Video 1. File System

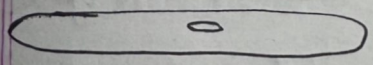
File → read/write. { OS এ file system এটি Deal করে }.

HDD structure / Magnetic Disk

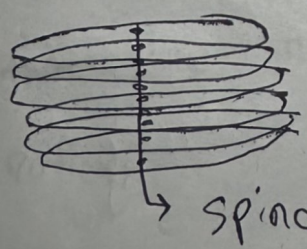
RAM is just a piece of continuous memory.
RAM এর ভিতরে কোন file থাকে না।

Secondary memory (HDD) এর মধ্যে থাকে।

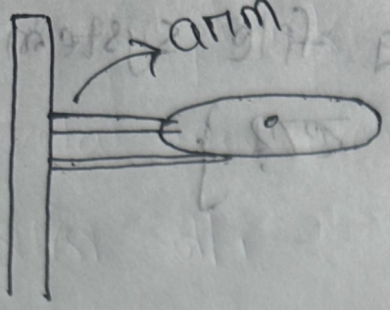
File is just an imaginary boundary bounded by 0/1 and which stores information

 in HDD is called a platter.

A stack of platter altogether is called a HDD.



Platter এর top & bottom দুই side এই independently information store করা যায়।

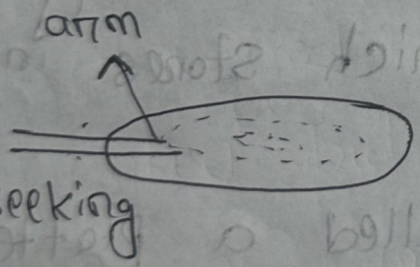


Arm assembly.

Platter route হচ্ছে but arm same place থাকে & data read হচ্ছে।

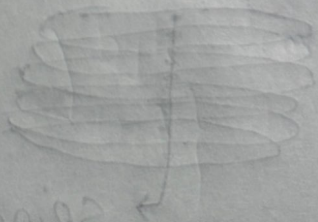
So, a platter should have two arms. And, will work independently.

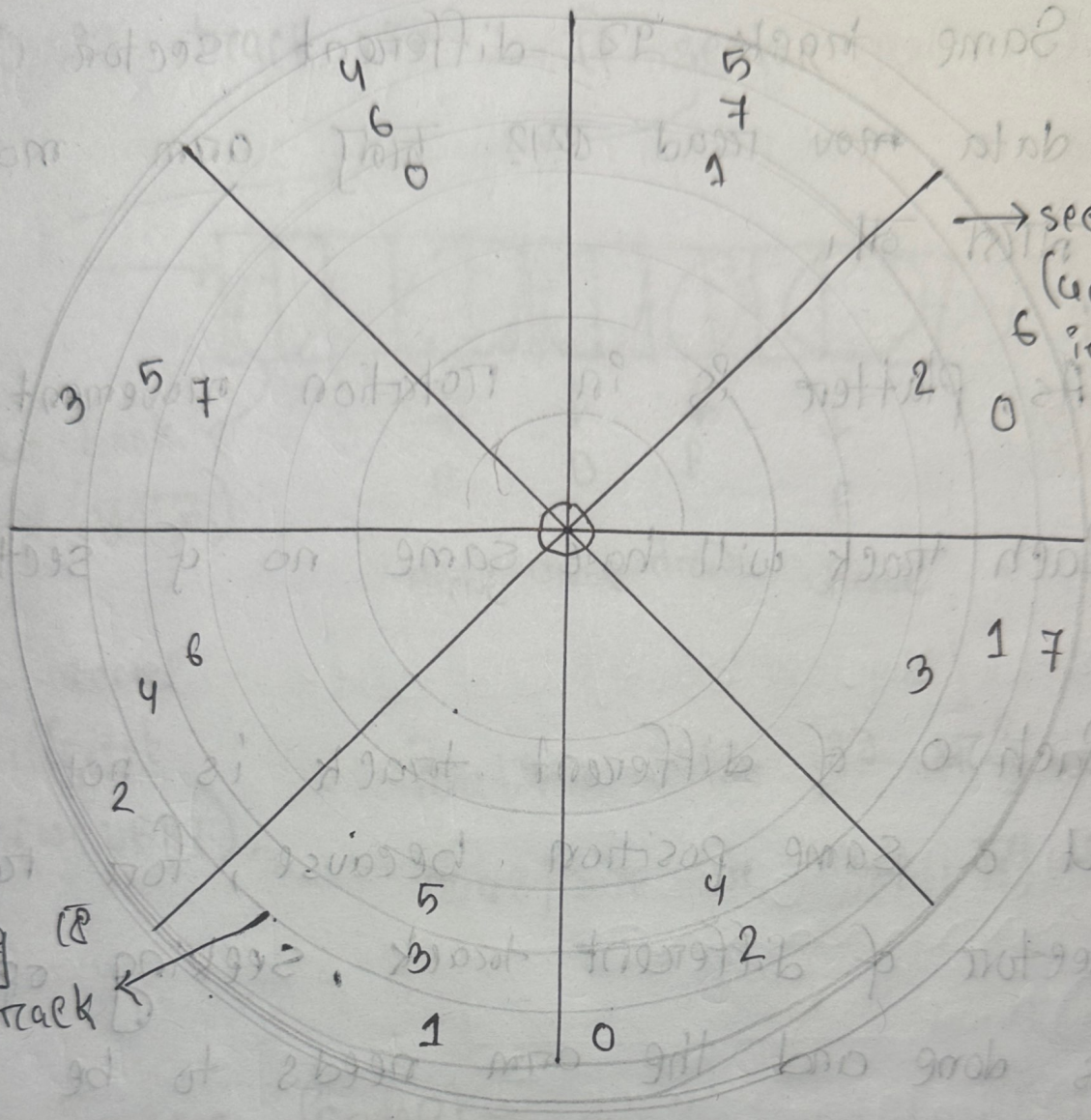
If we have n platters then we will have 2n arms.



arm সামনে বা পিছে যাওয়ার mechanism কে seeking বলি।

Arm assembly same place এ থাকে। Arm just extend হচ্ছে।





each sector will have equal amount of data always.

