CS3.301 Operating Systems and Networks

Persistence: File System Implementation

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Software Engineering Research Centre

Acknowledgement

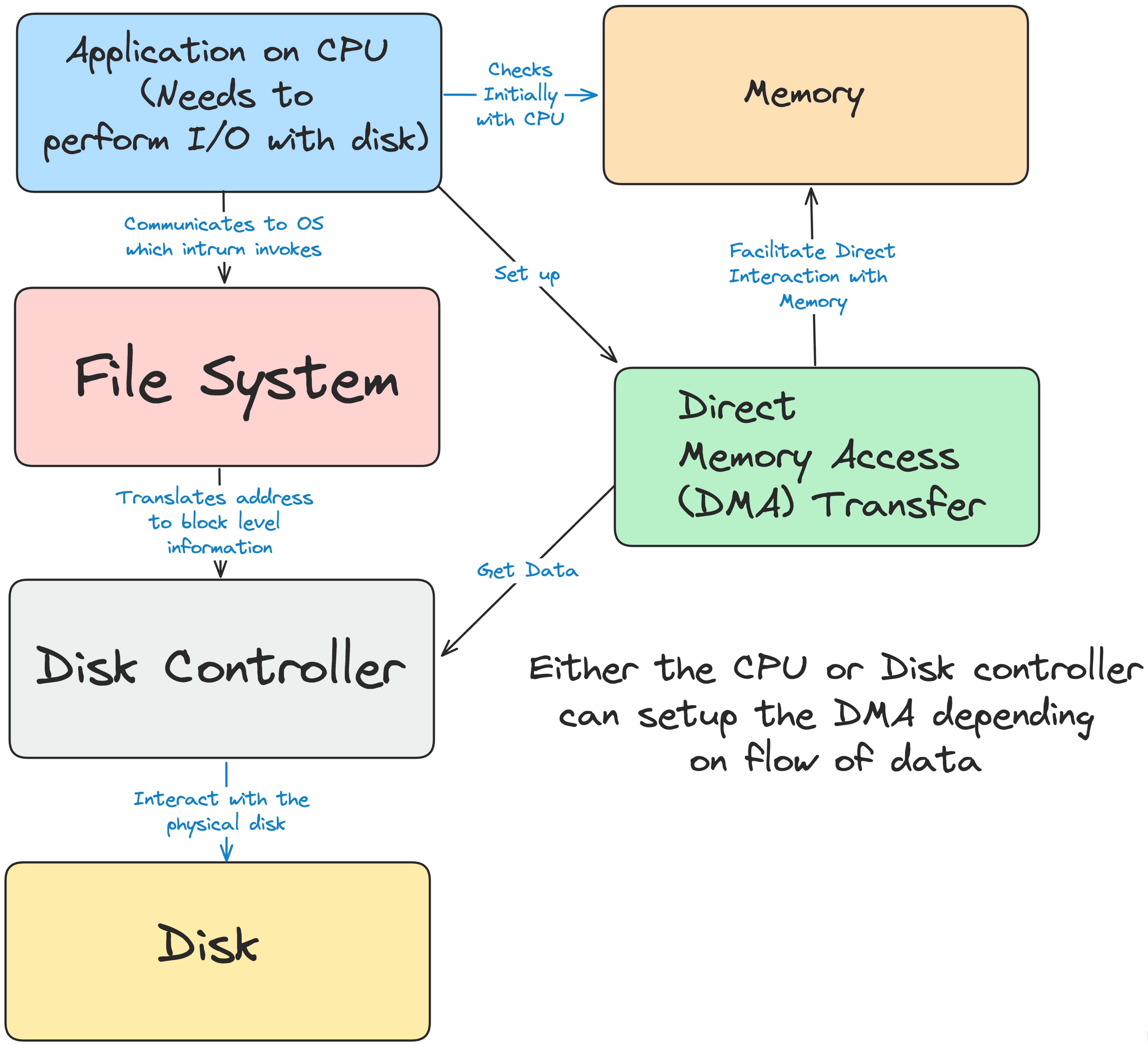
The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in Three Easy Pieces by Remzi et al.
- File System implementation by Youjip Won, Hanyang University

