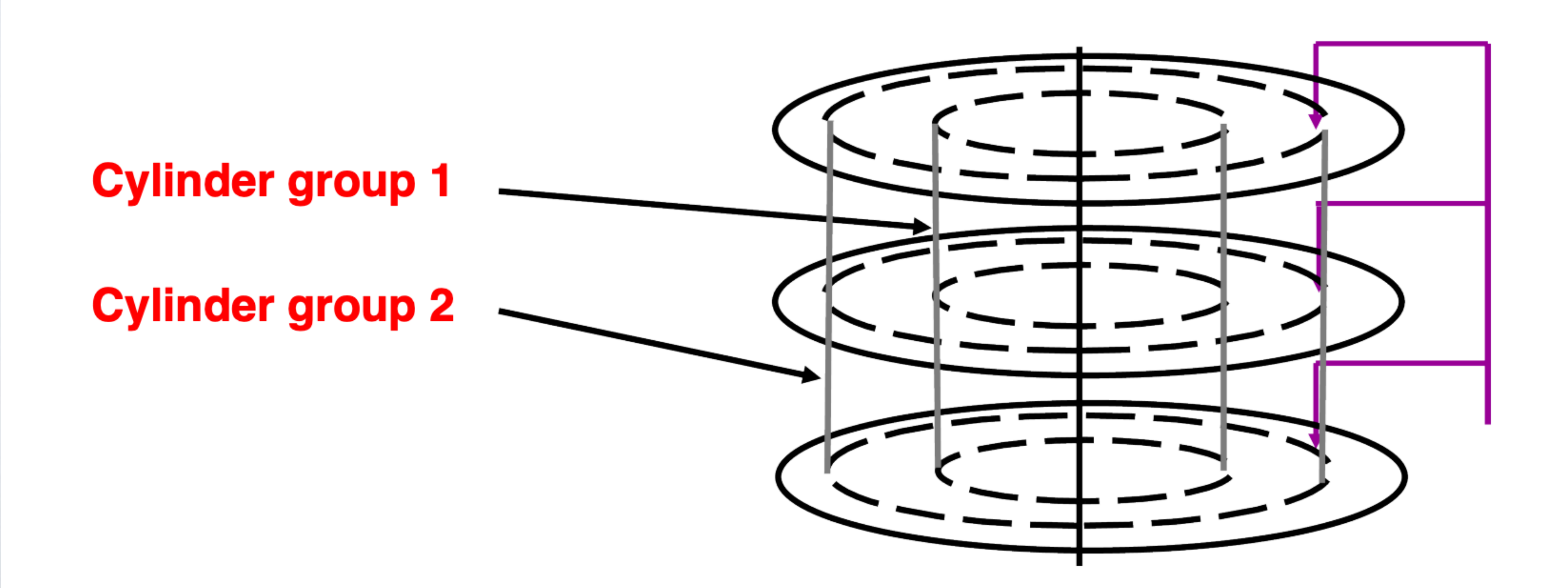
- How to keep inode close to data block?



FFS Solution - Cylinder Group

➡ Group sets of consecutive cylinders into "cylinder groups"

