OPERATING SYSTEMS

File Systems

Presented by NTR

File System Concept

- The file system reside permanently on secondary storage, which is desired to hold a large amount of data permanently.
- Main memory is usually too small and can not accommodate data and programs permanently.
- So, the computer system must provide secondary storage to back up main memory.
- Most computers use disk as the most common secondary storage device for both programs and data.

Magnetic Disks/Hard Disks Structure Overview

- Provide the bulk of secondary storage for modern computer systems.
- Store data permanently.
- Hard disk architecture consists of:
  - Platter
  - Spindle
  - Read-write head
  - Track

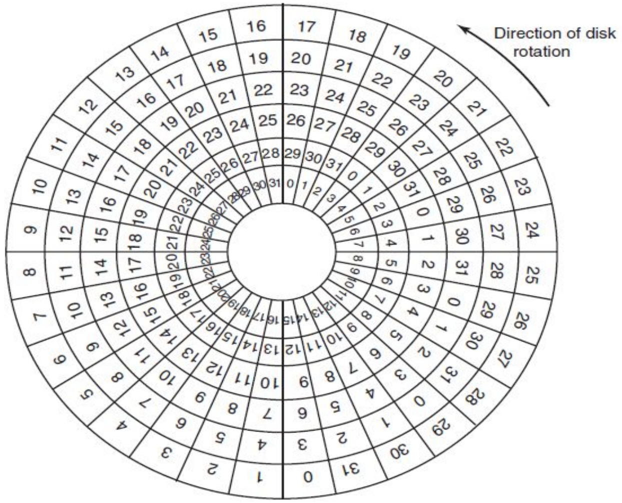


Figure 2: An illustration of platter, tracks and sectors.

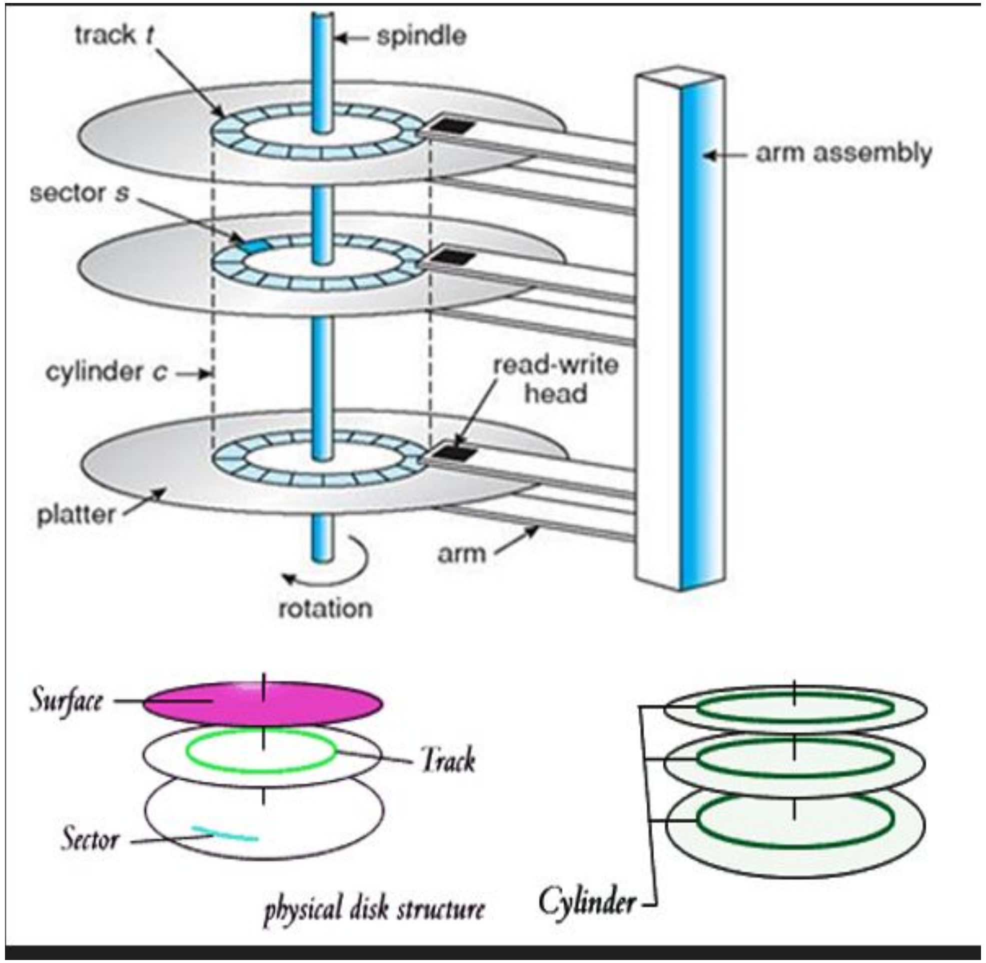


Figure 1: Moving-head disk mechanism.

Magnetic Disks/Hard Disks Structure Overview

- Disk has multiple platters having circular shape like a CD.
- Platters have 2 surfaces upper and lower. Both surfaces are being used.
- All the platters are connected with a spindle.
- Spindle rotates all platters together unidirectionally in a centric way either clockwise or anti clockwise.
- When spindle moves all platters move together.
- Actuator arms named read-write heads are connected with every platter.
- Read-write heads are connected with both surfaces of platters. Which means a platter having upper and lower surfaces consists of two separate heads for two surfaces.

Magnetic Disks/Hard Disks Structure Overview (cont.)

- Purpose of read-write heads is fetching data.
- It moves back and forth (backward and forward) in order to read data. As separate heads are connected to every surfaces of a platter that means heads can read data from both surfaces.
- Upper and lower both surfaces of a platters consist of same number of multiple tracks.
- The outer most track is known as external track and the inner most is known as internal track.
- If the heads requires to fetch data from an inner track from its current position, then it needs to move forward.
- If the heads requires to fetch data from an outer track from its current position, then it needs to move backward.
- Each track (both upper and lower surface) is divided into fixed number of multiple sectors.
- Data is being stored in sectors.

File System Concept

- The file system provides the mechanism for:
  - Storage of data and programs on the disk.
  - Access to data and programs on the disk

- Files are mapped by OS to physical devices.
  - A collection of related information defined by its creator.
  - Files are normally organized into directories for ease of use.
- Definition of file: A file is a named collection of related information that is recorded on secondary storage.
  - Data cannot be written to secondary storage unless they are within a file.
  - Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator
  - Many types, Consider: **text file, source file, executable file**

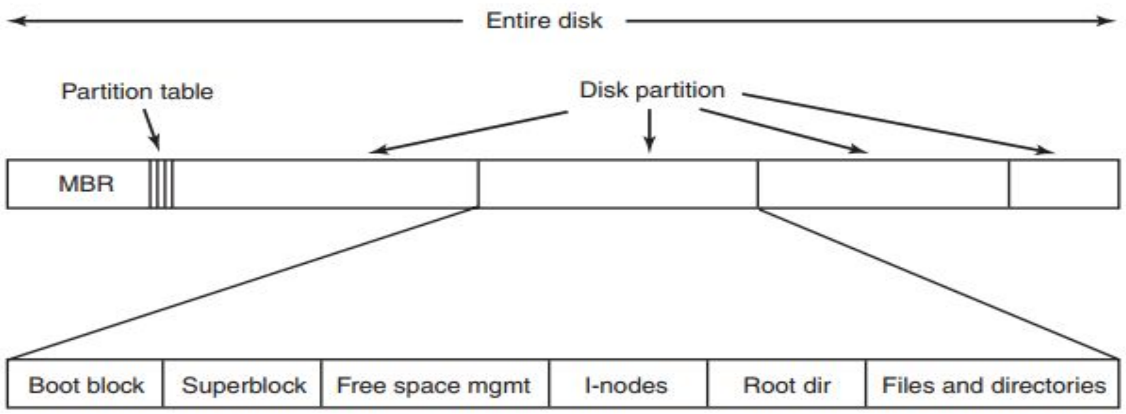
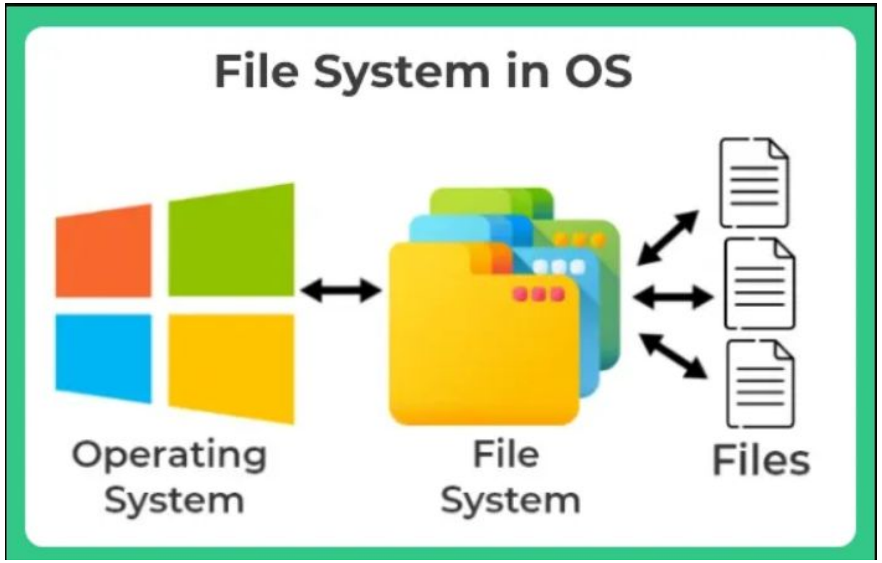


Figure: A possible file-system layout.

File System Concept

- It is one of the major functionalities of OS.
- It is a software component of OS which manages all files.
- The file manager administers the file system by:
  - Storing the information on a device
  - Mapping the block storage to a logical view
  - Allocating/deallocating storage
  - Providing directories



- Different OS has different file systems.
- Windows has NTFS, FAT32 etc. file systems, Mac OS has apple file system (APFS), Unix has unix file system (UFS), Linux has extended file systems etc.
- It determines how user data in form of files gets stores in the secondary memory or disk permanently and how these data can be fetched from the disk.

File System Concept (cont.)

- In an OS every data from user end is managed in form of files. As example: files of any formats like mp3, mkv, pdf, docx etc.
- Users creates a file in a particular folder or directory and can work on that file. They can modify data of that file and save changes on file. These actions can be performed from user perspective.
- Based on actions of a user file system manages files in the disk.
- In order to manage files in the disk, file system divides each file into multiple equal sized blocks logically which are known as logical blocks.
- Logical blocks are always equivalent to the power of 2 (2^n).
- Then logical blocks are mapped into corresponding physical blocks or sectors in the hard disk and that is how data is being stored in a hard disk.

File Attributes

- Every file in the disk has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. These information about the file is known as attributes of the file or metadata. The list of attributes varies considerably from system to system.
- Information about files are kept in the directory structure, which is maintained on the disk.
- Some of common attributes of files maintained by almost every OS are discussed below:
  - **Name:** The symbolic file name is the only information kept in human readable form.
  - **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
  - **Type or Extensions:** This information is needed for systems that support different types of files.
  - **Location:** This information is a pointer to a device and to the location of the file on that device.
  - **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
  - **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
  - **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Attributes

Figure 11.1 A file info window on Mac OS X.

File Types

- Windows (and some other systems) use special file extensions to indicate the type of each file:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

File Operations

- **Creating:** Creates a file with all of its attributes or metadata.
- **Reading:** Reads data from a file.
- **Writing:** Edits or modifies a file.
- **Deleting:** Removes a file alongside its all attributes.
- **Truncating:** Removes information from a file. It does not remove attribute of the file.
- **Repositioning:** Seek file.