So, I have 6 tracks.

When I want to move from one track to another, we need to move the arm. Seek operation.

Same track এর different sector এর data ~~not~~ read করার জন্য arm move করা লাগে না।

As, platter is in rotation movement.

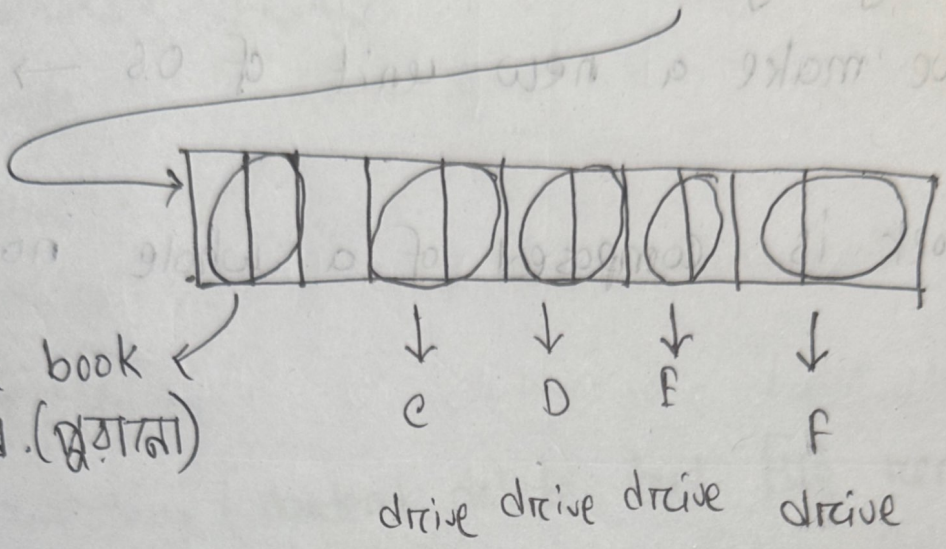
Each track will have same no of sectors.

Each 0 of different track is not placed at a same position because, for reading sector of different track, seeking operation is done and the arm needs to be extended/ shrinked but the platter is rotating. So, to align the arm and the sector, we don't place the zero at a same position.

* RAM এর unit 1 byte.
* HDD এর unit 1 sector.

} 1 sector হলো 1 byte এর (৫১২) ২টা.

HDD র track র 1 টি sector



master book record. (পুরানো)

↓

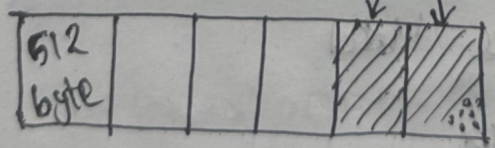
keeps record of different os. new (UEFI).

Partition table র মধ্যে, drive name / size etc store করা হয়।

File System Concept

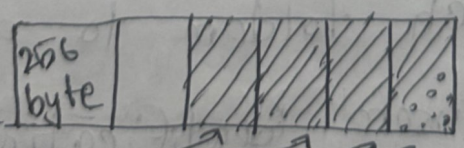
hi.txt = 900 bytes

Samsung { 1 block = 1 sector



So, now hi.txt file needs 2 block to store in both HDD.

We cannot change the size



Toshiba { 1 block = 2 sector

hi.txt → 900 bytes

Operating system এ sector নিয়ে কাজ করি না। we make a new unit of OS → a block.

A block is composed of a whole no of sectors (2^n).

sector / physical block / data block = Same.

File Attributes

i) Name : Yo

ii) Identifier : unique id

iii) Type or Extension : .pdf , .mp3

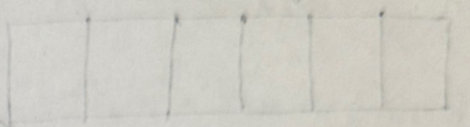
iv) Location : where HDD

v) size : 512 byte

vi) Protection : which folder to access & not.

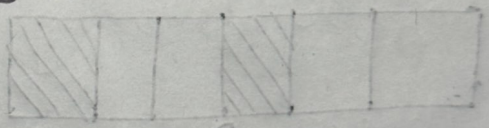
vii) Time , date & user identification : 1 pm , 4/08/25

File Operations



i) Create
ii) Read
iii) Write
iv) Delete
v) Truncating (content delete but File rename)
vi) Repositioning : changes the position of cursor.