The flow of access

- Application performs read or write to a file
- CPU communicates to OS which invokes the File System (FS)
- The OS may check in its cache if its already there
- FS prepares block level information to disk controller
- A Direct Memory Access (DMA) is set up
- Disk controller performs the physical read or write based on commands from DMA and file system
- If its read, Disk -> DMA, for writes, DMA -> Disk



Virtualization of Storage

- Just like memory, storage is virtualised
  - Supported by file system
  - User does not see disk but everything is through two major abstractions
- **Two Key abstractions**
  - Files
  - Directories

Image source: Dalle-3

Metadata of files

- File system stores fair amount of data about files
- Information include: file size, last access, last modified, user id of the owner, links count, pointers to data blocks, etc.
- This metadata is stored by file systems in a structure called **inode**
- **Inode** - persistent data structure used by the file system
  - They store all the metadata information for a file
  - They are stored in the disks but copies are cached to main memory when needed!

How can we build a simple File System?

What structures are needed in disk and how to access?

File System

- Organization of files and directories on disk
- OS has one more file systems
- File system is **pure software,** features:
    - Provide support for the sys calls
    - Manage the storage of data
    - No additional hardware support
- Great deal of **flexibility** when building FS
- Details vary with various file systems

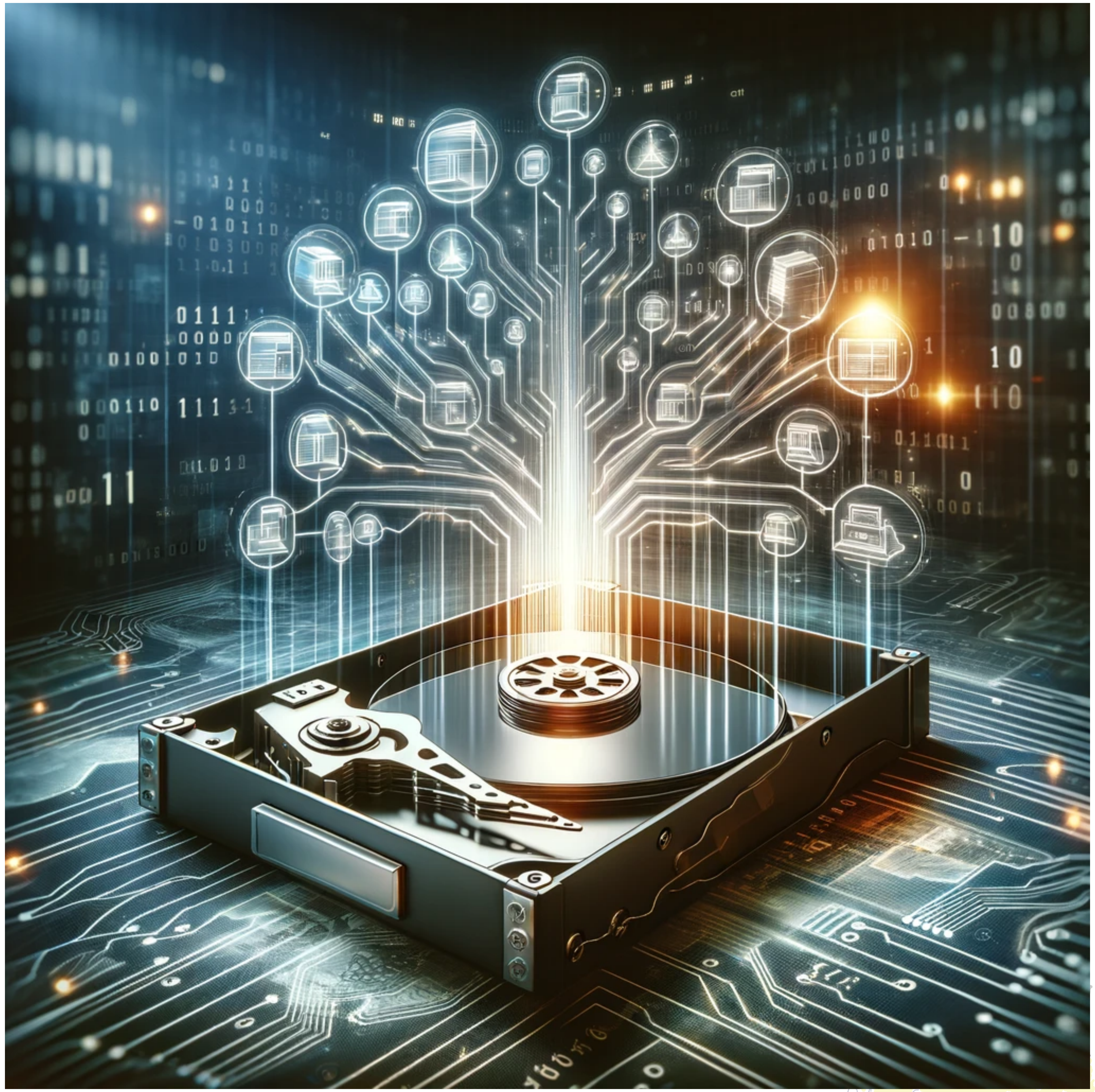


Image credits: Dalle-3

Breaking down into two main aspects

- Lets try building a simple file system - **Very Simple File System (VSFS)**
- In any FS, two key things make the difference

Data Structures

- What types of on-disk data structures are utilized by the file system to organise its data and metadata?
- VSFS can make use of simple structures like array of blocks (complex ones: trees)

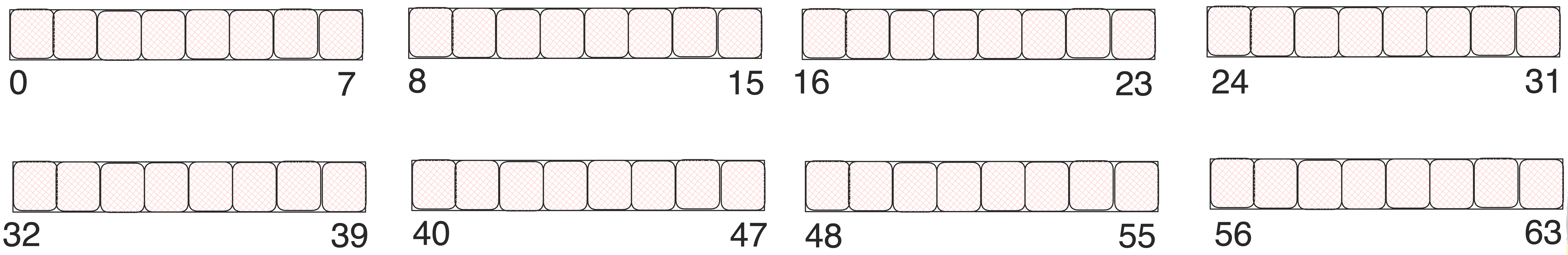
Access Methods

- How can the calls like open(), read(), write(), etc made by process be mapped?
- Which structures are read during the execution of a system call?
- What about the efficiency?