- 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

Access one block, probably access next

➡ Let's try to put sequential blocks in adjacent sectors

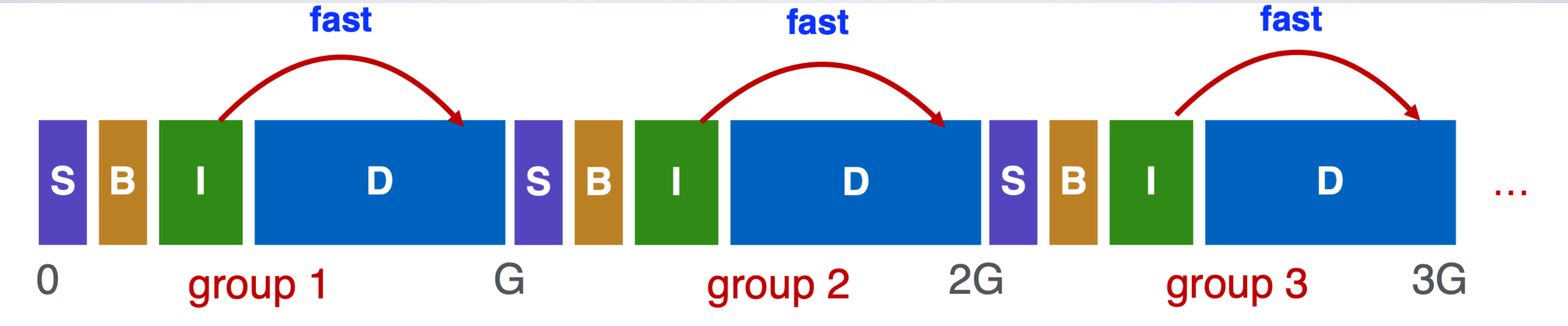
If you look at inode, most likely will look at data too

➡ Let's try to keep inode in same cylinder as file data

Access one name, frequently access many, e.g., "ls -l"

➡ Let's try to keep all inodes in a dir in same cylinder group

What Does Disk Layout Look Like Now?



How to keep inode close to data block?

➡ Use groups across disks and allocate inodes and data blocks in same group

✓ Each cylinder group basically a mini-Unix file system

Conclusion on FFS

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

Other enhancements

- Long file names
- Parameterization
- Free space reserve (10%) that only admin can allocate blocks from

Improving Reliability

with Log-Structured File system (LFS)
and Journaling File System (ext3)

What happen when power loss or system crash?

✓ Sectors (but not a block) are **written atomically** by the hard drive device

◉ But an FS operation might modify several sectors

- modify metada blocks (free bitmaps and inodes)
- modify data blocks

➡ Crash-consistency problem
a crash has a high chance of corrupting the file system

Solution 1 - Unix fsck (File System Checker)

When system boot, check system looking for inconsistencies

e.g. inode pointers and bitmaps, directory entries and inode reference counts

➡ Try to fix errors automatically

- Cannot fix all crash scenarios
- Poor performance
  - Sometimes takes hours to run on large disk volumes
  - Does fsck have to run upon every reboot?
- Not well-defined consistency

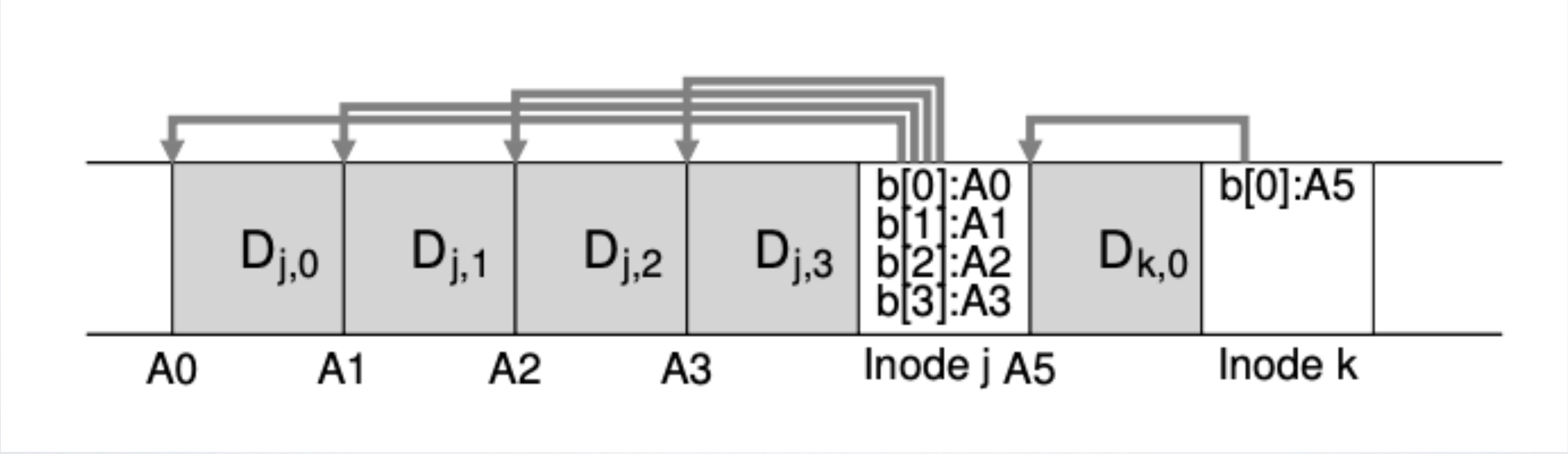
Solution 2 - Log Structure File System (LFS) or (Copy-On-Write Logging)