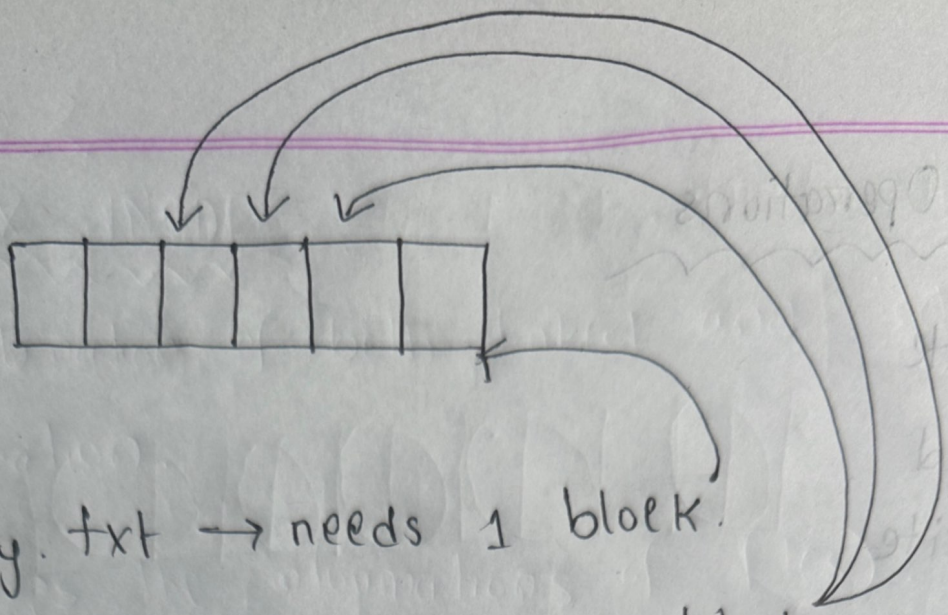
File Allocation Methods



i) linked allocation
ii) indexed allocation
iii) UNIX inode.

allocation methods are of 2 types:

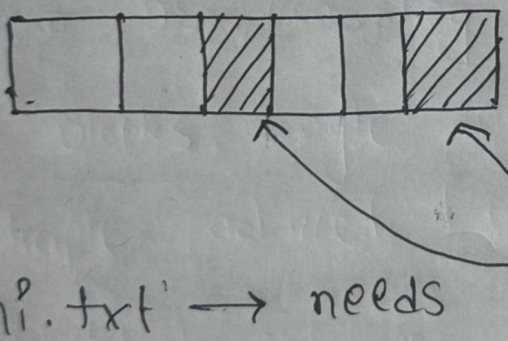
i) Contagious location
ii) non-contagious location.



hey.txt → needs 1 block.

hello.txt → needs 3 block.

This is called, contiguous (continuous) allocation.



hi.txt → needs 2 block

This is called non-contagious allocation.

We will study this.

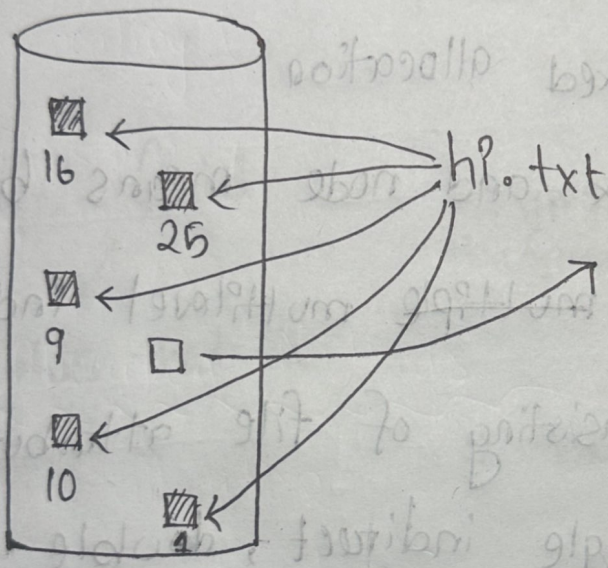
↳ contains a list of numbers to know where are my files are located.

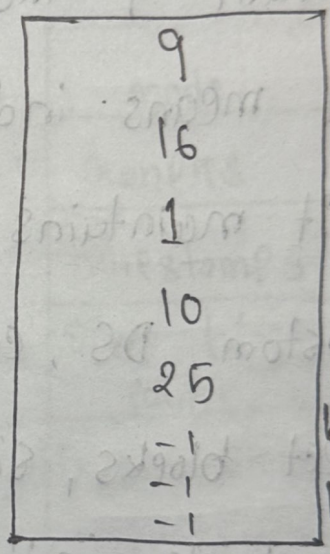
↳ it can be in any order.

↳ -1 caused the list is unused.

↳ my list is fixed.

Non-contagious allocation (indexed allocation)





This is called index block. Here, highest 8 blocks can be hold. we have given -1. So, that we can understand that it is empty.

Benefits

→ supports random or direct access.

→ No external fragmentation.

Drawbacks

→ pointers overhead

→ multilevel index when file is too large.

Video: 2

UNIX INODE

Combination of both linked and indexed allocation.

it keeps the informations of files).

i) Variant of indexed allocation

ii) I means index and node means block.

iii) it maintains ~~multiple~~ multilevel index.

iv) Custom DS, consisting of file attributes, direct blocks, single indirect, double indirect and triple indirect fields.

v) Direct blocks: Stores such data blocks where each block stores pointer to a block where data is stored.

vi) Single indirect: Stores such data blocks where each blocks store pointer to a direct block.

vii) Double indirect: Stores such data blocks where each block stores pointers to a single indirect.

viii) Triple indirect: Stores such data blocks where each block stores pointers to a double indirect.

field ↙

metadatas that represent a file.