OPERATING SYSTEMS

File Allocation Methods

File Allocation Methods

- After the creation of a file, file system allocates it in two steps.
- Firstly, it divides the file into multiple equally sized logical blocks. Size of each logical block is equivalent to the power of 2 (2^n).
- Then, in order to store data of each block permanently in the disk it maps logical blocks to corresponding physical blocks or sectors and allocates the logical blocks in the corresponding sectors. Size of logical blocks and sectors are always equal.
- Mechanisms used for storing a logical block into its corresponding sector by the file system are known as allocation methods.
- Allocation methods are of two types.
  - Contiguous allocation
  - Non-contiguous allocation
    - Linked allocation
    - Indexed allocation
    - UNIX inode

OPERATING SYSTEMS

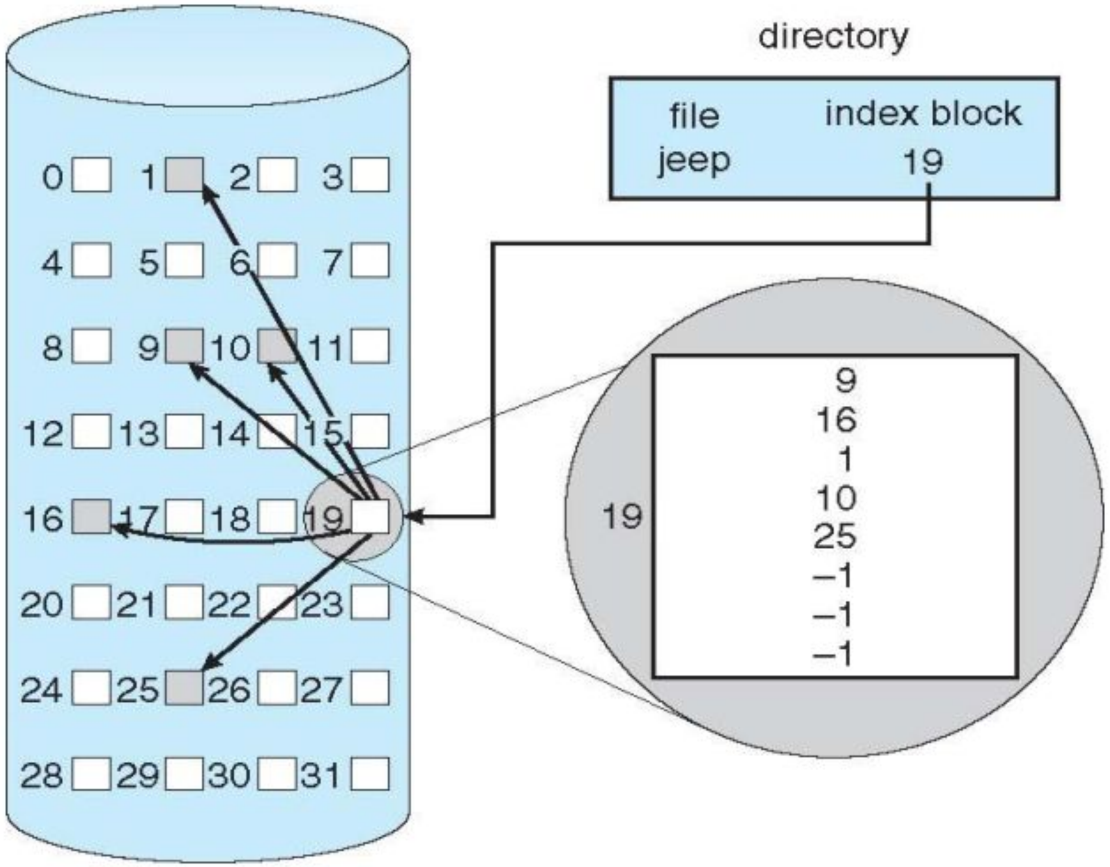
Non-Contiguous Allocation

Indexed Allocation

- It is also non-contagious allocation but support efficient direct access
- In index allocation bring all the pointers together into one block: the index block
- Name of a file alongside its attributes and index block number can be found from the directory where the file is stored.
- By accessing the index block number of a file information about its blocks and information pointers to their corresponding sectors or physical blocks are found.
- Each file has one or multiple index block/s in the directory.
- Multiple index blocks are maintained by a file when the file is of a very large size.
- Information of all index blocks of all files is managed by file-allocation table which is stored in the main memory.

Instead of link list structure like link allocation, the file directory content the address of first block and the first block stores the sequential list of all blocks

Indexed Allocation



32/4=8
32*5=160bytes

Figure: Indexed allocation of disk space.

Indexed Allocation

Benefits:

- Supports random or direct access.
- No external fragmentations.

Drawbacks:

- Pointer overhead.
- Multilevel index when file is too large.

UNIX Inode

- Variant of indexed allocation.
- I means indexed and node means block.
- It maintains multilevel index.
- UNIX inode is a custom data structure consists of file attributes, direct blocks, single indirect, double indirect and triple indirect fields.
- **Direct blocks:** Stores such data blocks where each block stores pointer to a block where data is stored.
- **Single indirect:** Stores such data blocks where each block stores pointer to a direct block.
- **Double indirect:** Stores such data blocks where each block stores pointer to a single indirect.
- **Triple indirect:** Stores such data blocks where each block stores pointer to a double indirect.

UNIX Inode is similar to directory

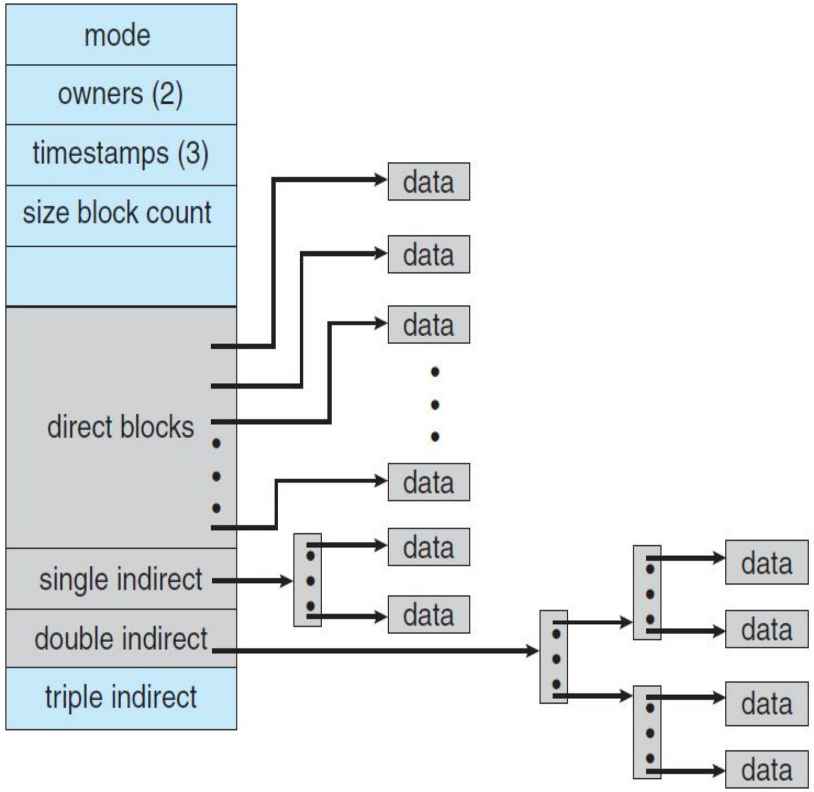


Figure: The UNIX inode.

UNIX Inode

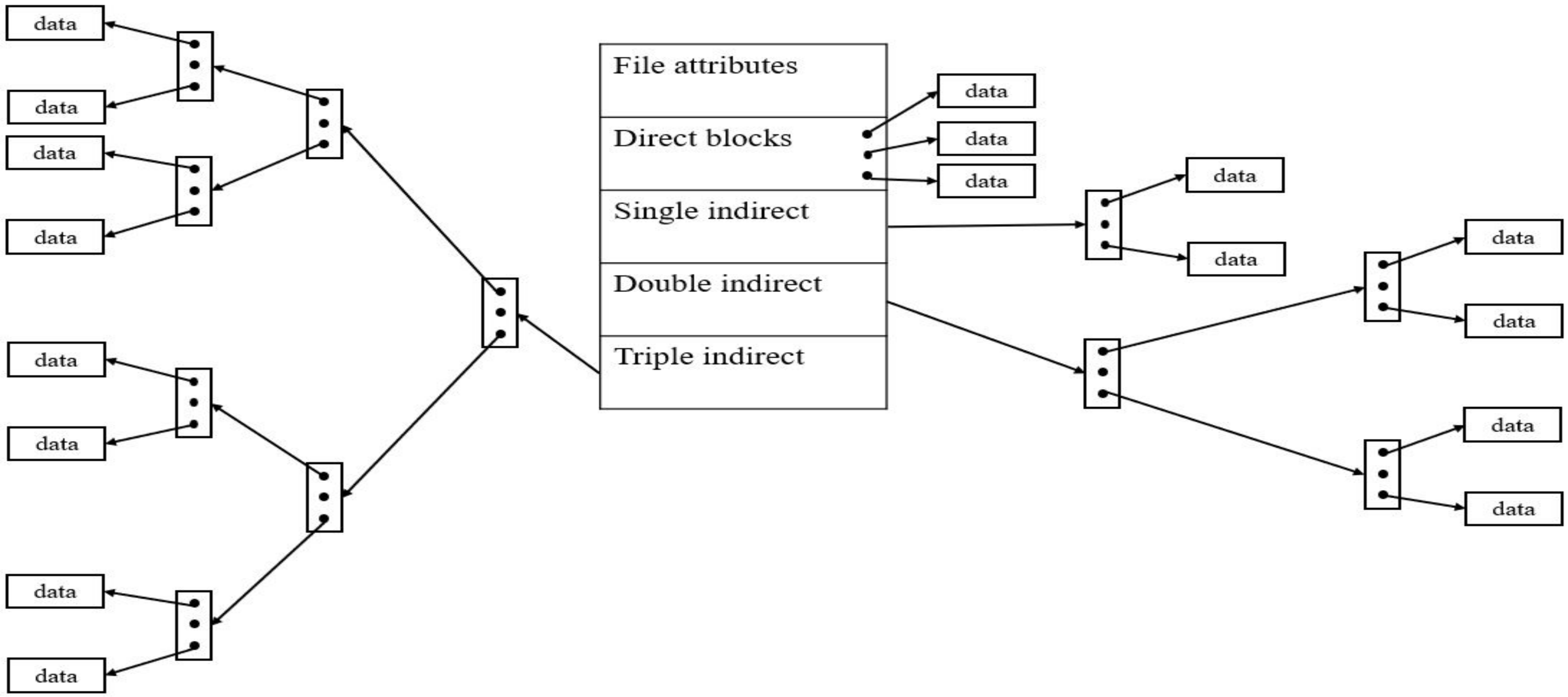


Figure: The UNIX inode with detailed triple indirect.

UNIX Inode

Problem:

A file system uses UNIX inode data structure which contains 4 direct block addresses, 1 single indirect block, 1 double indirect block and 1 triple indirect block. Size of each block is 64 Bytes and size of each block address is 4 Bytes. Find the maximum possible file size?

Solution:

Total number of pointers in one data block = $\dfrac{\text{size of each data block}}{\text{size of each block address or pointer}}$

$= {}^{64}/_{4} = 16$

Total number of pointers = # of pointers in direct blocks + # of pointers in single indirect + # of pointers in double indirect + # of pointers in triple indirect

$= (4 + 16 + 16^2 + 16^3) = 4372$

Maximum file size = # of total pointers $\times$ block size

$= 4372 \times 64$

$= 279808\ Byte = 273.25\ KB$

UNIX Inode

Problem:

A file system uses UNIX inode data structure which contains 4 direct block addresses, 1 single indirect block, 1 double indirect block and 1 triple indirect block. Size of each block is 64 Bytes and size of each block address is 4 Bytes. Find the maximum possible file size?

Solution:

Total number of pointers in one data block = $\dfrac{\text{size of each data block}}{\text{size of each block address or pointer}}$

$= {}^{64}/_{4} = 16$

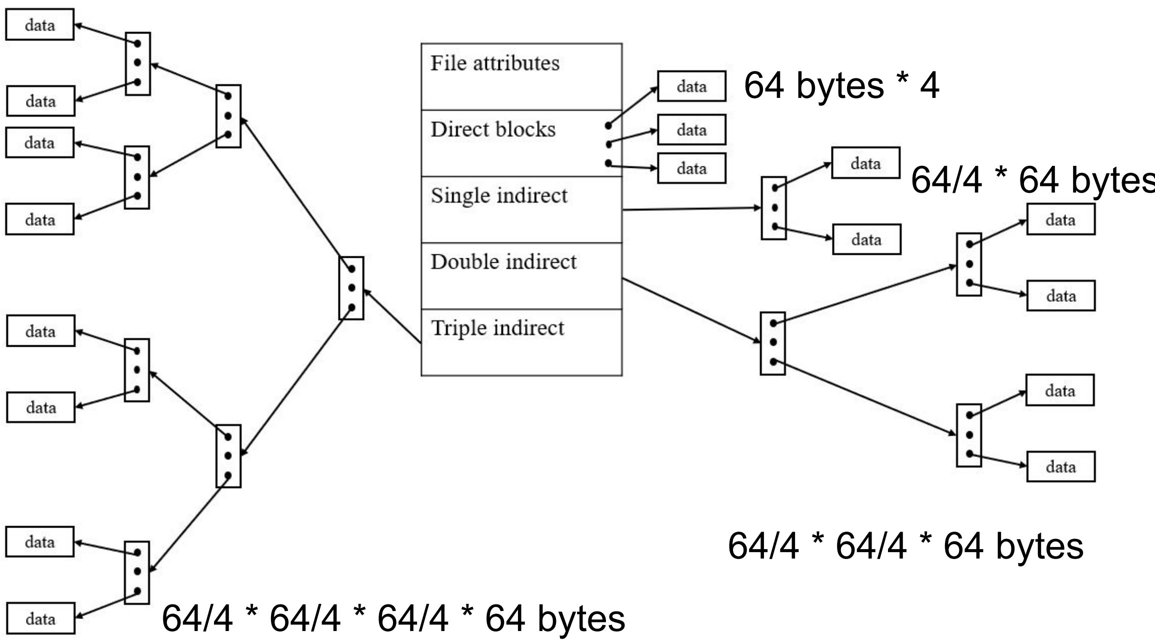
Total number of pointers = # of pointers in direct blocks + # of pointers in single indir
indirect