Idea - treat disk like a tape-drive

1. Buffer all data (including inode) in memory segment
2. Write buffered data to new segment on disk in a sequential log

➡ Existing data is not overwritten
Segment is always written in free location

✓ Best performance from disk for sequential access

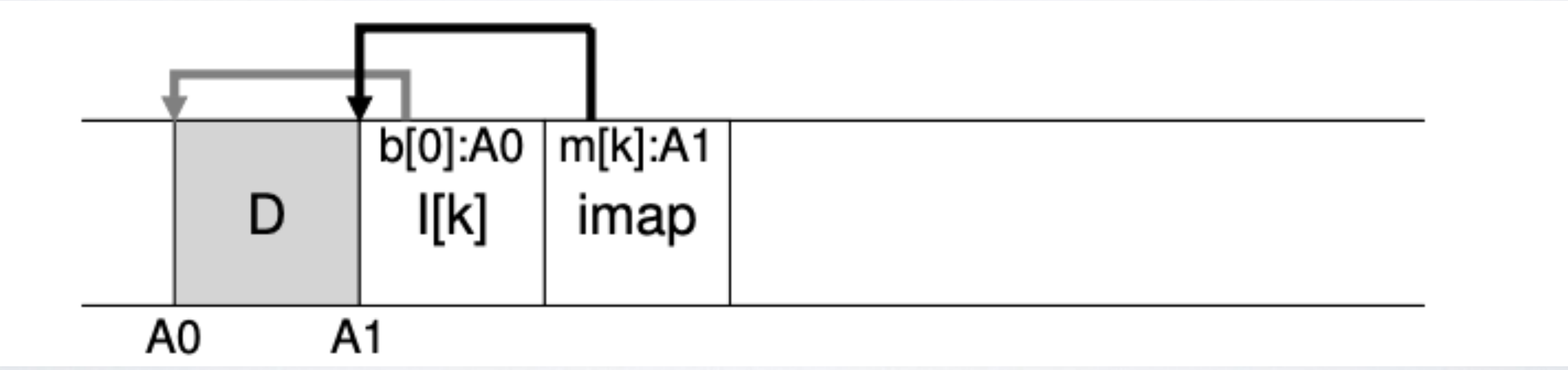


LFS - how to find the inode table?