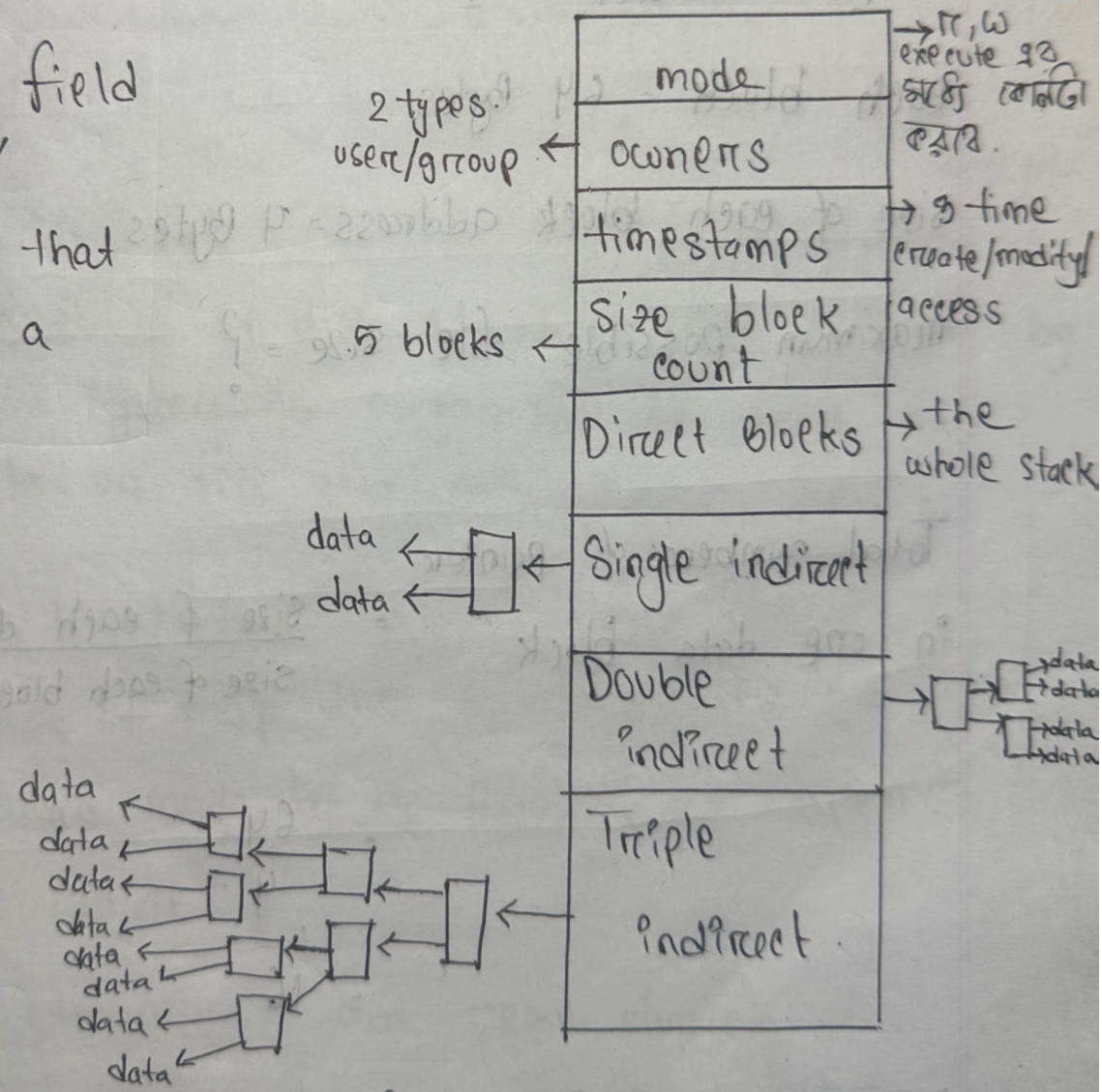


Fig: The UNIX INODE.

Math

UNIX INODE Structure.

direct blocks = 4

single indirect = 1

double indirect = 1

triple indirect = 1

each block = 64 Bytes

size of each block address = 4 Bytes

maximum possible file size = ?

Total number of pointer in one data block $= \frac{\text{size of each data block.}}{\text{size of each block address/pointer}}$

$= \frac{64}{4}$

$= 16$

Total number of pointers $= (4 + 16 + 16^2 + 16^3) = 4372$

maximum file size = no of total pointers x block size

= 4372 x 64

= 279808 Byte

= 273.25 kB.

File System Implementation

2 aspect

→ Data Structures

what type 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()

