File System Implementation

Yajin Zhou (http://yajin.org)

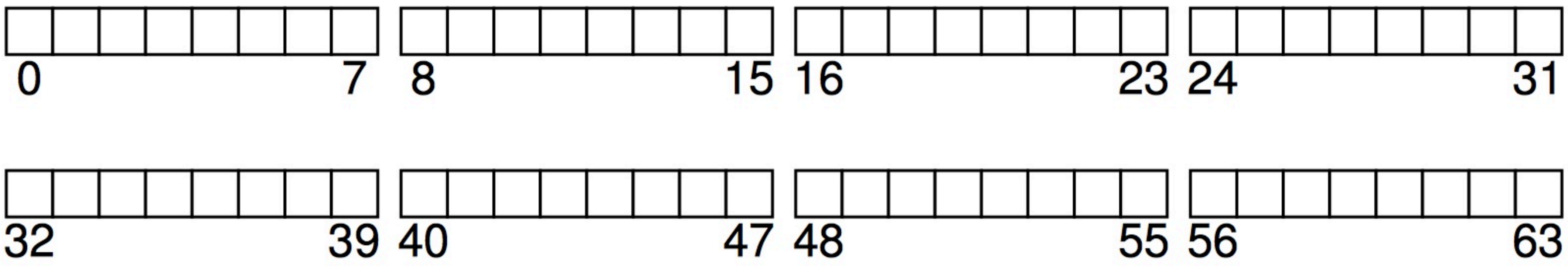
Zhejiang University

Two different aspects of FS

- (on-disk) Structure about FS
- Access methods
    - how does the FS maps the calls (open/read/write) onto its structures