$= (4 + 16 + 16^2 + 16^3) = 4372$

Maximum file size = # of total pointers $\times$ block size

$= 4372 \times 64$

$= 279808\ Byte = 273.25\ KB$



File System Implementation

The Way to Think

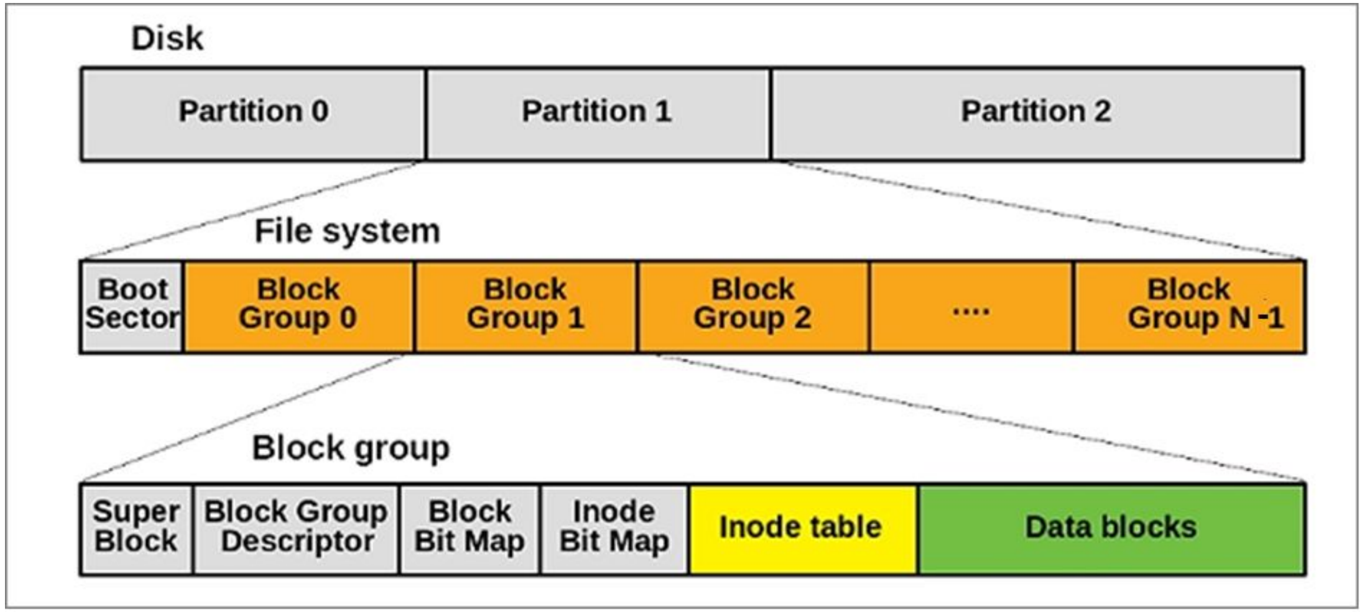
There are two different aspects to implement file system

- Data structures
  - What types of on-disk structures are utilized by the file system to organize its data and metadata or file attributes?

- Access methods
  - How does it map the system calls or operations made by a process as open(), read(), write(), etc.
  - Which structures are read during the execution of a particular system call?

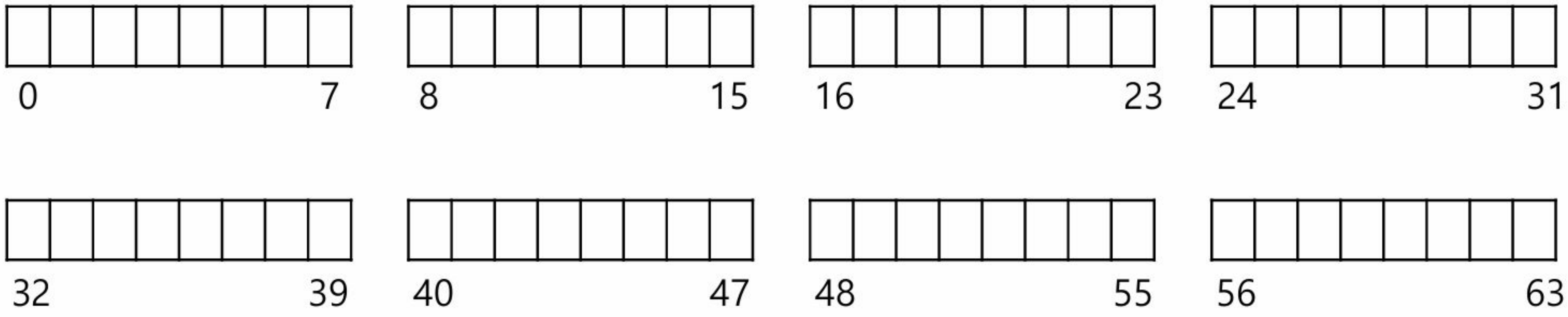
Ext2 file System

- The Ext2 file system organizes data on disk using block groups, which are contiguous sequences of blocks. These blocks are then grouped into larger units called block groups, which contain metadata and data blocks. This structure allows for efficient data access and management, especially for large files.



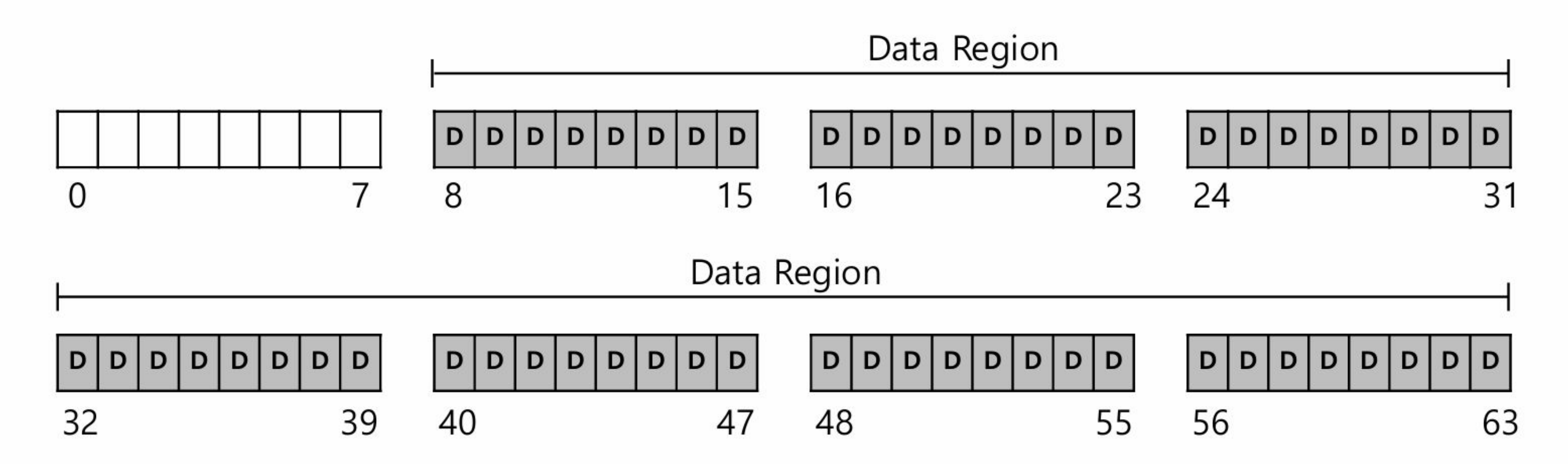
Overall Organization of VSFS (Very Simple File System [ext2])

- Let’s develop the overall organization of the file system data structure.
- Divide the disk into data blocks [data block numbers are logical or virtual addresses to the physical sectors].
    - Each block size is 4 KB.
    - The blocks are addressed from 0 to N -1.



Data region in file system

- Reserve data region to store user data



  - File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

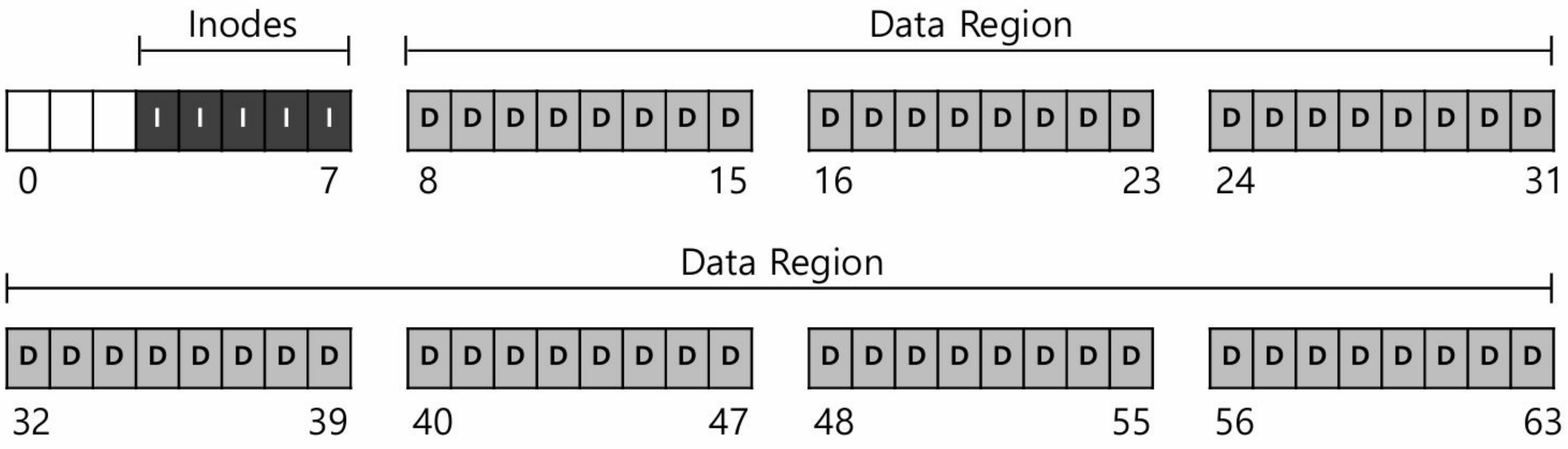
Ext2 Layout

Super Block	Block Group Descriptor	Block Bit Map	Inode Bit Map	Inode table	Data blocks

Block Offset	Length	Description
byte 0	512 bytes	boot record (if present)
byte 512	512 bytes	additional boot record data (if present)
block group 0, blocks 1 to 8192		
byte 1024	1024 bytes	superblock
block 2	1 block	block group descriptor table
block 3	1 block	block bitmap
block 4	1 block	inode bitmap
block 5	214 blocks	inode table
block 219	7974 blocks	data blocks
block group 1, blocks 8193 to 16384		
block 8193	1 block	superblock backup (Rev 0 has 1 every block group)
block 8194	1 block	block group descriptor table backup (Rev 0 has 1 every block group)
block 8195	1 block	block bitmap
block 8196	1 block	inode bitmap
block 8197	214 blocks	inode table
block 8408	7974 blocks	data blocks

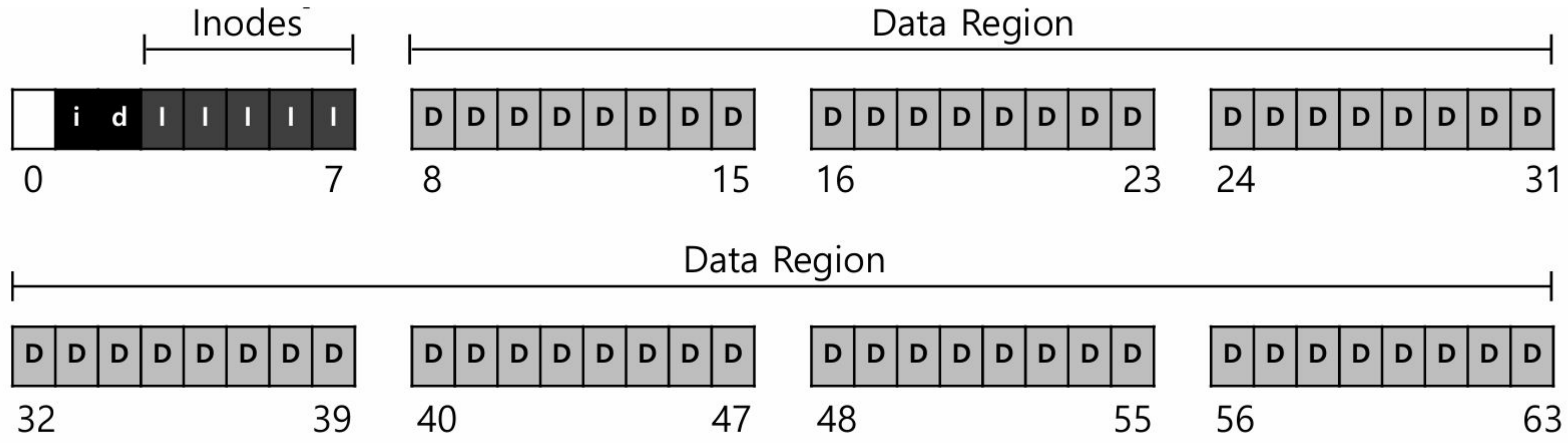
Inode table in file system

- Reserve some space for inode table
  - This holds an array of on-disk inodes.
  - Ex) inode tables : 3 ~ 7, inode size : 256 bytes
    - 4-KB block can hold 16 inodes.
    - The filesystem contains 80 inodes. (maximum number of files)