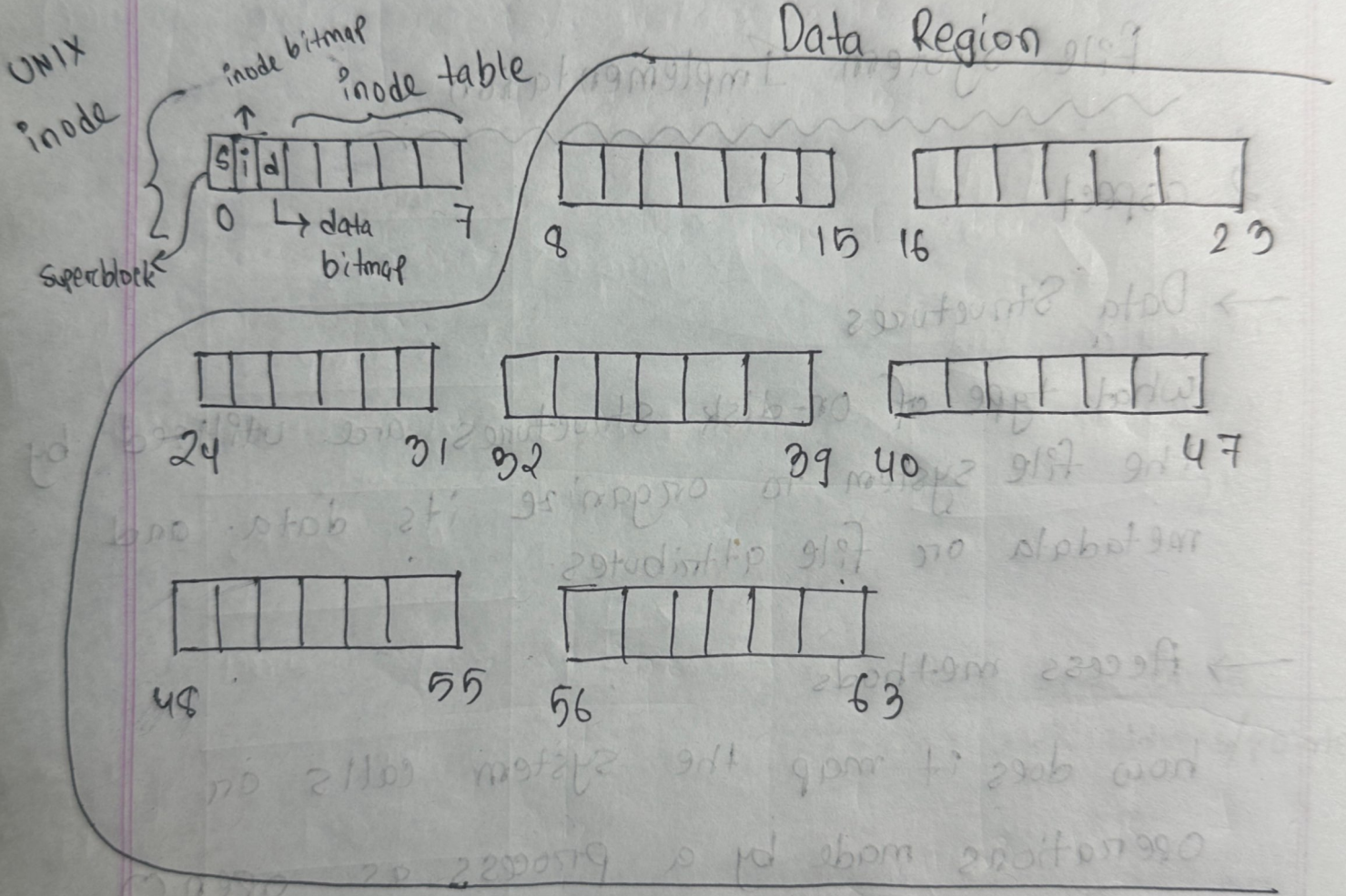
which structures are read during the execution of a particular system call

Overall Organization of VSFS (Very Simple file system)

Divide the disk into data blocks.

→ each block size is 4 kB

→ the blocks are addressed from 0 to N-1.



File organization : The INODE

(*) inode number = 32

Size of inode = 256 bytes

offset = 32 × 256 = 8192

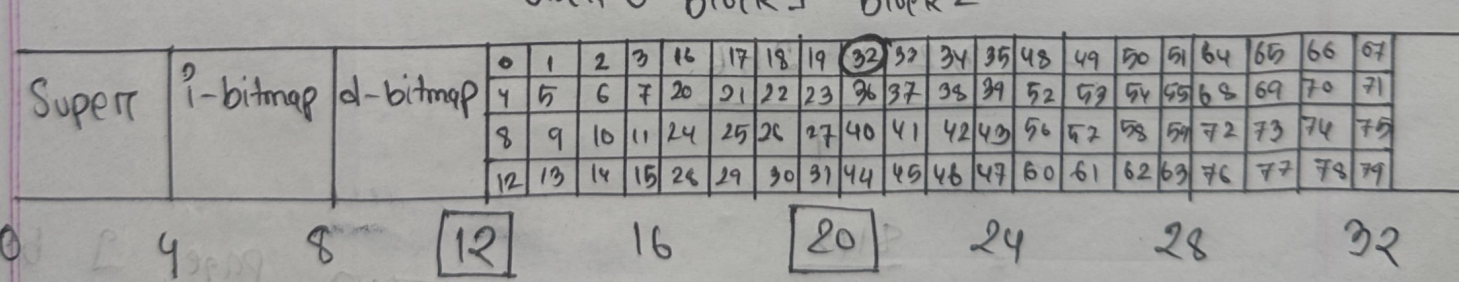
↳ মানে আমরা jump দিয়ে কোথায় যাবো।

byte address or location of inode no = start address of the inode table + offset

location of 32 no inode = 12kB + 8192 B

= 12kB + 8 kB

= 20 kB.



(*) Disk are not byte addressable, sector addressable.

Disk consist of a large no of addressable sectors, (512 Bytes)

fetch = inode no 32
size of inode 256 byte
size of block = 4 kB.

sector address is the physical address of the inode block:

$$\text{block} = \frac{\text{inode no} \times \text{size of inode}}{\text{block size}}$$

$$= \frac{32 \times 256}{4}$$

$$= \frac{8192}{4}$$

$$= \frac{8\ kB}{4\ kB} = 2$$

আসছে page 1 block 2.

$$\text{sector address} = \frac{(\text{block no} \times \text{block size}) + \text{inode table start address}}{\text{sector size}}$$

$$= \frac{\{(2 \times 4) + 12\} kB}{512 B}$$

$$= 40.$$

Video 3

Directory Organization / Structure

→ contains a list of (entry name, inode no) pairs

→ each directory has 2 files

{file path is directory}

i) . for current directory

ii) .. for parent directory.

for example

Directory has 3 files. (foo, bar, foobar)

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

Free Space Management

→ file system track which inode and data block are free or not.

→ In order to manage free space, we have 2 simple

bitmaps.

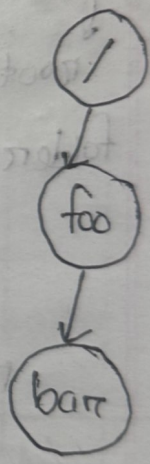
↳ when file is newly created, it allocated inode by searching the inode bitmap and update on disk.

↳ Pre-allocation policy is commonly used for allocate contiguous blocks.

Access Paths: Reading a File from Disk

issue an open ("/foo/bar", O_RDONLY)

↳ root directory



traverse recursively the path name until the desired inode 'bar'.