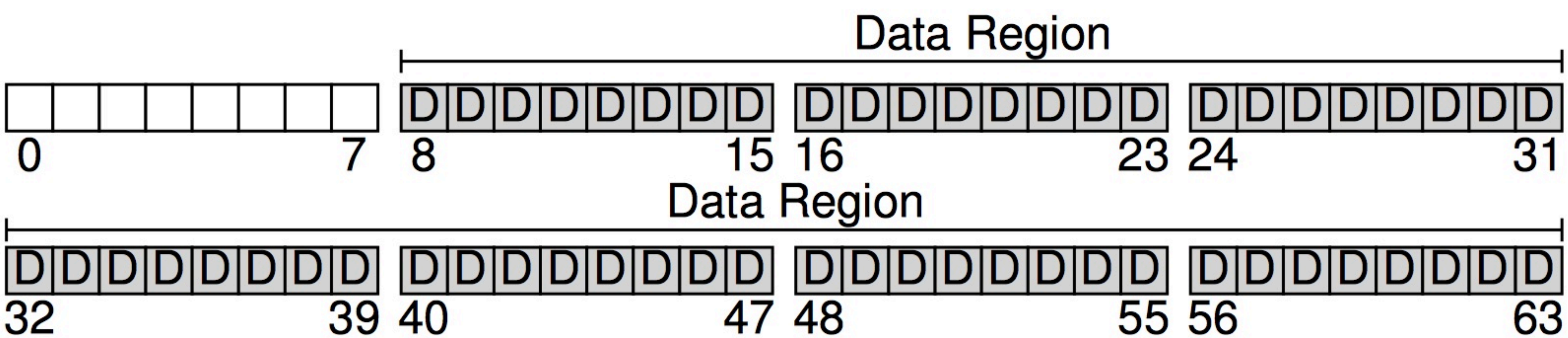
An Example

- Suppose we have a serial of blocks
  - Block size: 4k
  - 64 blocks



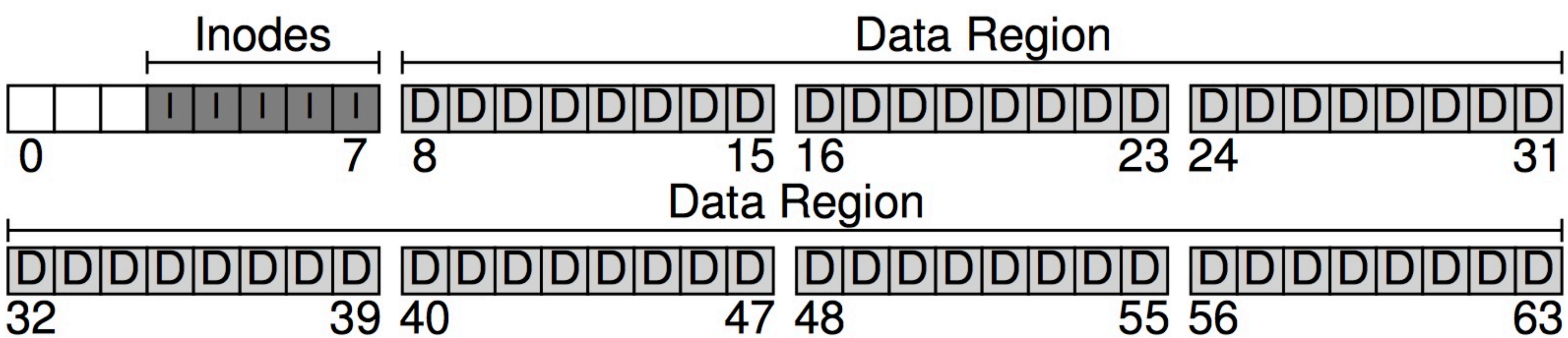
Data Region

- We reserve some blocks for data
  - 56 of 64 blocks



Inodes

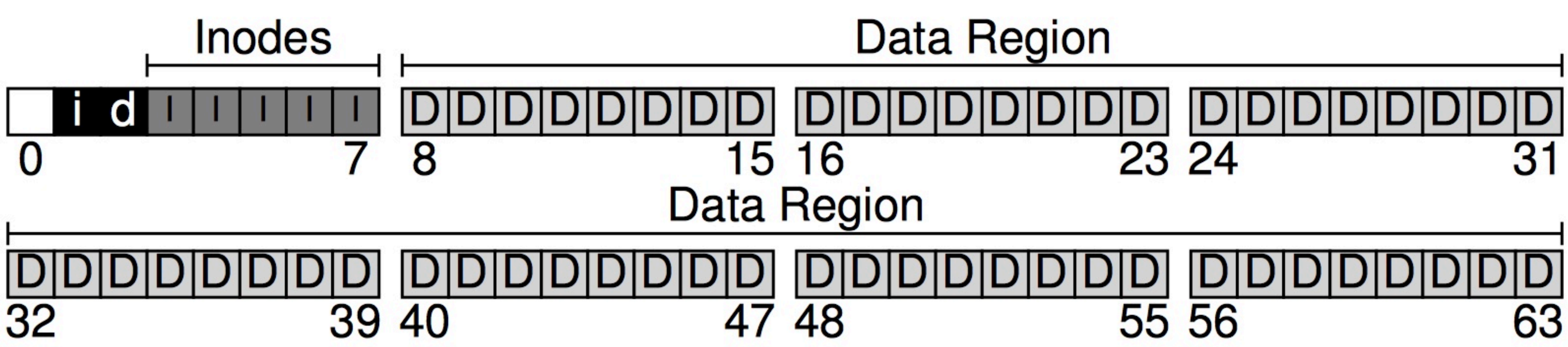
- Inode table: contains inodes
- 5 of 64 blocks are reserved for inodes
    - Suppose inodes are 256 bytes, 4 kb block can hold 16 inodes, then 5 blocks -> 80 inodes -> 80 files (directories)