Allocation Structures

- This is to track whether inodes or data blocks are free or allocated.
- Use bitmap, each bit indicates free(0) or in-use(1)
  - **data bitmap**: for data region
  - **inode bitmap**: for inode table



Superblock

- Super block contains this information for particular file system or the metadata of the file system.
  - Ex) The number of inodes, begin location of inode table. etc

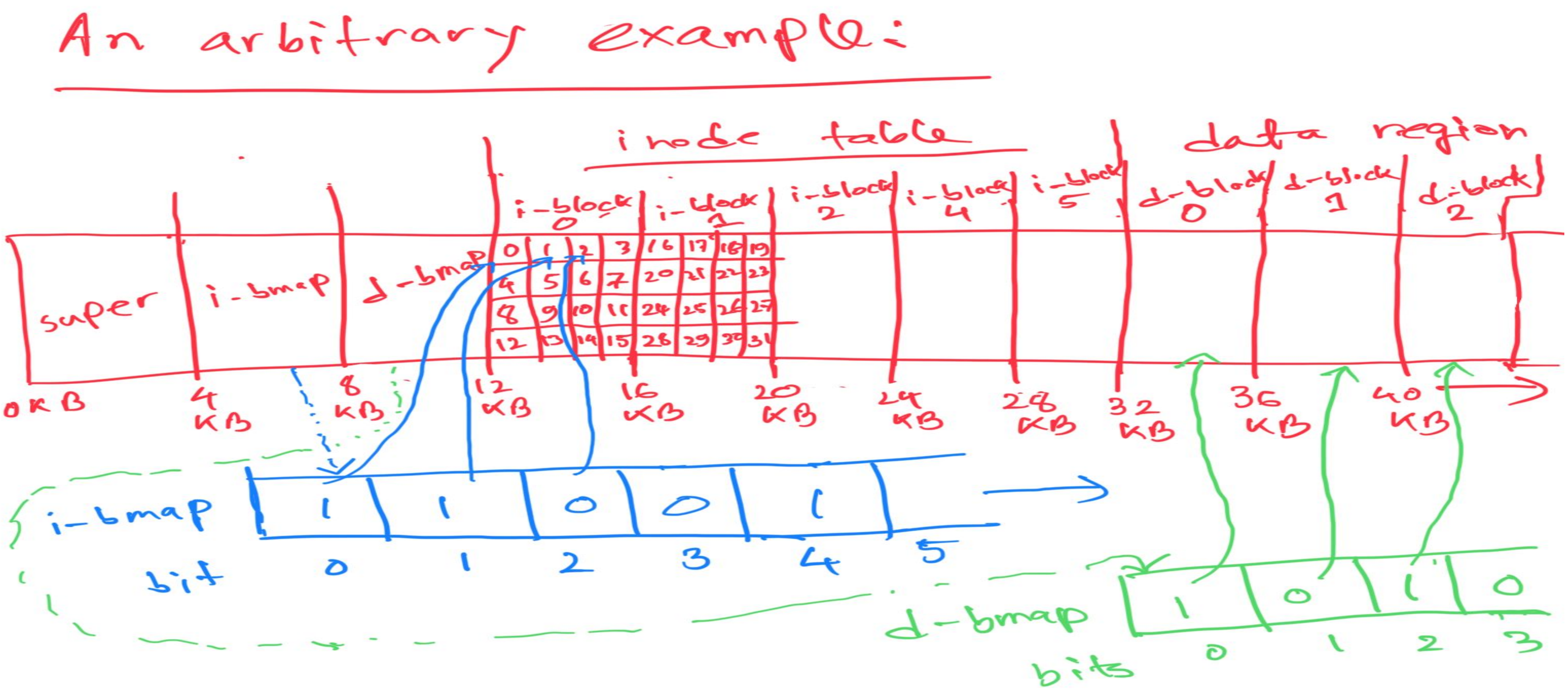
  - Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

Super Block

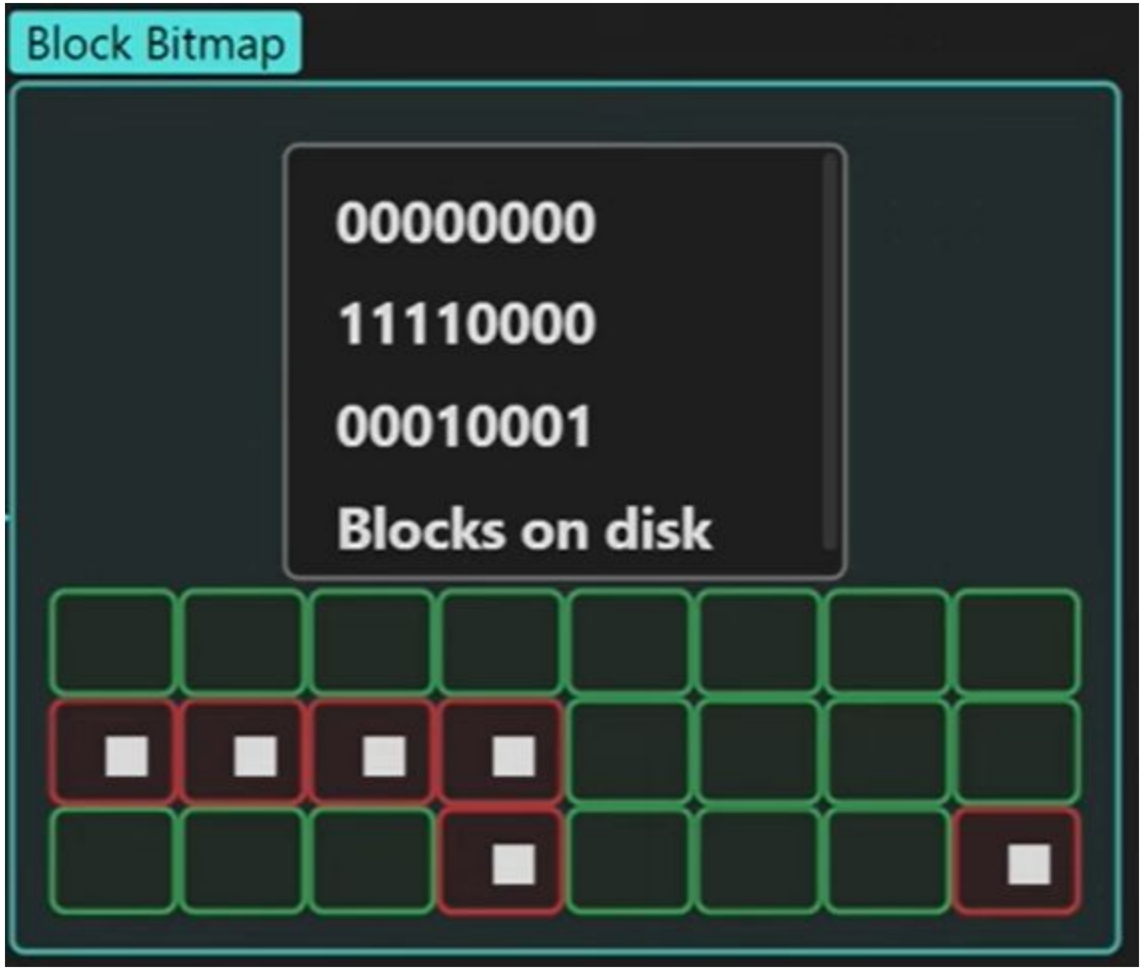
Superblock

bytes Offset	size bytes	Description
0	4	s_inodes_count
4	4	s_blocks_count
8	4	s_r_blocks_count
12	4	s_free_blocks_count
16	4	s_free_inodes_count
20	4	s_first_data_block
24	4	s_log_block_size
28	4	s_log_frag_size
32	4	s_blocks_per_group
36	4	s_frags_per_group
40	4	s_inodes_per_group
44	4	s_mtime
48	4	s_wtime
52	2	s_mnt_count
54	2	s_max_mnt_count
56	2	s_magic
58	2	s_state
60	2	s_errors
62	2	s_minor_rev_level
64	4	s_lastcheck
68	4	s_checkinterval
72	4	s_creator_os
76	4	s_rev_level
80	2	s_def_resuid
82	2	s_def_resgid
84	940	Reserve for upgrade

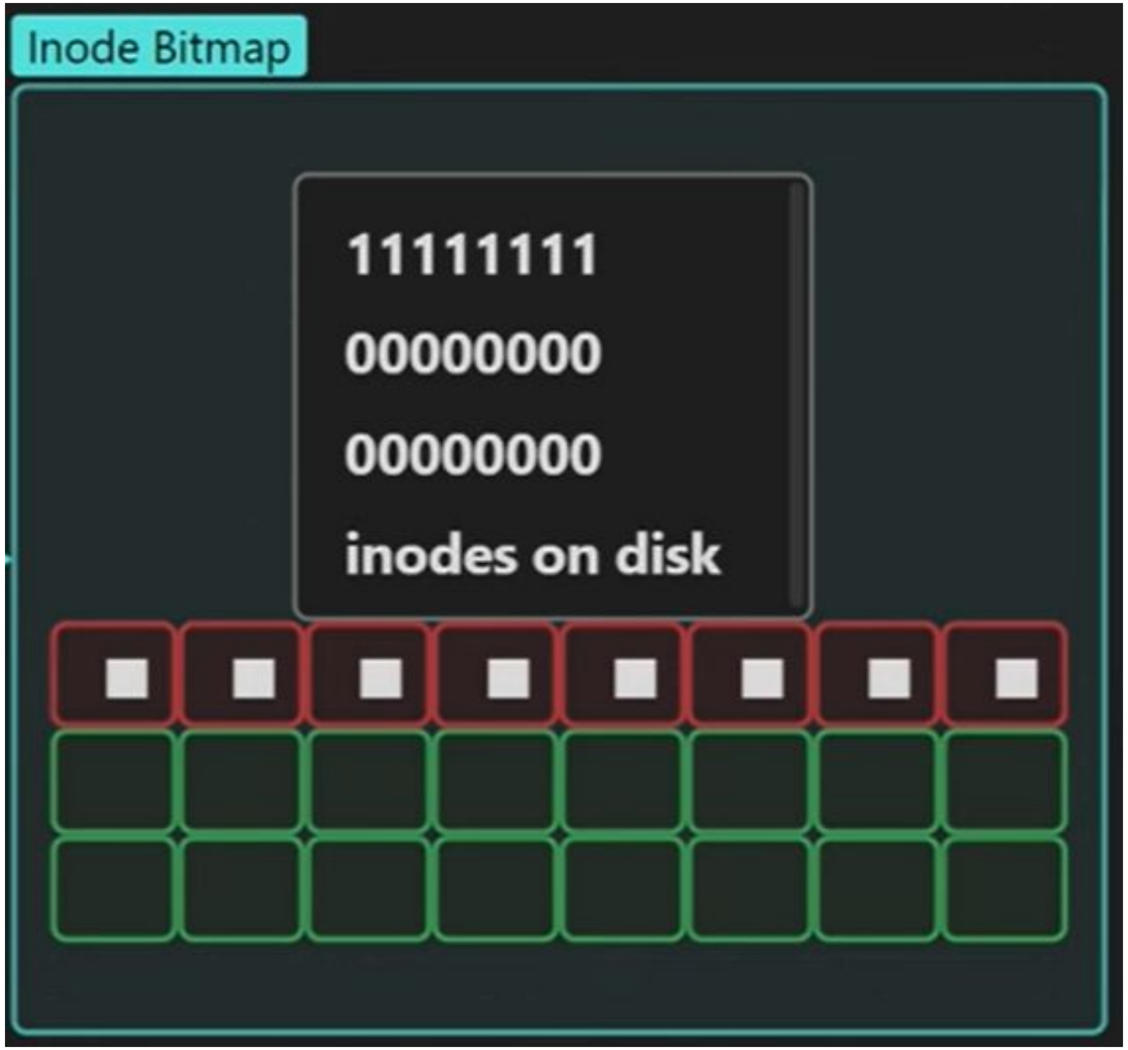
Dissection of the bitmaps of the file system (ext 2)



Block Bitmap



Inode Bitmap



Super Block	Block Group Descriptor	Block Bit Map	Inode Bit Map	Inode table	Data blocks

Block Group description table

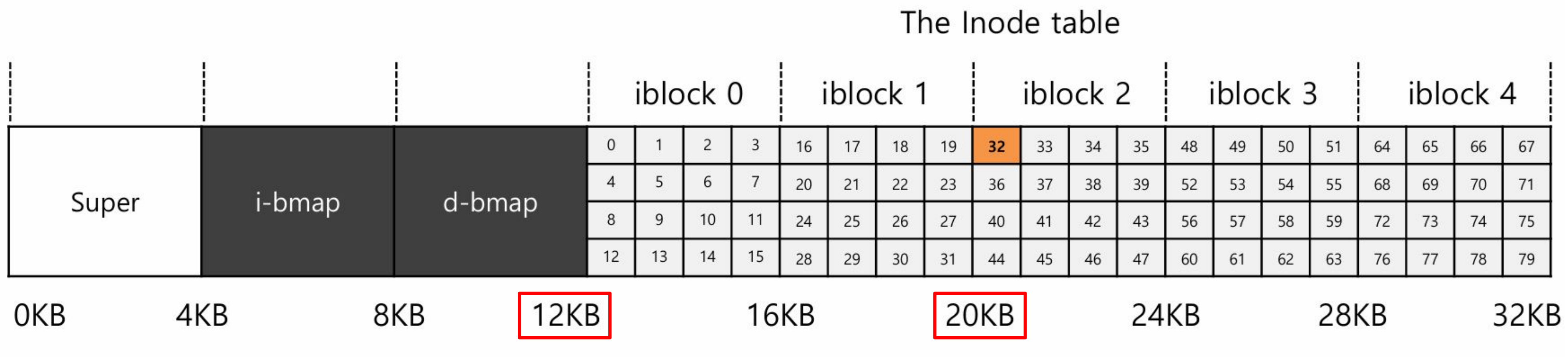
Block group descriptor table

This is 1 entry in the table		
bytes Offset	size bytes	Description
0	4	bg_block_bitmap
4	4	bg_inode_bitmap
8	4	bg_inode_table
12	2	bg_free_blocks_count
14	2	bg_free_inodes_count
16	2	bg_used_dirs_count
18	2	bg_pad
20	12	bg_reserved