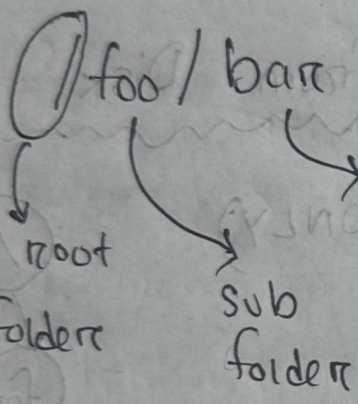
→ issue read() to read from file {system call}.
read the first block of the file, consulting the inode to find the location of such a block.

- 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 allocated, but for now, that is all the file system really needs to do.

No disk / I/O take place.



root ও parent ও নিচ্ছি।

read() → call once for each block.

target is to read bar.

file read() করতে গেলে আগে open() করতে হয়।

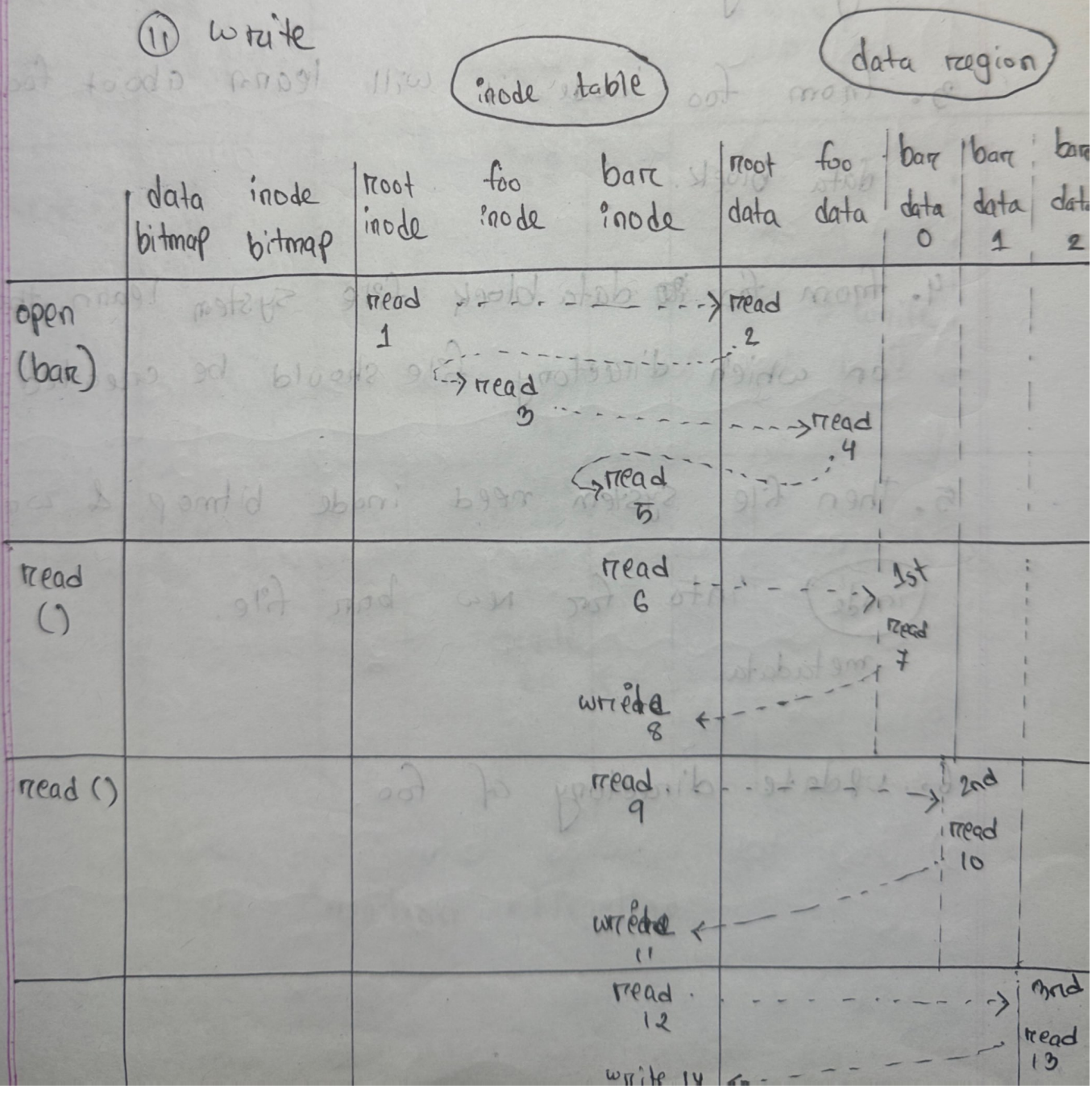
1. open file → read inode of file
2. to find inode no of bar, read datablock of parent directory.

3. to find data block of x read 'inode' of x.

- root: 2? inode no fixed
- /foo/bar → create bar and keep 3 blocks.