D: Data block
I: inode

Bitmap

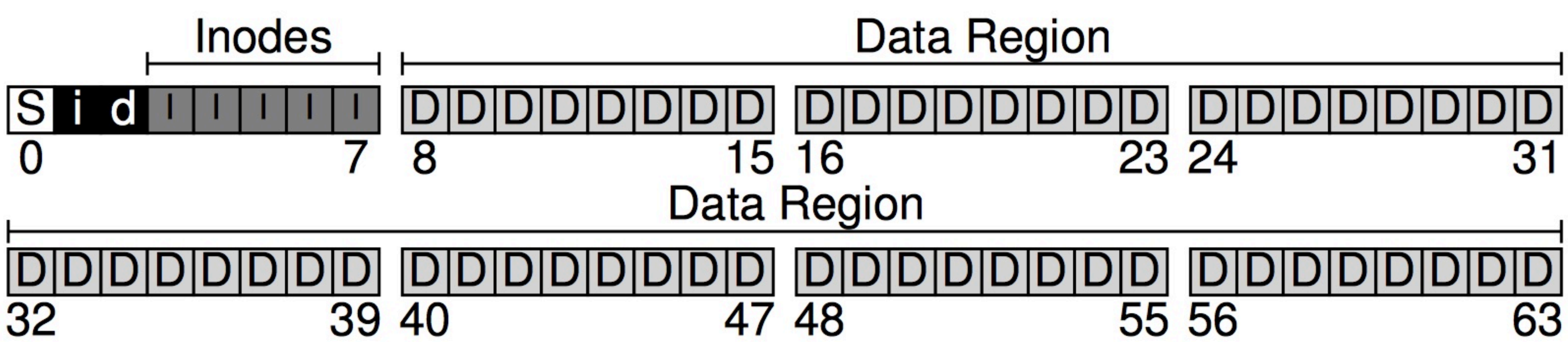
- Suppose we use bitmap to manage the free space
  - One bitmap for free inodes
  - One bitmap for free data region



D: Data block
I: inode
I:inode bitmap
d: data region bitmap

Superblock

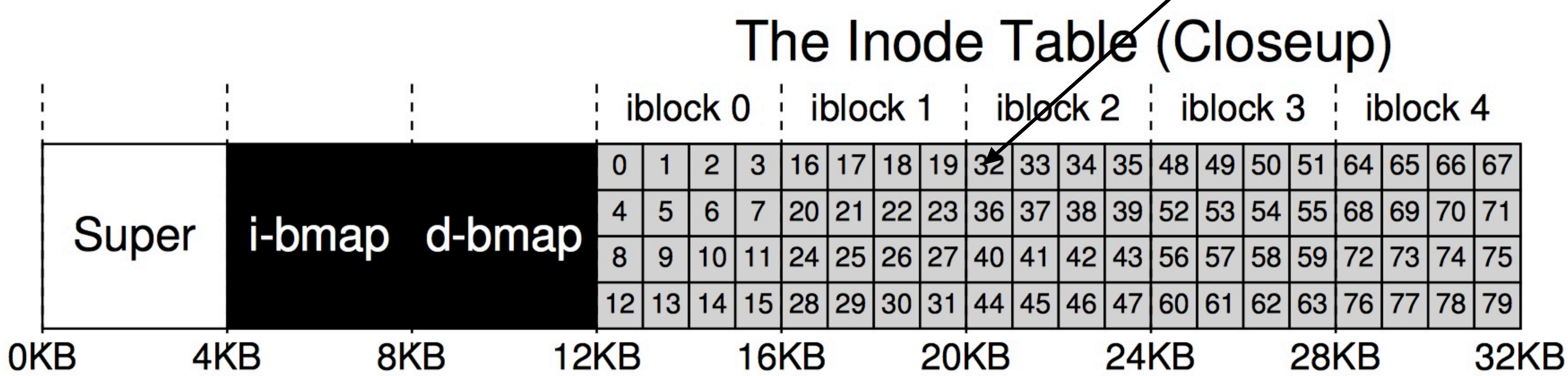
- Superblock
    - Contains information about this file system: how many inodes/ data blocks, where the inode table begins, where the data region begins, and **a magic number**