Data structures

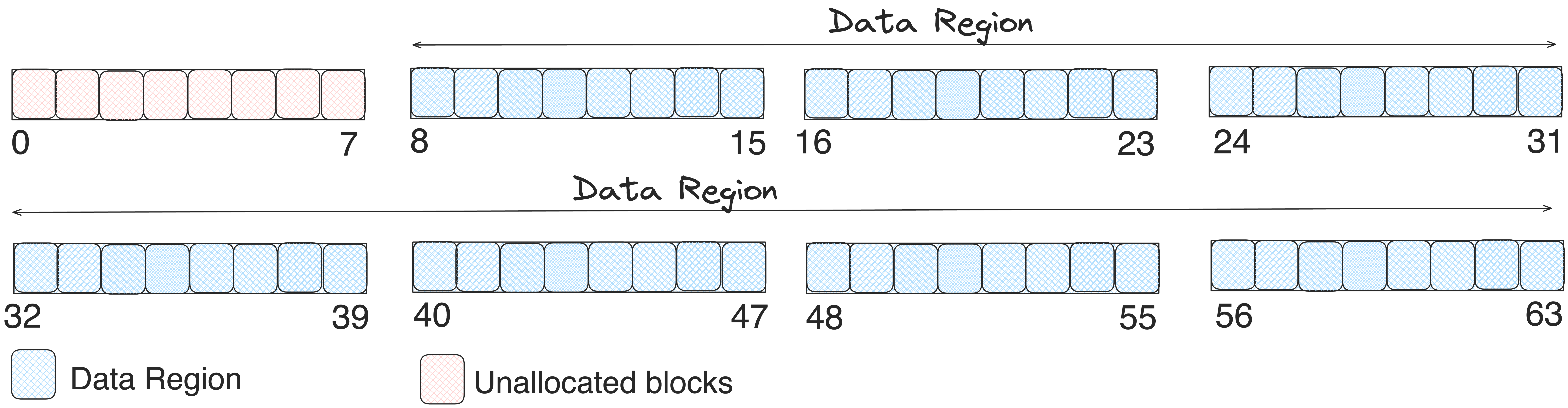
On-disk organisation of VSFS

- Remember: Disk exposes a set of **blocks**
- File system has to organise the files into blocks - **Data**
- The information about the files also have to be stored - **metadata**
- Consider a disk with 64 blocks, each of size 4 KB (same sized blocks)
    - 0 to 63 in general **0 to N-1**
    - **What needs to be stored in these blocks?**



Data Region in the File System

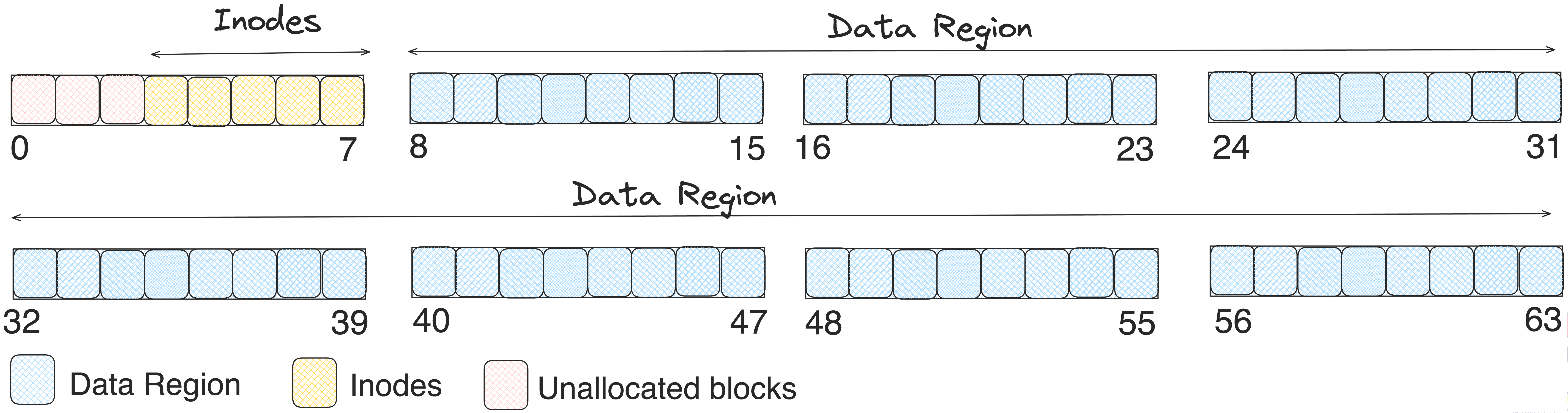
Some blocks needs to be reserved for storing data - **data region**



- More information needs to be stored about where the data blocks are located, type of file, etc
- The inodes need to be stored

Some Space for Inodes!