① Create

⑪ write



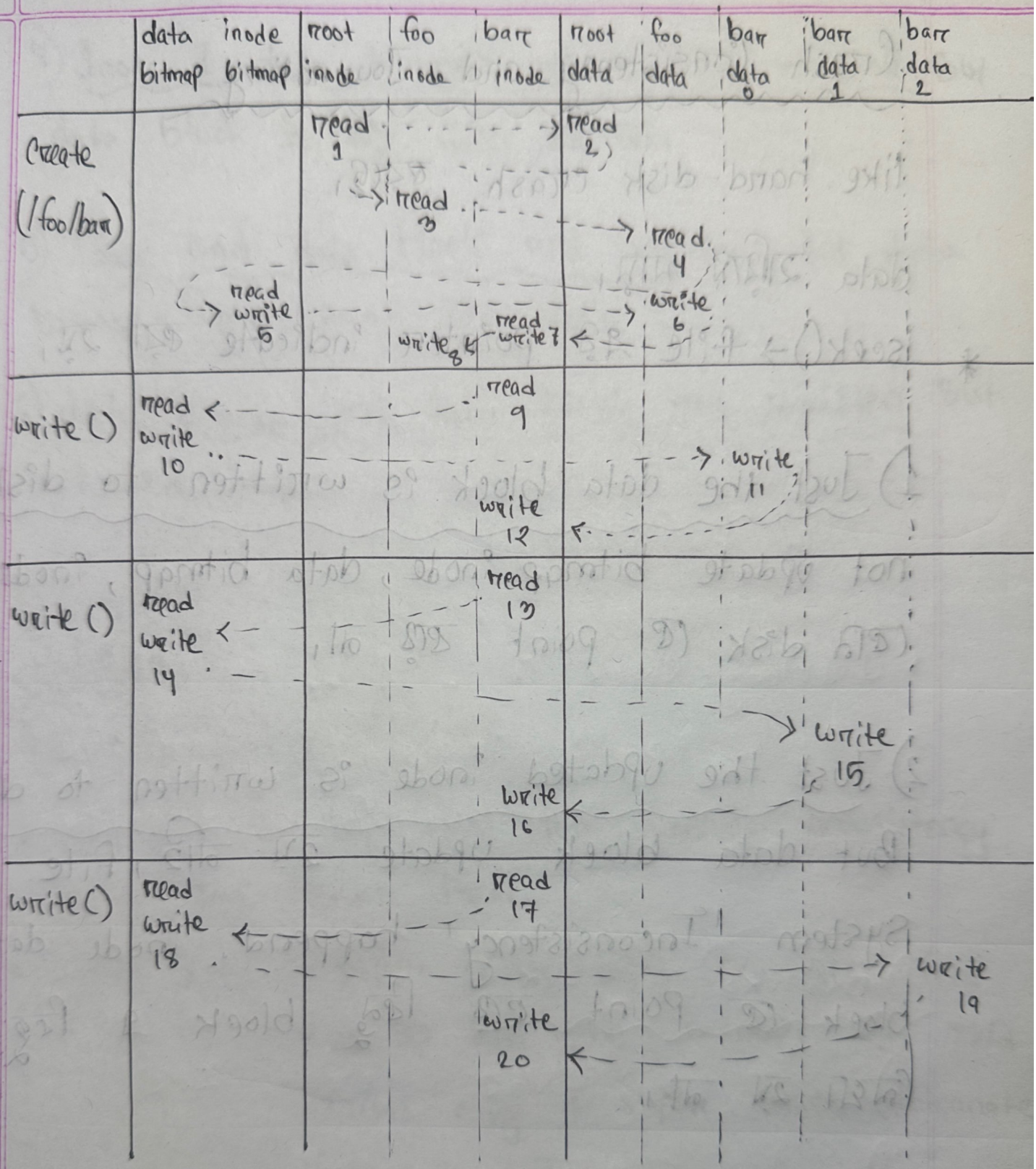
File Read Timeline.

Access Paths : Writing to Disk

1. reading root inode and retrieve into root data block
2. From root data block it will retrieve into regarding foo inode.
3. from foo inode it will learn about foo data block.
4. from foo data block file system learn that on which directory file should be created.
5. Then file system need inode bitmap & write inode into for new bar file.

metadata

6. update directory of foo.



File Creation Timeline.

Crash Consistency and Journaling

like hard-disk crash করছে।

data হারিয়ে যায়।

* lseek() → file এর pointer indicate করা হয়।

1) Just the data block is written to disk. But not update bitmap, inode, data bitmap, inode কোন disk কে point করে না।

2) Just the updated inode is written to disk. But data block update হয় নাই, file system 'Inconsistency' happend. inode data block কে point করে কিন্তু block এ কিছু লিখা হয় না।

3) Just the updated bitmap is written to the disk.
bitmap indicates that data block is allocated but inode কোনো data block কে point করে না। Space leak হয়।

4) Inode & data bitmap are written to disk. But data block এ কিছু লেখা হয় না।

5) Inode and data block are written. But data bitmap কিছুকে point করে নাই।

6) data bitmap and data block are written. But not the inode.

Solution

The file System checker

* fsck → is a unix tool for finding inconsistency and repair them.

* Super block → super block copy করে রাখে। file size বড় হলে file corrupt হয় তখন আগের copy of super block alternate হিসেবে use হয়।

* free blocks → inode, indirect blocks, double indirect block গুলো check করে দেখে কোনো block free আছে কিনা।

* inode state → consistancy check করা

* inode link → check if the reference coont of each inode is consistent.

→ check if a block is shared by two inodes.

→ check for 'bad' block pointer is the one that points to the location that lies outside the file partition.

* directory: check if . & .. are properly set up. Make sure that there are only one hard link for a directory.

Drawbacks of FSCK

i) অনেক slow. অনেক overhead হয়ে

ii) Notun solution journaling.

iii) Very complex knowledge to build.

Video: 04

Journaling

A থেকে B তে file copy করতে চাইলে file system তখনি ৪ জায়গায় লিখে রাখবে যে A point থেকে B point এ copy হবে। Direct copy করে না। লিখার পর copy করার কাজ শুরু হবে / করে। This structure that is the destination of the 'write ahead' is called "log."