File Organization: The inode

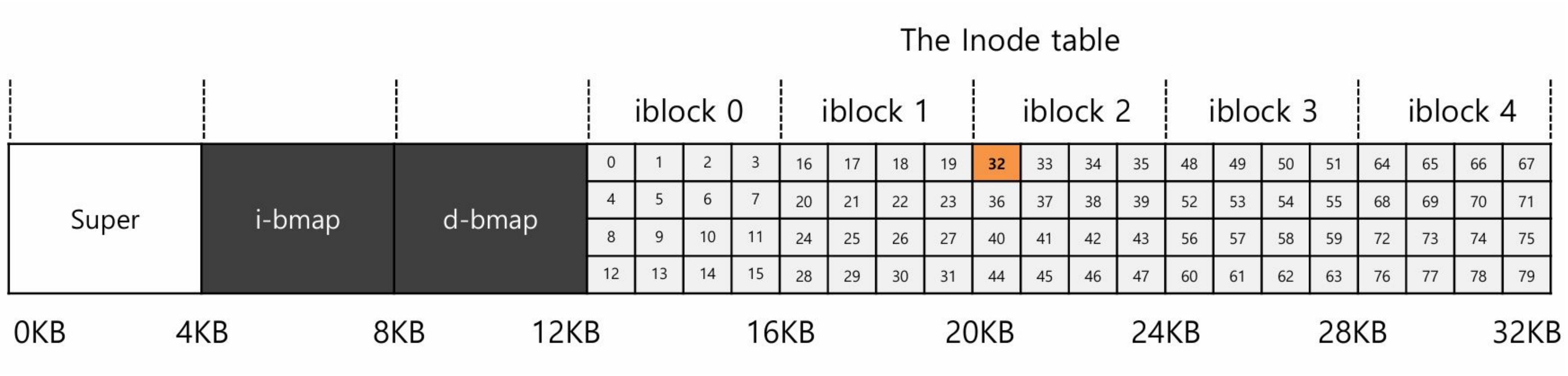
- Each inode is referred to by inode number.
  - by inode number, File system calculates where the inode is on the disk.
  - Ex) **inode number: 32** and **size of inode: 256 Bytes**
    - Theorem for calculating **the offset into the inode region = inode number * size of inode**
      - Therefore, offset in this scenario = **32 * 256 = 8192**
    - **Byte address or location of the inode number = start address of the inode table + offset**
      - Therefore, location of the inode number 32 = **12 KB + 8192 B = 12 KB + 8 KB = 20 KB**

    - To read inode number **32**, the file system would first calculate the offset into the inode region (**32** sizeof(inode) or **8192**), addit to the start address of the inode table on disk (inodeStartAddr = **12 KB**), and thus arrive upon the correct byte address of the desired block of inodes : **20 KB**.



File Organization: The inode

- Disks are not byte addressable, sector addressable.
- Disks consist of a large number of addressable sectors, (512 Bytes)
  - Ex) Fetch the block of inode (inode number: 32, size of inode: 256 Bytes, size of block: 4 KB)
    - Sector address is the physical address of the inode block:
    - **block # = (inode number * sizeof(inode)) / blocksize = (32 * 256 B) / 4 KB = 8192 B / 4 KB = 8 KB / 4 KB = 2 KB**
    - **sector address = (block # * block size) + inode table start address) / sector size = {(2 * 4 KB) + 12 KB} / 512 B = 40**



File Organization: The inode

- inodes have all of the information or meta data or attributes of a file.
  - File type (regular file, directory, etc.),
  - Size, the number of blocks allocated to it.
  - Protection information(who owns the file, who can access, etc).
  - Time information.
  - Etc.

File Organization: The inode Table

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?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	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
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

The EXT2 Inode

Super Block	Block Group Descriptor	Block Bit Map	Inode Bit Map	Inode table	Data blocks

Inode Table

First 11 entries of the FS are reserved

This is 1 entry in the table

bytes Offset	size bytes	Description
0	2	i_mode
2	2	i_uid
4	4	i_size
8	4	i_atime
12	4	i_ctime
16	4	i_mtime
20	4	i_dtime
24	2	i_gid
26	2	i_links_count
28	4	i_blocks
32	4	i_flags
36	4	i_osd1
40	15x4	i_block
100	4	i_generation
104	4	i_file_acl
108	4	i_dir_acl
112	4	i_faddr
116	12	i_osd2

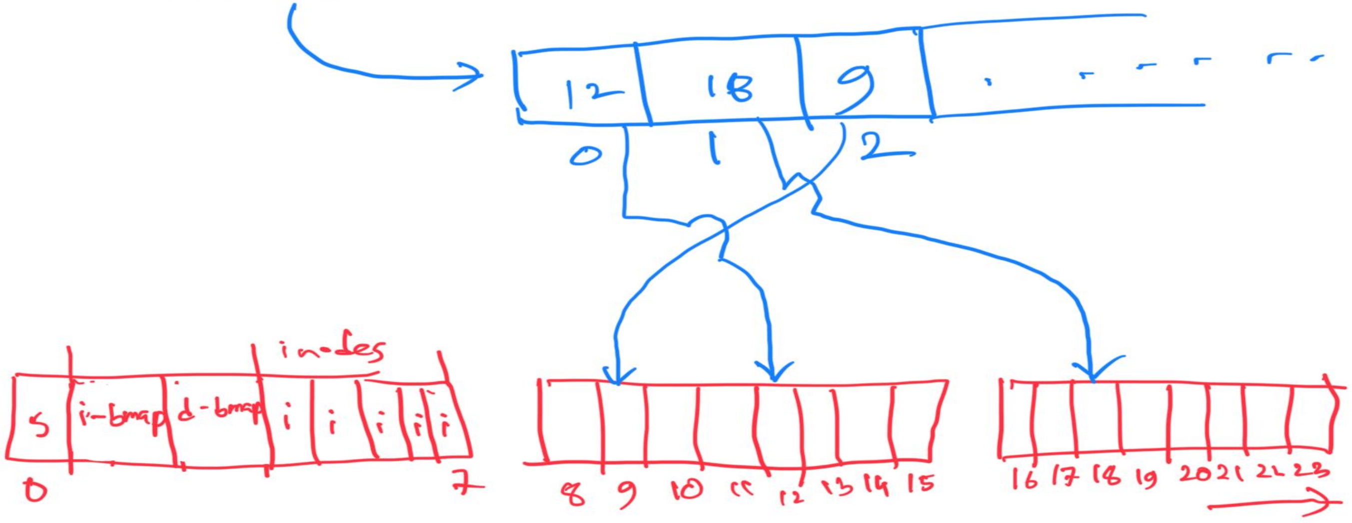
Dissection of the inode block field

An arbitrary example:

inode:

file attributes

block



Inode : Iblock field

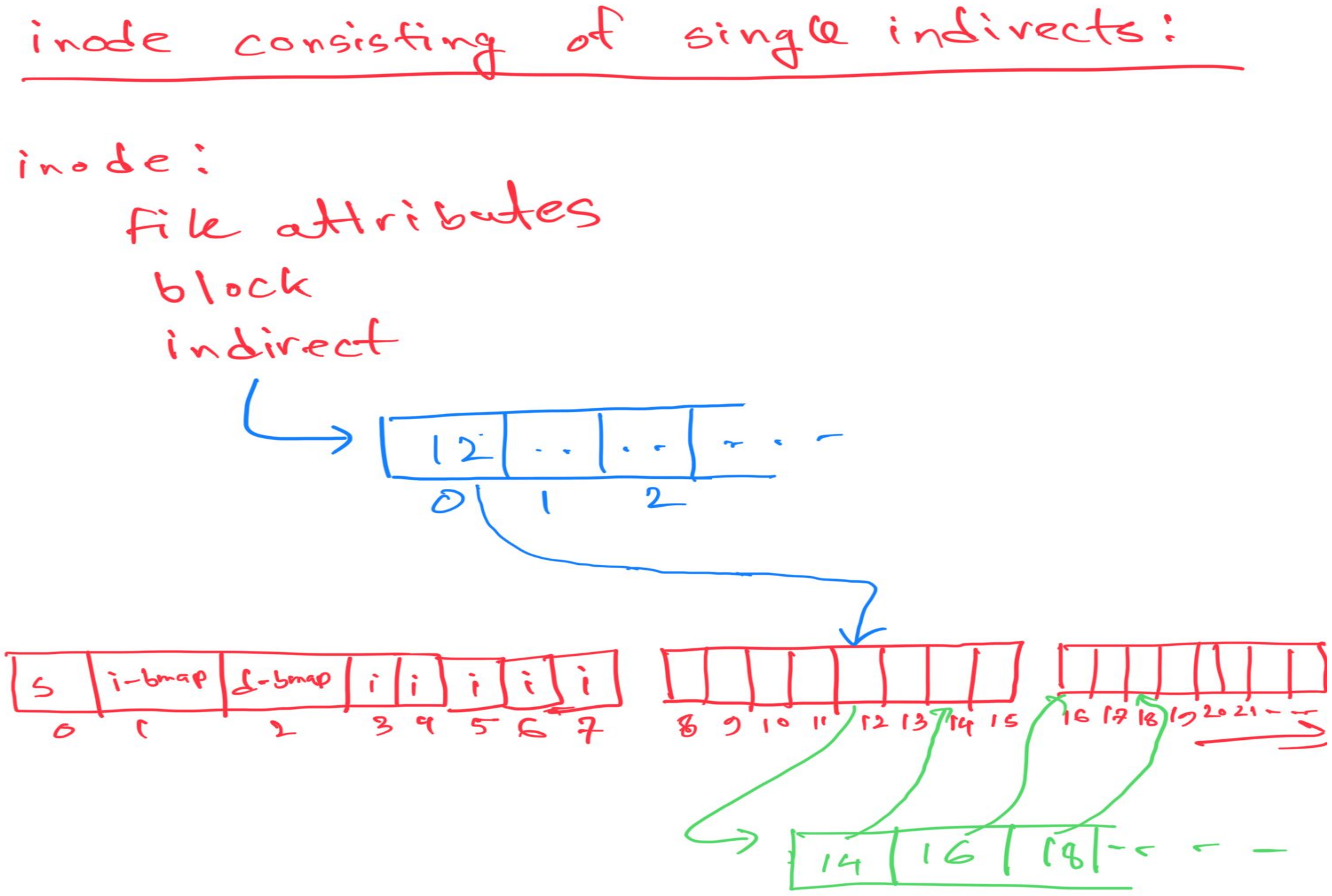
- 12 Direct Block Pointers
- 1 indirect Block Pointers
- 1 Doubly-indirect Block Pointers
- 1 Triply-indirect Block Pointers

- Total: 15 Block Pointers

i_block

bytes Offset	size bytes	Description
0	4	Direct Block Pointers
4	4	Direct Block Pointers
8	4	Direct Block Pointers
12	4	Direct Block Pointers
16	4	Direct Block Pointers
20	4	Direct Block Pointers
24	4	Direct Block Pointers
28	4	Direct Block Pointers
32	4	Direct Block Pointers
36	4	Direct Block Pointers
40	4	Direct Block Pointers
44	4	Direct Block Pointers
48	4	Indirect Block Pointers
52	4	Doubly-indirect Block Pointers
56	4	Triply-indirect Block Pointers