Original Unix File System
the inode table is placed at fixed location

Log-structured File System
the inode table is split and spread-out on the disk

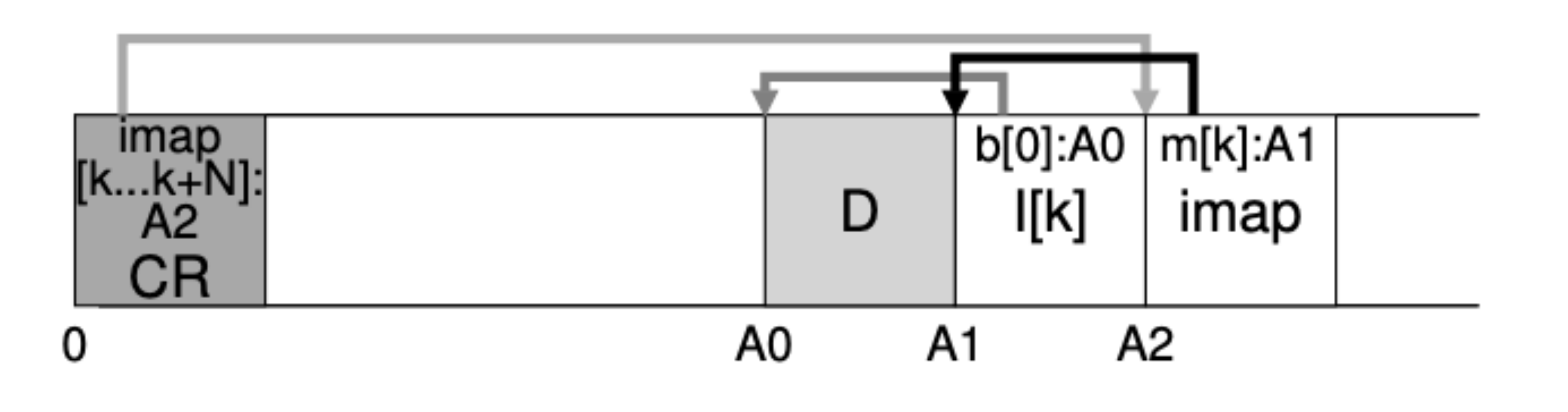
➡ LFS requires an inode map (imap) to map the inode number with its location on disk



LFS - how to find the inode map?

The OS must have some fixed and known location on disk to begin a file lookup

- ➡ The check-point region (CR) contains a pointer to the latest pieces of the inode map
- ✓ The CR is updated periodically (every 30 sec or so) to avoid degrading the performances



LFS - Crash recovery

The check-point region (CR) must be updated atomically

➡ LFS keeps two CRs and writing a CR is done in 3 steps

1. writes out the header with a timestamp #1
2. writes the body of the CR
3. writes one last block with another timestamp #2

✓ Crash can be detected if timestamp #1 is after #2

✓ LFS will always choose the most recent and valid CR

✓ All logs written after a successful CR update will be lost in case of a crash

LFS - Disk Cleaning (a.k.a Garbage Collection)

LFS leaves old version of file structures on disk

➡ LFS keeps information of the version of each segment and runs a disk cleaning process

- A cleaning process removes old versions by compacting contiguous blocks in memory
- That cleaning process runs when the disk is idle or when running out of disk space

Solution 3 - Journaling (or Write-Ahead Logging)

Write "intent" down to disk before updating file system

➡ Called the "Write Ahead Logging" or "journal" originated from database community

When crash occurs, look through log to see what was going on

- Use contents of log to fix file system structures
- The process is called "recovery"

Case Study - Linux Ext3

Physical journaling - write real block contents of the update to log

1. Commit dirty blocks to journal as one transaction (TxBegin, inodes, bitmaps and data blocks)
2. Write commit record (TxEnd)
3. Copy dirty blocks to real file system (checkpointing)
4. Reclaim the journal space for the transaction

Logical journaling - write logical record of the operation to log

- "Add entry F to directory data block D"
- Complex to implement
- May be faster and save disk space

Ext3 - What if there is a crash

- **Recovery** - Go through log and "redo" operations that have been successfully committed to log

What if …

- TxBegin but not TxEnd in log?
- TxBegin through TxEnd are in log, but inodes, bitmaps, and data have not yet been checkpointed?
- What if Tx is in log; inodes, bitmaps and data have been checkpointed; but Tx has not been freed from log?

Journaling Modes

Journaling has cost - one write = two disk writes (two seeks with hard disks)

➡ Several journaling modes balance consistency and performance

- **Data journalling** - journal all writes, including file data
  - expensive to journal data
- **Metadata journaling** - journal only metadata
  Used by most FS (IBM JFS, SGI XFS, NTFS)
  - file may contain garbage data
- **Ordered mode** - write file data to real FS first, then journal metadata
  Default mode for ext3
  - old file may contain new data