D: Data block
I: inode
I:inode bitmap
d: data region bitmap
S: superblock

Inode

- Each inode is identified by a number(inumber)
- To read inode number 32
    - 32 * sizeof(inode) = 8k
    - Address: 8k + 4k(super block) + 8Kk(BITMAP) = 20K



Ext2 Inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: **Simplified Ext2 Inode**

Multi-Level Index

- To support bigger file, we need multi-level index for the nodes

Directory Organization

- Suppose a dir (inode number 5) has 3 files: foo,bar, footer
- Name:
- Strlen: length of the name
- Reclen: length of the name plus **left over space (what's this?)**
    - For reuse the entry purpose

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Free Space Management

- Bit map
- Some OS will use the pre-allocation policy
  - For instance, when a file is created, a sequence of blocks (say 8) will be allocated
    - This can guarantee that the file on the disk is contiguous

Read /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

What about the system-wide/per-process open file table?

Write to Disk: /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			**read**							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					**read**					
	read									
	write									
								write		
					write					
write()					**read**					
	read									
	write									
									write	
					write					
write()					**read**					
	read									
	write									
										write
					write					

Caching and Buffering

- Without caching, each file open would require two reads for each level of the directory
    - One for the inode, and one for data
- Early system allocate a **fixed-size** cache to hold popular blocks
- Morden systems use a **unified page cache** for both virtual memory pages and file system pages
- Write buffering: does not write to disk immediately, instead sync to disk for like 5 - 30 seconds
- Database: direct IO with raw data

Log-structured File Systems

The Design and Implementation of a Log-Structured File System

Motivation

- System memories are growing:
  - can cache more data, disk operations are mostly write since read are serviced by the cache -> need to optimize write performance
- There is a large gap between random I/O and sequential I/O performance
  - Use the disk in sequential manner

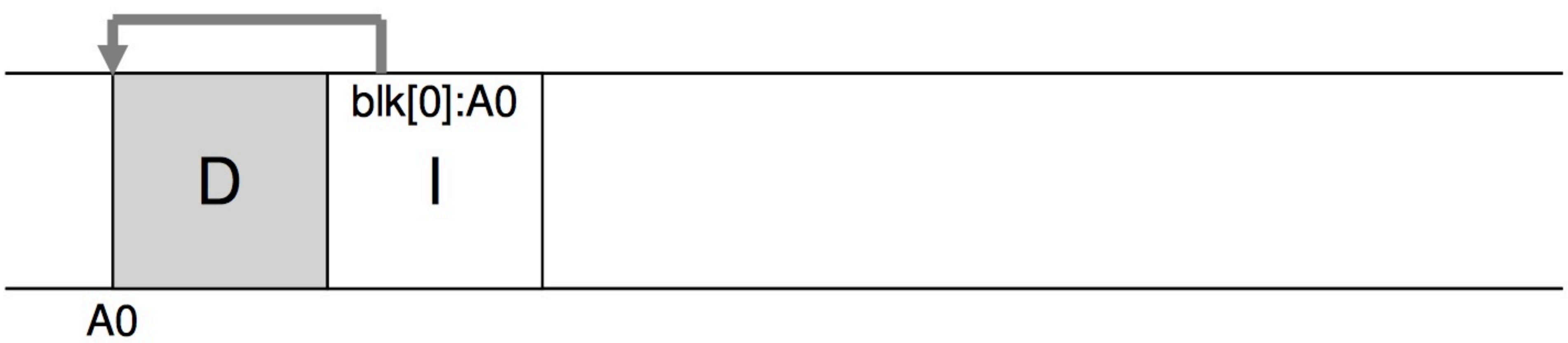
Idea: try to make use of the sequential bandwidth of the disk