Dissection of the inode indirect

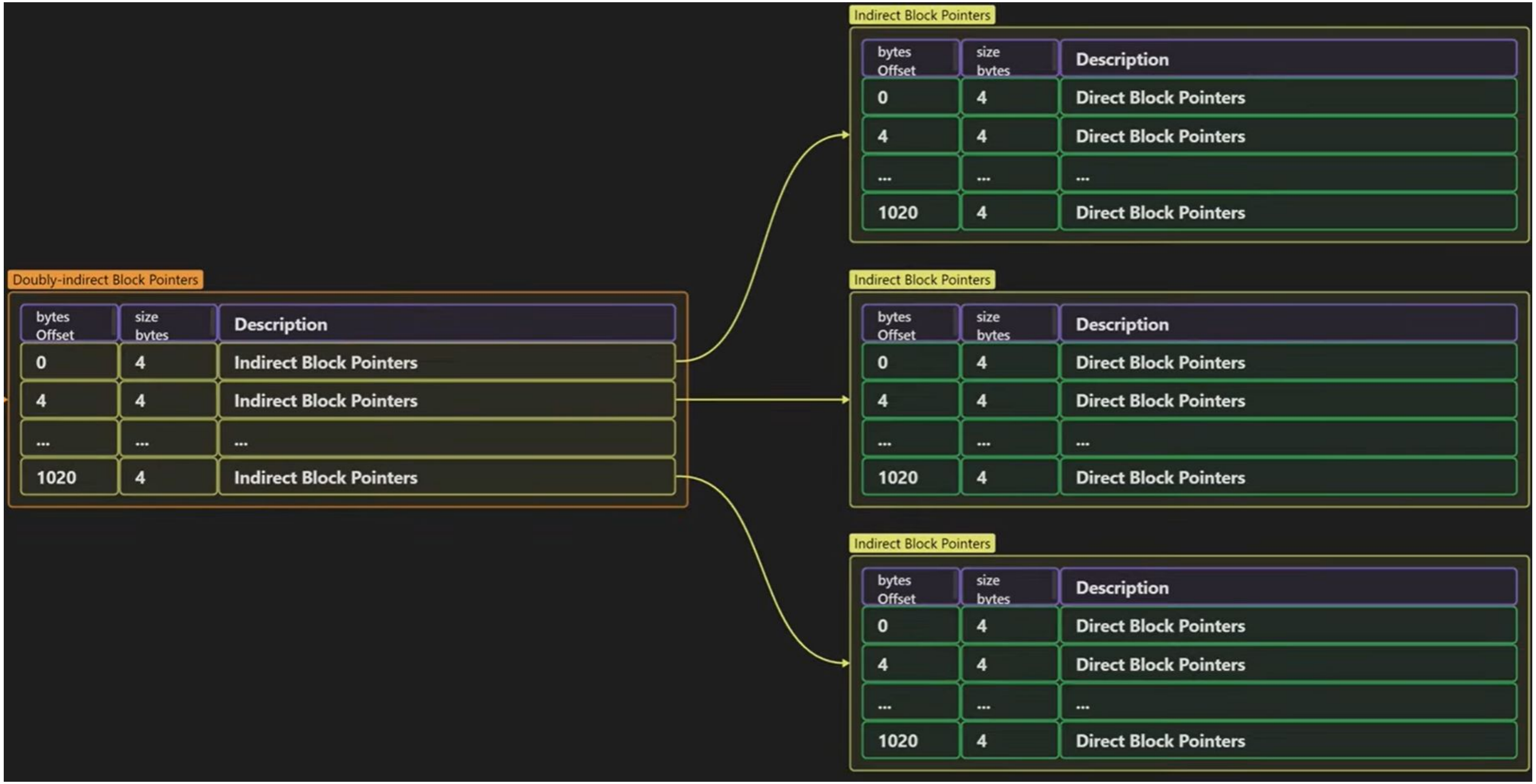


Inode Table: Indirect Block pointer (256)

Indirect Block Pointers

bytes Offset	size bytes	Description
0	4	Direct Block Pointers
4	4	Direct Block Pointers
...	...	...
1020	4	Direct Block Pointers

Inode Table: Doubly-Indirect Block pointer (256^2)



Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files .”dot” for current directory and ..”dot-dot” for parent directory
  - For example, dir has three files (foo, bar, foobar)

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

on-disk for dir

Sample Directory Table

bytes Offset	size bytes	Description
Directory Entry 0		
0	4	inode =783362
4	2	rec_len =12
6	2	name_len =1
8	1	name =.
9	3	Padding
Directory Entry 1		
12	4	inode =1109761
16	2	rec_len =12
18	2	name_len =2
20	2	name =..
22	2	Padding
Directory Entry 2		
24	4	inode =783364
28	2	rec_len =24
30	2	name_len =13
32	13	name =.bash_profile
45	3	Padding
Directory Entry 3		
48	4	inode =783363
52	2	rec_len =16
54	2	name_len =7
56	7	name =.bashrc
63	1	Padding
Directory Entry 4		
64	4	inode =783377
68	2	rec_len =12

Free Space Management

- File system track which inode and data block are free or not.
- In order to manage free space, we have two simple bitmaps.
  - When file is newly created, it allocated inode by searching the inode bitmap and update on-disk bitmap.
  - Pre-allocation policy is commonly used for allocate contiguous blocks.

Access Paths: Reading a File From Disk

- Issue an open("/foo/bar", O_RDONLY)
  - Traverse the pathname and thus locate the desired inode.
  - Begin at the root of the file system (/)
    - In most Unix file systems, the root inode number is 2
  - Filesystem reads in the block that contains inode number 2.
  - Look inside of it to find pointer to data blocks (contents of the root).
  - By reading in one or more directory data blocks, It will find "foo" directory.
  - Traverse recursively the path name until the desired inode ("bar")
  - Check finale permissions, allocate a file descriptor for this process and returns file descriptor to user.

Access Paths: Reading a File From Disk

- Issue read() to read from the file.
    - Read in the first block of the file, consulting the inode to find the location of such a block.
        - Update the inode with a new last accessed time.
        - Update in-memory open file table for file descriptor, the file offset.
- When file is closed:
    - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No disk I/Os take place

Access Paths: Reading a File From Disk

Problem:

- An existing file named “bar” needs to be read which is allocated in 3 data blocks.
- Path of the file: “/foo/bar”
- To read the file it was opened first by open() system call.
- After opening the file read() system call was issued in the file to read the contents.

Illustrate the file access path timeline according to the scenario described above.

Access Paths: Reading a File From Disk

	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 1							
						read 2				
				read 3						
							read 4			
					5 read					
read()					6 read					
								7 read		
					8 write					
read()					9 read					
									10 read	
					11 write					
read()					12 read					
										13 read
					14 write					

File Read Timeline (Time Increasing Downward)

Access Paths: Writing to Disk

- Issue write() to update the file with new contents.
- File may allocate a block (unless the block is being overwritten).
  - Need to update data block, data bitmap.
  - It generates five I/Os:
    - one to read the data bitmap
    - one to write the bitmap (to reflect its new state to disk)
    - two more to read and then write the inode
    - one to write the actual block itself.
  - To create file, it also allocate space for directory, causing high I/O traffic.

Access Paths: Writing to Disk

Problem:

- A file named “bar” has been created by create() system call.
- Path of the newly created file: “/foo/bar”
- After creating the file write() system call was issued in the file to write new contents and after the write operation the file has been allocated in 3 data blocks.

Illustrate the file access path timeline according to the scenario described above.

Access Paths: Writing to Disk

	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)		5 read write	read 1	3 read 8 write	7 read write	read 2	4 read 6 write			
write()	read write 10				9 read 12 write			11 write		
write()	read write 14				13 read 16 write				15 write	
write()	read write 18				17 read 20 write					19 write

File Creation Timeline (Time Increasing Downward)

Crash Consistency and Journaling

Overview

- File system data structures must **persist.**
  - files, directories, all of the other metadata ,etc

- How to update persistent data structure?
  - If the system crashes or loses power, on-disk structure will be in **inconsistent** state.

- In this chapter, we describe how to update file system consistently

Chapter 42 from this Book:
https://www.dropbox.com/scl/fi/v44z7gtbacrqlxo71brra/Remzi-H.-Arpaci-Dusseau-Andrea-C.-Arpaci-Dusseau-Operating-Systems-Three-Easy-Pieces-2023.pdf?rlkey=77bvmf6k57mrr701tryc8tj0u&dl=0

What is Crash Consistency?

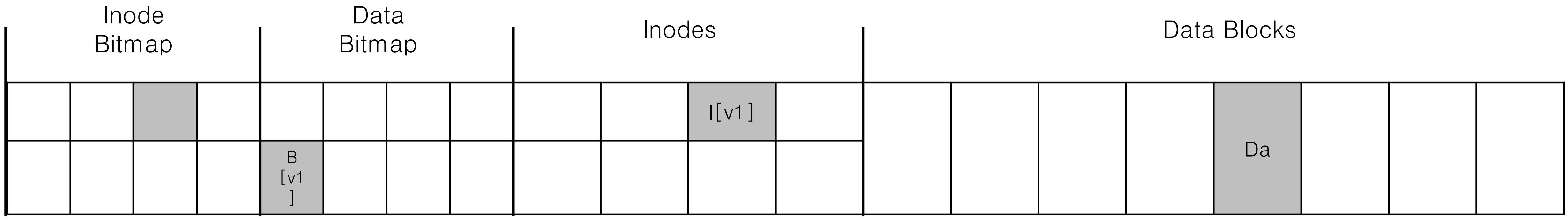
- Crash consistency refers to the property of a file system ensuring that data remains in a correct and usable state even if the system crashes (e.g., due to power loss, kernel panic, or hardware failure).

- When a system crashes during a write operation, data structures (like metadata and file contents) may be left in an inconsistent state.

- Without crash consistency mechanisms, file systems could become corrupted, leading to data loss or system instability.

An Example of Crash Consistency

- Scenario
  - Append of a single data block to an existing file.
  - open() ⇒ lseek() ⇒ write() ⇒ close()

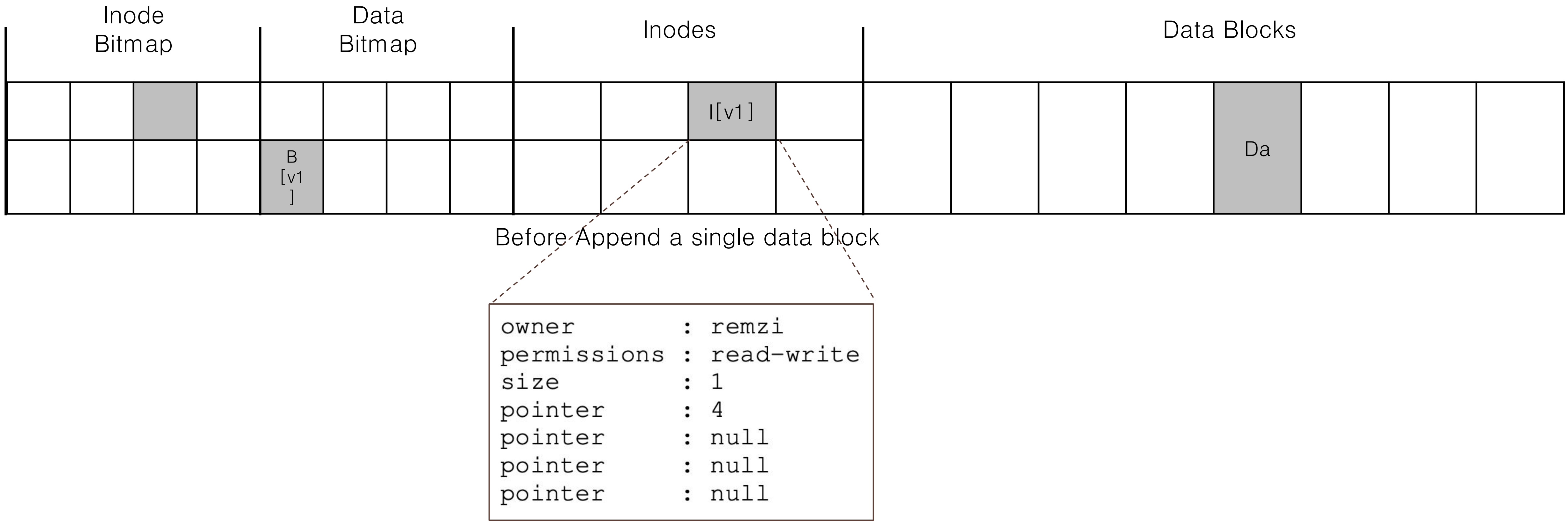


Before Append a single data block

- Inode Bitmap represents the number of the Inode block (0, 1, …, 7 in this example) which is filled, 1 (shaded blox) indicates full, 0 empty (blank box)
- Data Bitmap represents the number of the Data block (0, 1, …, 7 in this example) which is filled, 1 (shaded blox) indicates full, 0 empty (blank box)
- B[v1] represents the first version of the data block, just one single data is added
- I[v1] represents the the first version of the Inode where the Inode is data structure representing a file
- Da represents a data

An Example of Crash Consistency

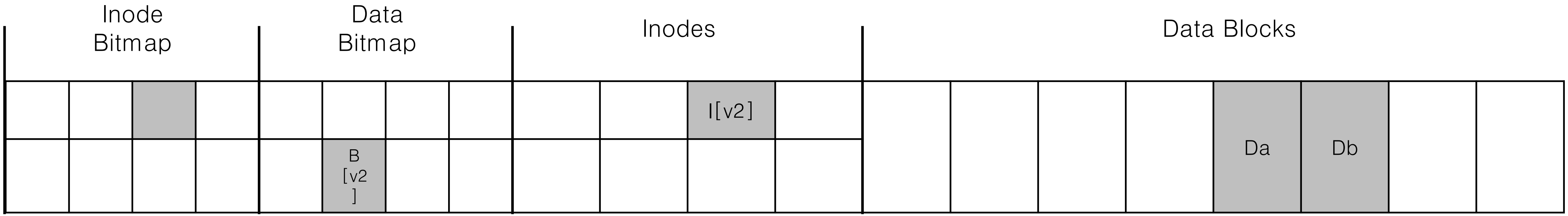
- Scenario
    - Append of a single data block to an existing file.