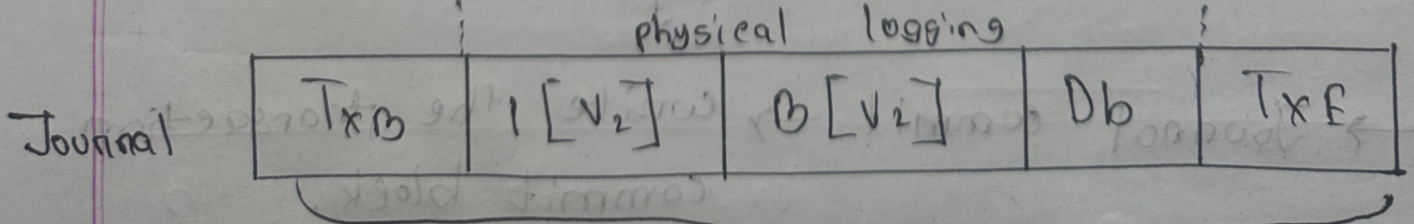
* Journaling itself has a super-block.

Super	journal	inode	Data bitmap	Data block.

* যা update হবে তখনি, journal এ update করে রাখে।

TxB → entry in journal. Transaction begin.

TxE → out in journal. Transaction End.



এখন এগুলো legit Disk এ update হবে।

Replay → when failure happens at the time of TxE. Then, after the failure, we have to update the things of journal in DISK again.

Disk এর মধ্যে Disk scheduler থাকে, can reorder the operation. And, the block loses.

To avoid being inconsistent.

When, inode and Data bitmap matches with each other, then it is consistent.

Journal doesn't care about data.

→ 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.

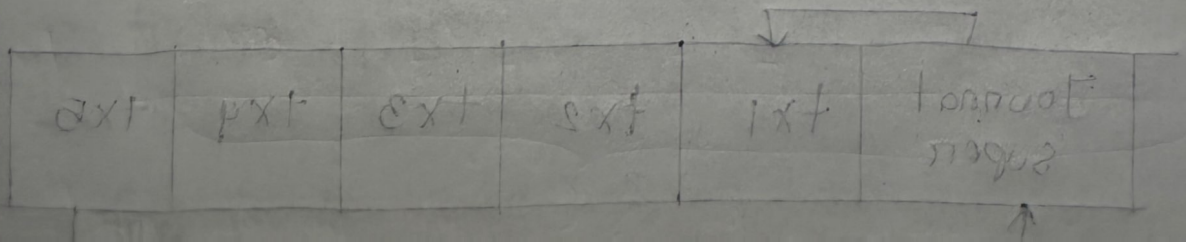
Recovery

→ if the crash happens, before the transaction is written to the log then the pending update is skipped.

→ if the crash happens after the transaction is written to the log, but before the checkpoi

recover the updated as follows.

- Scan the log and look for transactions that have committed to the disk.
- transactions are replayed.



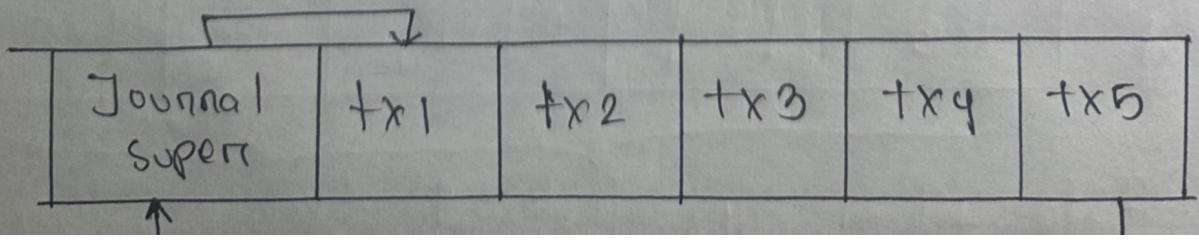
Batching Log Update

- 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 infinite

The Journal is a circular linked list.

Journal has a super block.



Journal does not include Data blocks because it doesn't care. So, space reduces of data blocks. This is MetaData Journal. Means information of Journal.

"circular log"

= (index + 1) % array - length

Metadata Journaling

Because of the high cost of writing every data block to disk twice.

- commit to log journal
- checkpoint to on-disk location.

→ Data write
→ journal metadata (inode & bitmap) write
→ journal commit
→ checkpoint metadata
→ free

* To start writing in Disk is called issuing.

Tricky Case: Block Reuse

directory

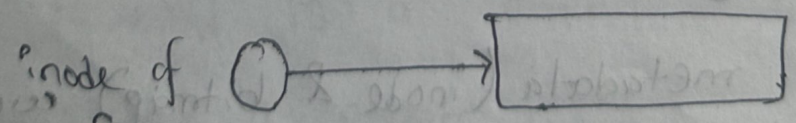
foo inode

direct : 1000

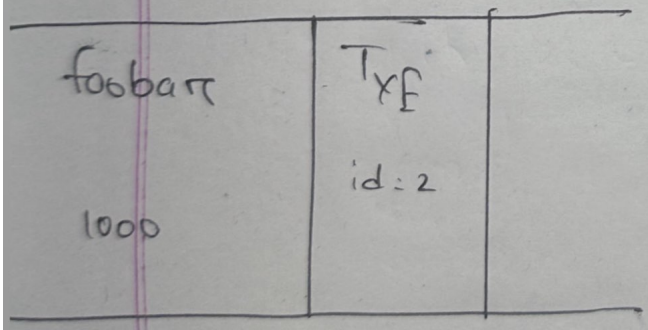
owner : CSE 321

modified :

TxB id=1	inode (foo) bitmap 1000	Data (foo) bitmap 1000	TxE id=1	TxB id=2



data of foo is deleted, but the inode is still there. After power on, 1000 data bitmap is empty as foo is deleted. But, inode is there. So, we need to remove the inode too, like pointer. Then assign a file to it.



Now,