LFS

- LFS: log-structured File System
- When writing to disk, LFS first buffers all updates (including metadata) into a **memory segment**; when the segment is full, it is written to disk in **one long and sequential transfer** to an **unused part** of the disk
- LFS **never overwrites existing data**, but rather always writes segments to free locations

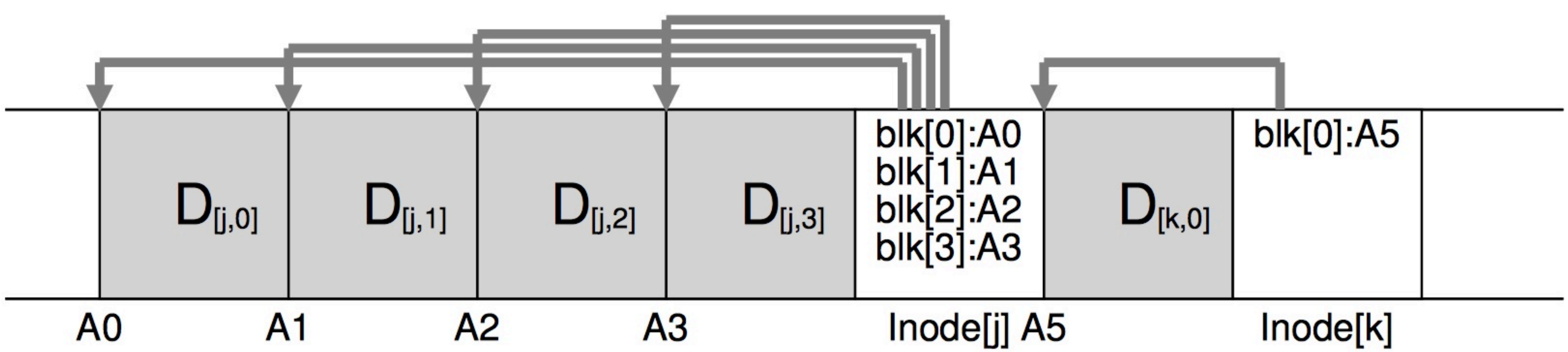
Writing To Disk Sequentially

- Write the data block and metadata into the disk
  - I: Inode



Write Buffering

- We can write to the disk when the **memory segment** is full
  - First is writing four blocks to file j
  - Second is one block being added to file k



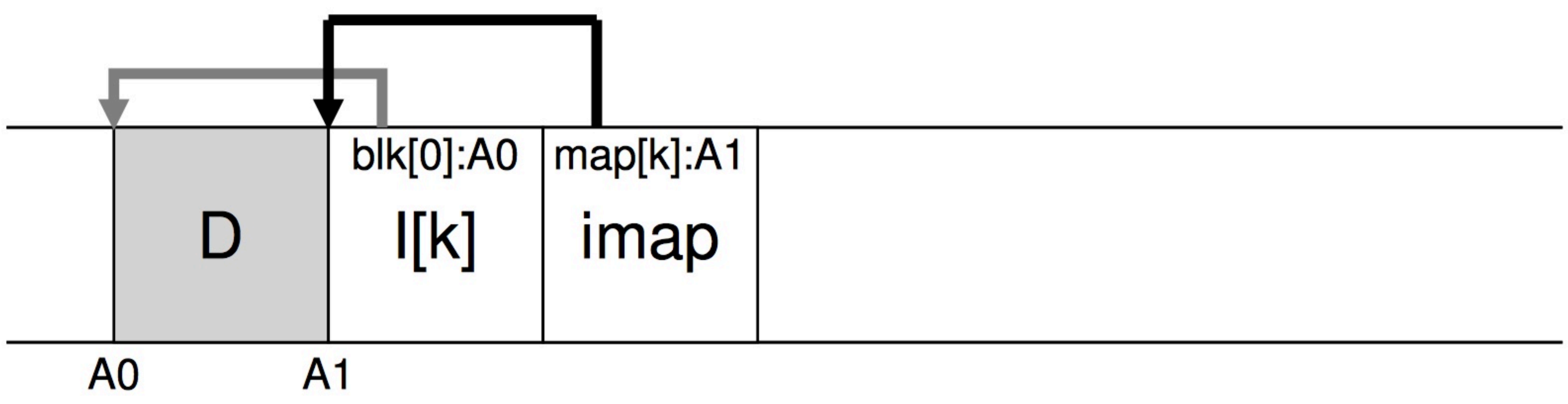
How to find inode?