Before Append a single data block

An Example of Crash Consistency (Cont.)

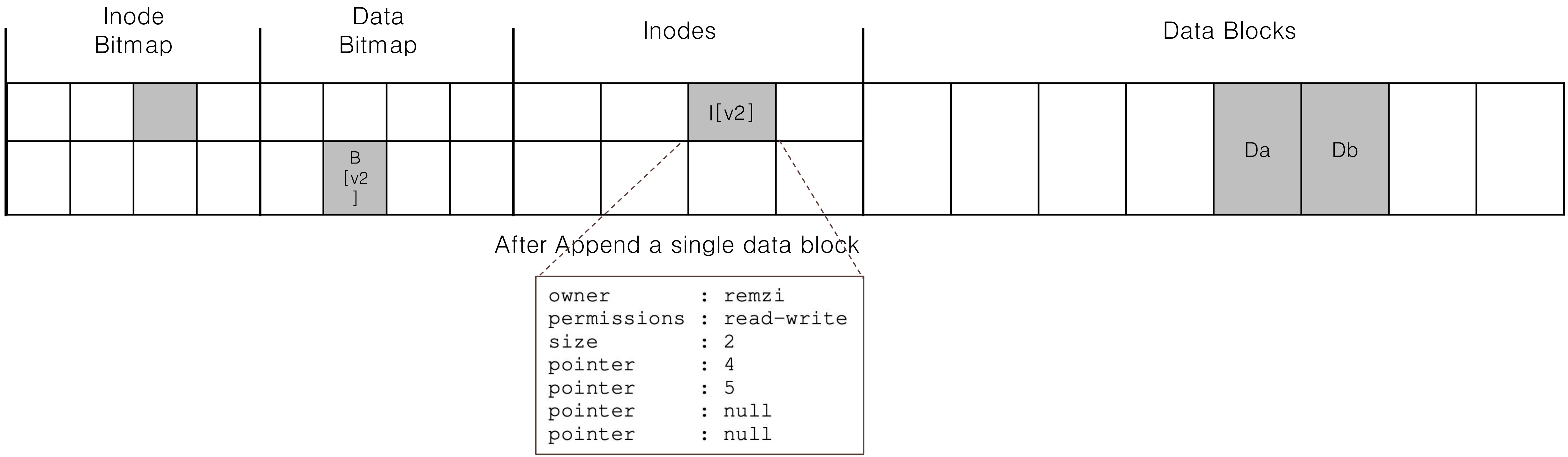
- File system perform three writes to the disk.
  - Data bitmap is updated (B[v2])
  - Inode is updated (I[v2])
  - New Data block (Db) is added



After Append a single data block

An Example of Crash Consistency (Cont.)

- File system perform three writes to the disk.
  - inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)



After Append a single data block

Crash Scenario

- Only one of the below block is written to disk.
  - Data block (Db): lost update
  - Update inode (I[v2]) block: garbage, consistency problem
  - Updated bitmap (B[v2]): space leak

- Two writes succeed and the last one fails.
  - The inode(I[v2]) and bitmap (B[v2]), but not data (Db).: consistent from the system's metadata
  - The inode(I[v2]) and data block (Db), but not bitmap(B[v2): inconsistent
  - The bitmap(B[v2]) and data block (Db), but not the inode(I[v2]): inconsistent

Crash-consistency problem (consistent-update problem)

Solution

- The File System Checker (**`fsck`**)
  - **`fsck`** is a Unix tool for finding inconsistencies and repairing them.
  - **super block[*]:** if the number of blocks in the filesystem is larger than the filesystem size. If the block is corrupted, the corrupted block is replaced by an alternate copy of the superblock.
  - **free blocks:** scans the inodes, indirect blocks, double indirect blocks. Goal is to make sure the file system metadata is internally consistent.
  - **inode state:** check if the state of each inode is valid (Ex: valid type file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed. The inode is considered suspect and cleared by **`fsck`.**
  - **inode link:** check if the reference count for each inode is consistent
  - check if a block is shared by the two inodes.
  - **check for "bad" block pointer:** "bad" block pointer is the one that points to the location that lies outside the filesystem partition.
  - **directory:** Check if . and .. are properly set up. Make sure that there are only one hard link for a directory.

[*]Super block contains this information for particular file system w Ex) The number of inodes, begin location of inode table. etc

Drawbacks of FSCK

- Building a working **`fsck`** requires complex knowledge of the file system.
- **`fsck`** have a bigger and fundamental problem: too slow
  - scanning the entire disk may take many minutes or hours.
  - Performance of fsck became prohibitive.
    - As disk grew in capacity.

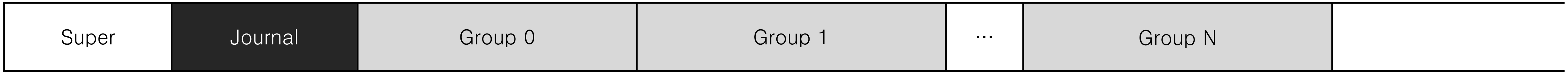
Solution

- **Journaling** (or Write-Ahead Logging)
  - Before overwriting the on-disk structures in place, write down a little note on the disk, describing what you are to do.
  - Writing this note is the “write ahead”. The structure that is the destination of the “write ahead” is called log. hence, This is Write-Ahead Logging.

Journaling

- File system reserves some small amount of space within the partition or on another device.

without journaling

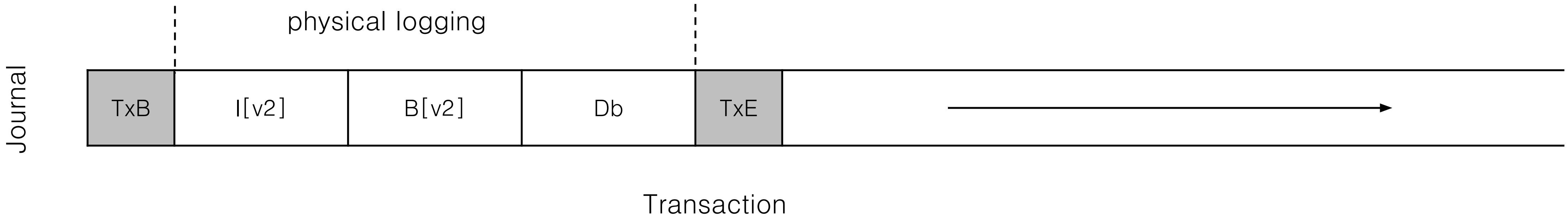


with journaling

Data Journaling

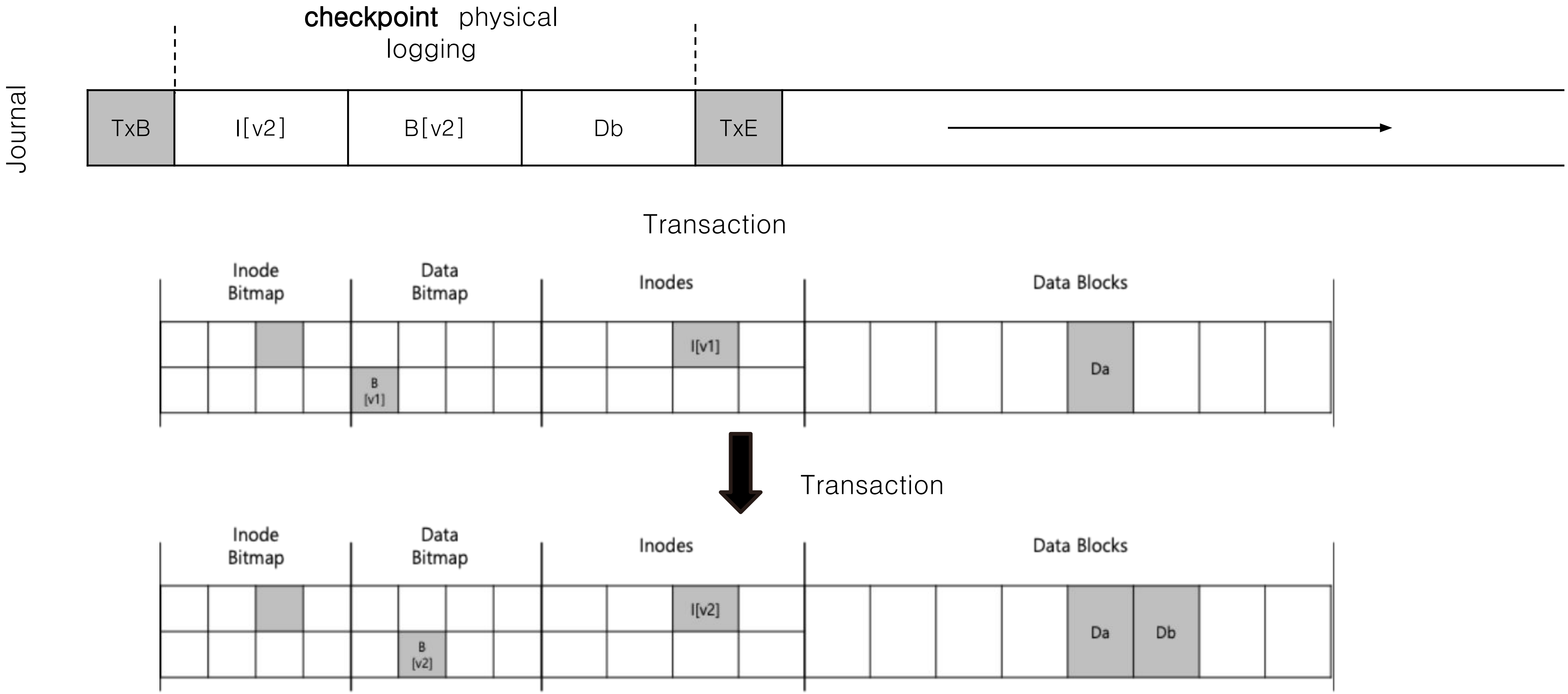
- Lets' update a file (appending a data block to a file). Following structures are updated.
  - inode (I[v2]), bitmap (B[v2]), and data block (Db)

- First, **Journal write**: write the transaction as below.
  - TxB: Transaction begin block (including transaction identifier)
  - TxE: Transaction end block
  - others: contain the exact contents of the blocks



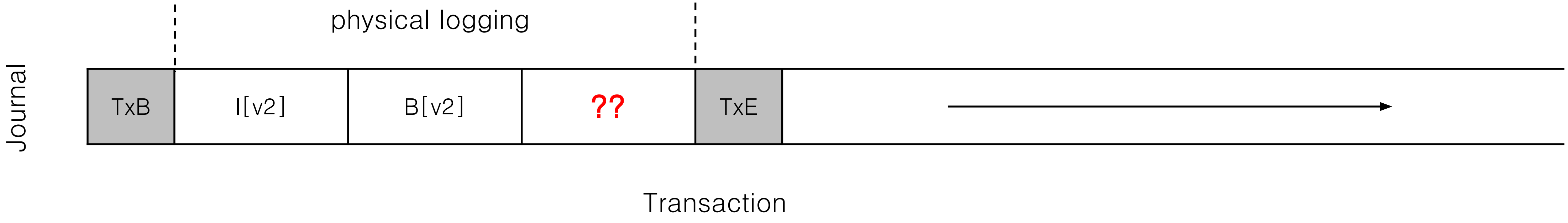
Data Journaling (Cont.)

- Second, **Checkpoint**: Write the physical log to their original disk locations.



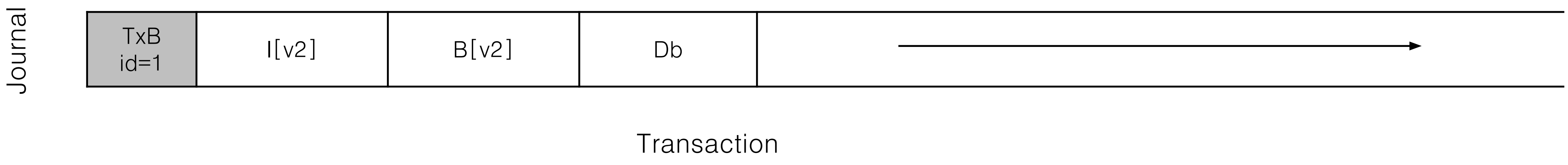
Crash during Data Journaling

- Wat if a *crash occurs* during the writes to the journal?

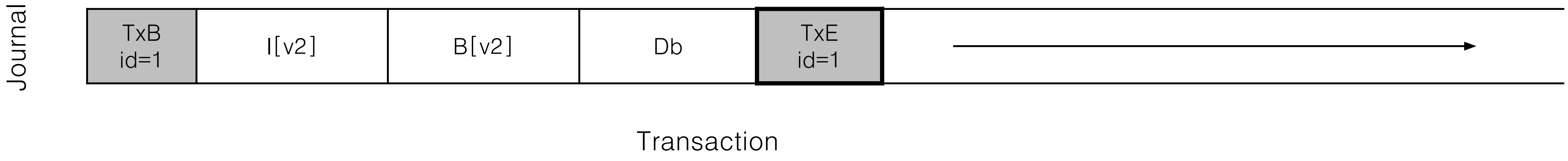


To avoid data being inconsistent

- First, write all blokes **except the TxE block** to journal.



- Second, The file system issues the write of the TxE.



To avoid data being inconsistent (Cont.)