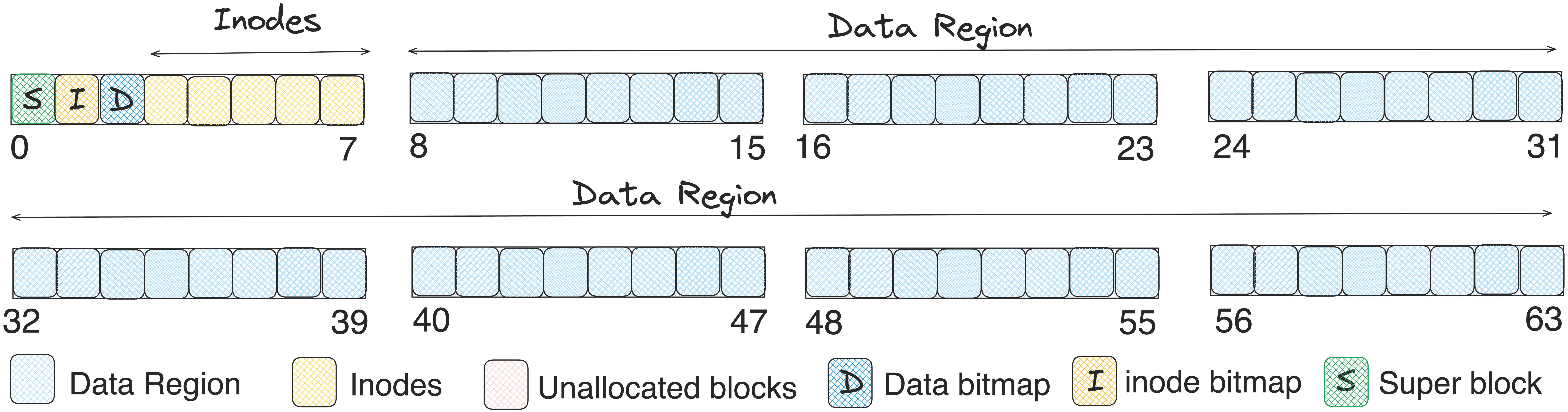
- Dedicate some space for inode table
  - This can hold an array of on-disk inodes
  - Consider each inode takes 256 bytes and 5 blocks are dedicated
  - Each block can hold 16 inodes => file system can hold 80 files



We still miss something!

- FS needs some mechanism to track which inodes are free and which data blocks are free
- How can such information be tracked? Which are free and which are available?
    - Use **bitmaps**, each bit can be used to denote if corresponding block is free or not
        - 0 if the corresponding block is free
        - 1 if the corresponding block is allocated
    - In our vsfs - 80 inodes and 56 blocks for data
    - Assume that we dedicate **two blocks for bitmaps** for **inode** and **data**

A more complete representation



- **Super block** holds the entire organisation of all other blocks
  - Which blocks are inodes, which are data blocks, where does data block start, where Inode begins, type of file system, etc
  - During the mount, OS reads super block to initialise various parameters

File Organization: The inode

- Each inode is referred to by the inode number
  - Using inode number, FS can locate inode, eg: inode number: 32
  - Calculate offset into inode: 32 X (sizeof(inode)) = 32 * 256 = **8192 => 8 KB**
  - Add offset with start address of inode = 12KB + 8KB = **20KB**

What does inode contain?