Find Inode

- UNIX FS
    - Keep Inode in fixed locations
    - LFS: is hard
        - Inodes are scattered throughout the disk

Inode Map

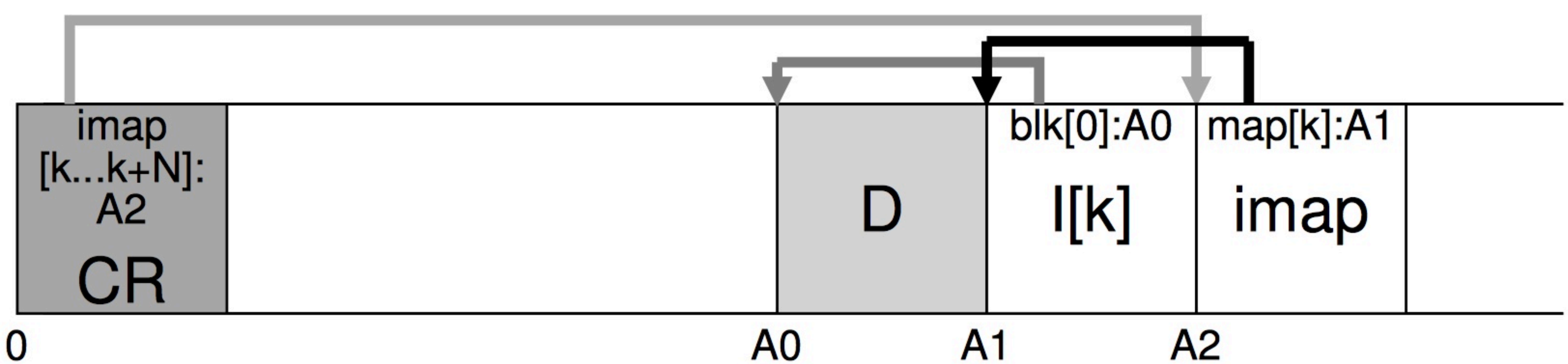
- Inode map(imap)
  - This map takes an inode number as input and produces the disk address of the most recent version of the inode
  - LFS places inode map right next to where it is writing all of the other new information



How to find imap?

Checkpoint Region

- Checkpoint region (CR)
    - Contains pointes to the latest pieces of the inode map
    - CR is updated periodically (say 30 seconds)



Directory is treated similar with file

Read

- Firs read CR
  - CR contains all the pointers to imap
- Read and cache imap
- Then given a inode number of a file, it looks up the imap to get the address of the data on the block
- Read data from block

Crash Recovery

Crash Recovery

- CR contains pointers to the head and tail segments
- Each segment points to next segment
- CR is updated periodically, 30s for example
- Segment is written into disk when it is full
- Crash can happen
  - Write to a segment
  - Write to the CR

Write to CR

- LFS keeps to CRs, and write to them alternatively
- Write protocol
  - First writes out header (with timestamp), then body, and last the one last block (with timestamp)
- If crash happen when writing CR, LFS can detect this by detecting the inconsistent of the timestamps,
- LFS always chooses to use the most recent CRT with consistent timestamps

Write to a segment

- If crash happens, then the CR has not been written into disk
- roll forward
  - Start with the last checkpoint, and find the end of the log, and then use that to find next segment and see if there are any new updates
  - Use this to recovery the data