- **Journal write**: write the contents of the transaction to the log
- **Journal commit**: write the transaction commit block
- **Checkpoint**: write the contents of the update to their locations.

Recovery

- If the crash happens, before the transaction is written to the log
  - The pending update is skipped.

- If the crash happens, after the transactions is written to the log, but before the checkpoint.
  - **Recover** the update as follow:
    - Scan the log and look for transactions that have committed to the disk.
    - Transactions are replayed.

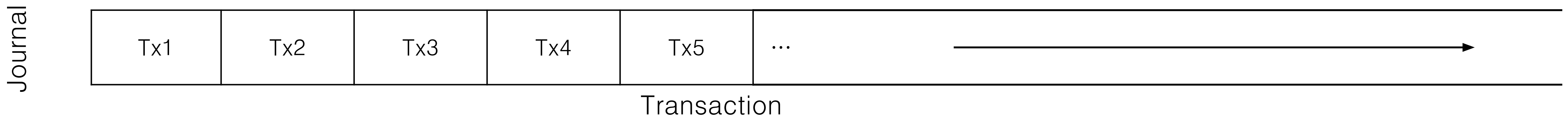
Batching Log Updates

- If we create two files in same directory, the same inode and the directory entry block is to the log and committed twice.

- To reduce excessive write traffic to disk, journaling manage the **global transaction**.
    - Write the content of the global transaction forced by synchronous request.
    - Write the content of the global transaction after timeout of 5 seconds.

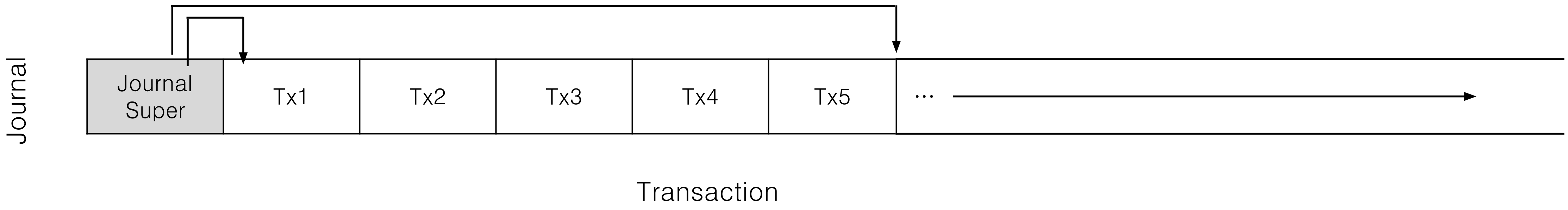
Making the log finite

- The log is of a finite size (**circular log**).
  - To re-using it over and over



Making The log Finite (Cont.)

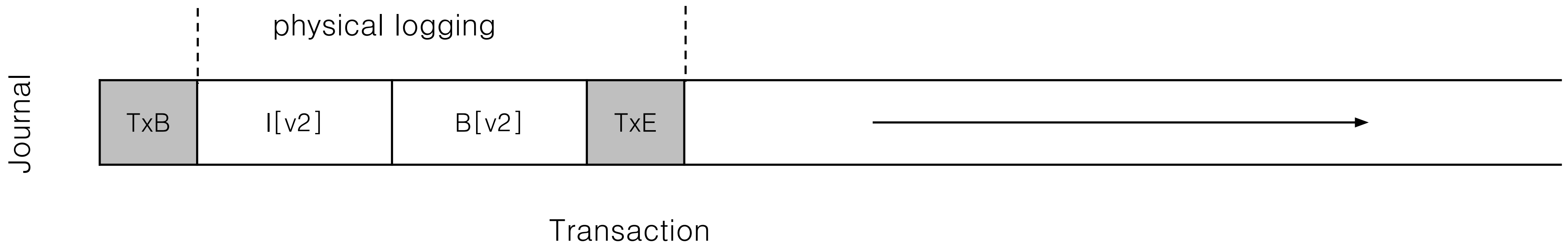
- journal super block
  - Mark the oldest and newest transactions in the log.
  - The journaling system records which transactions have not been check pointed.



Metadata Journaling

- Because of the high cost of writing every data block to disk twice
  - commit to log (journal)
  - checkpoint to on-disk location.

- Filesystem uses ordered journaling (metadata journaling).

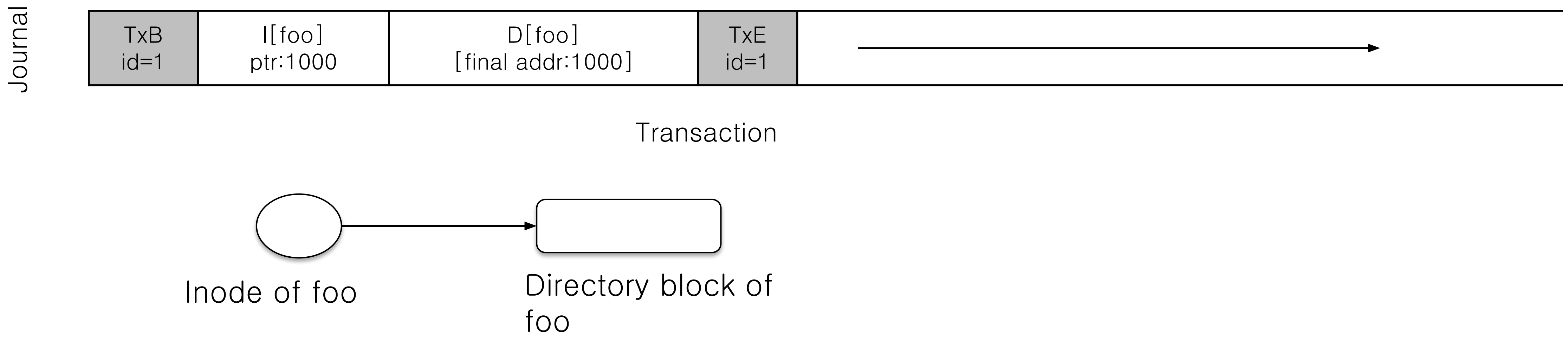


Metadata Journaling (Cont.)

- **Data Write**: Write data to final location
- **Journal metadata write**: Write the begin and metadata to the log
- **Journal commit**: Write the transaction commit block to the log
- **Checkpoint metadata**: Write the contents of the metadata to the disk
- **Free**: Later, mark the transaction free in journal super block

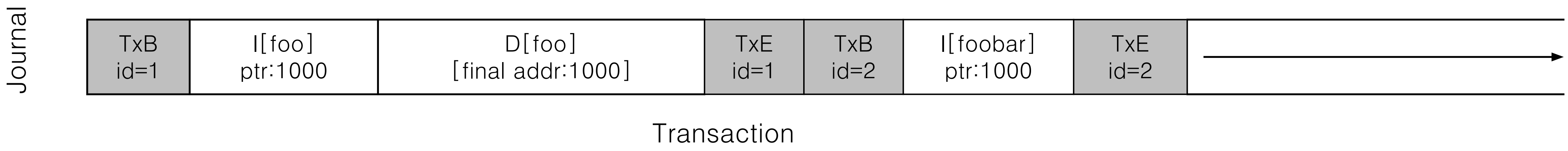
Tricky case: Block Reuse

- Revoke record: some metadata should not be replayed.
- Scenario.
  1. Directory “foo” is updated.

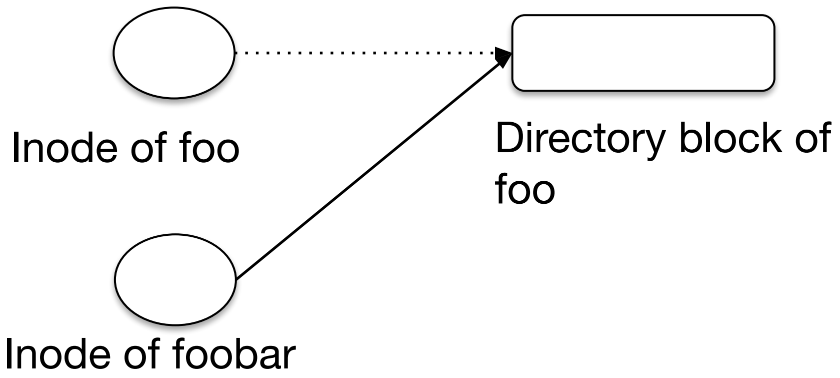


Tricky case: Block Reuse

- Scenario.
  1. Delete “foo” directory, freeing up block 1000 for reuse
  2. Create a file “foobar”, reusing block 1000 for data

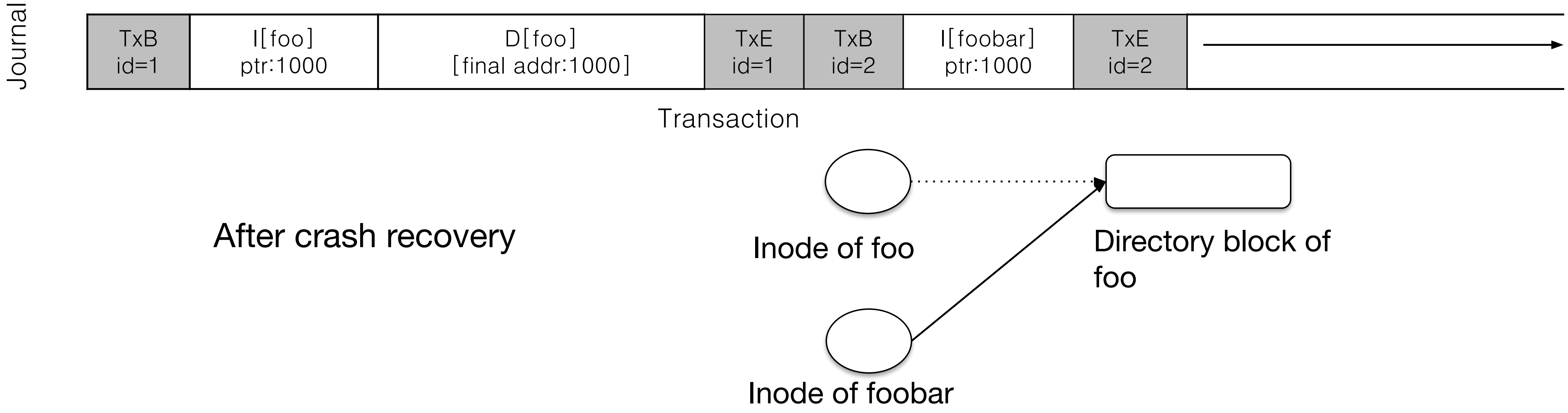


After crash recovery



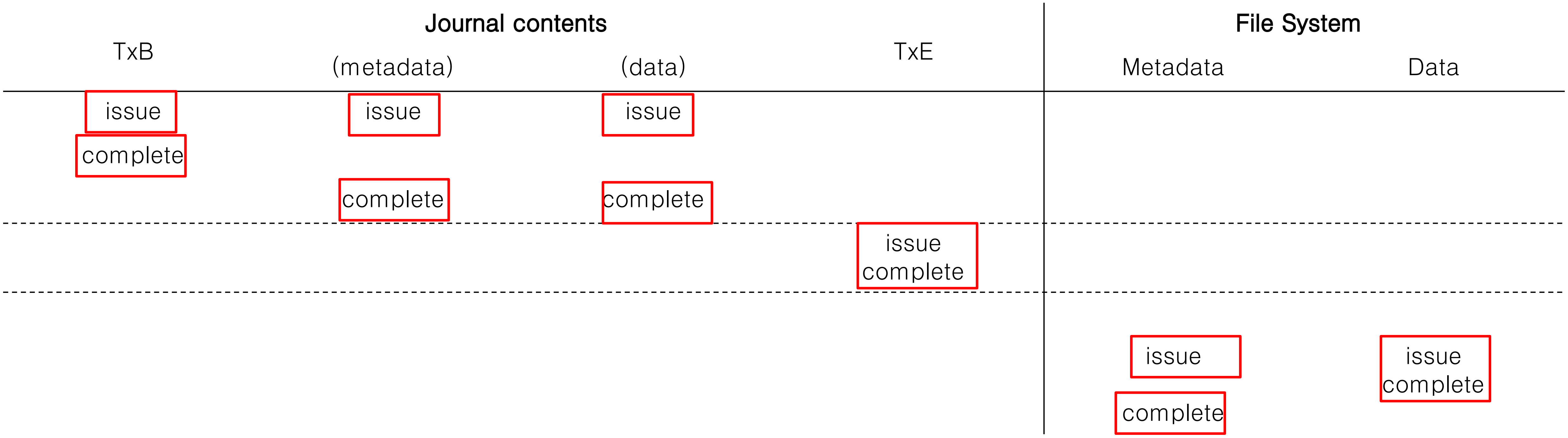
Tricky case: Block Reuse

- Problem: allocate the block that was deleted but was not checkpointed.



- Solution1: Do not use the deleted block until it is checkpointed.
- Solution 2: When deleting a directory, record “revoke” at the journal.

Data Journaling Timeline



Data Journaling Timeline

Metadata Journaling Timeline

Journal contents			File System	
TxB	(metadata)	TxE	Metadata	Data
issue	issue			issue
complete				
	complete			complete
		issue complete		
			issue	
			complete	

Metadata Journaling Timeline

End of Lecture