- inode contains all the information about a file - The metadata
    - File type (regular file, directory, etc.)
    - Size, number of blocks allocated to it
    - Protection information (who can access, what access, etc.)
    - Time information (modified time, access time, etc)
    - Many more

Simplified EXT2 inode

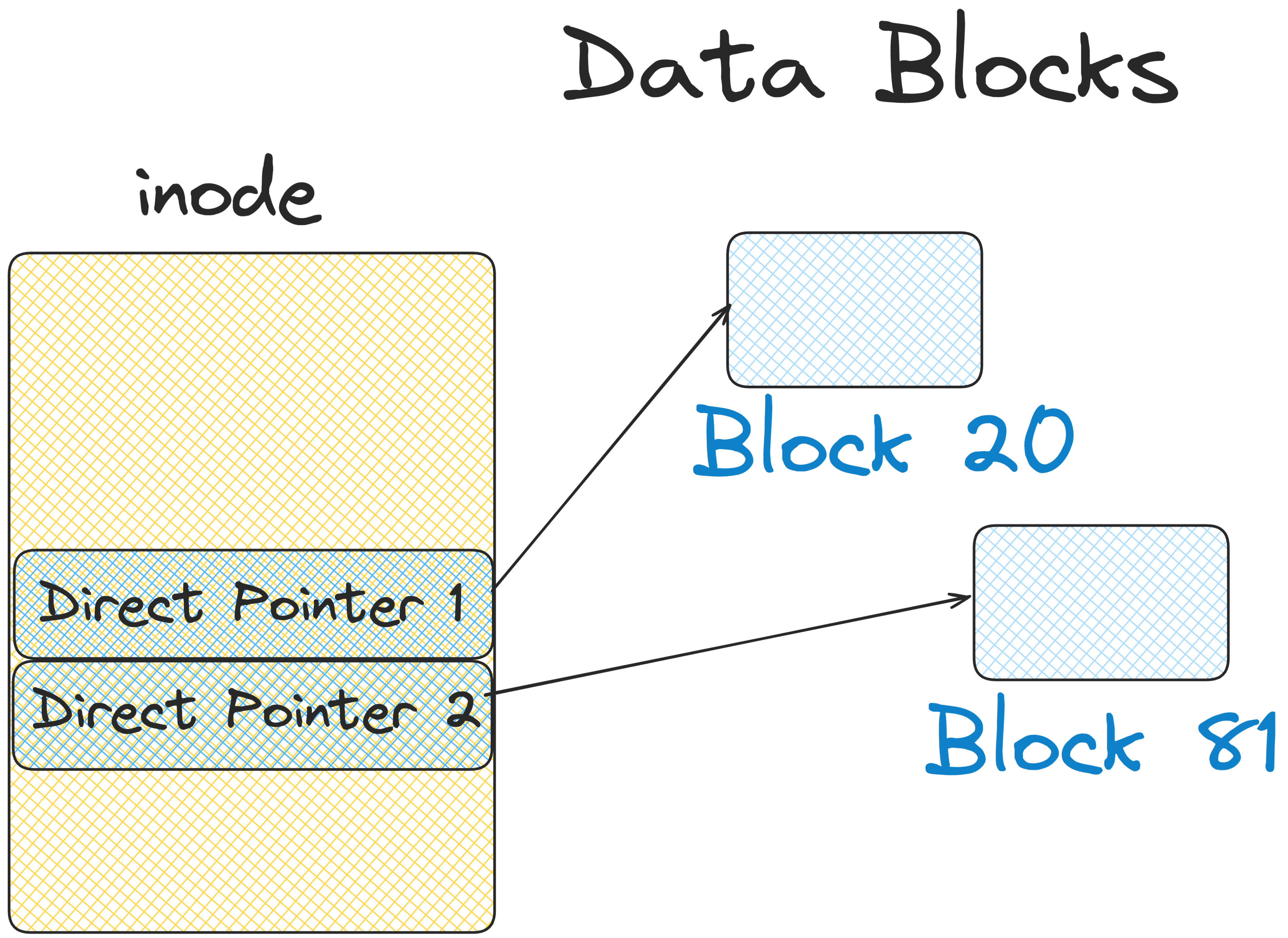
Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
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

Total 128 bytes

How can inode get to data blocks?

More about inodes

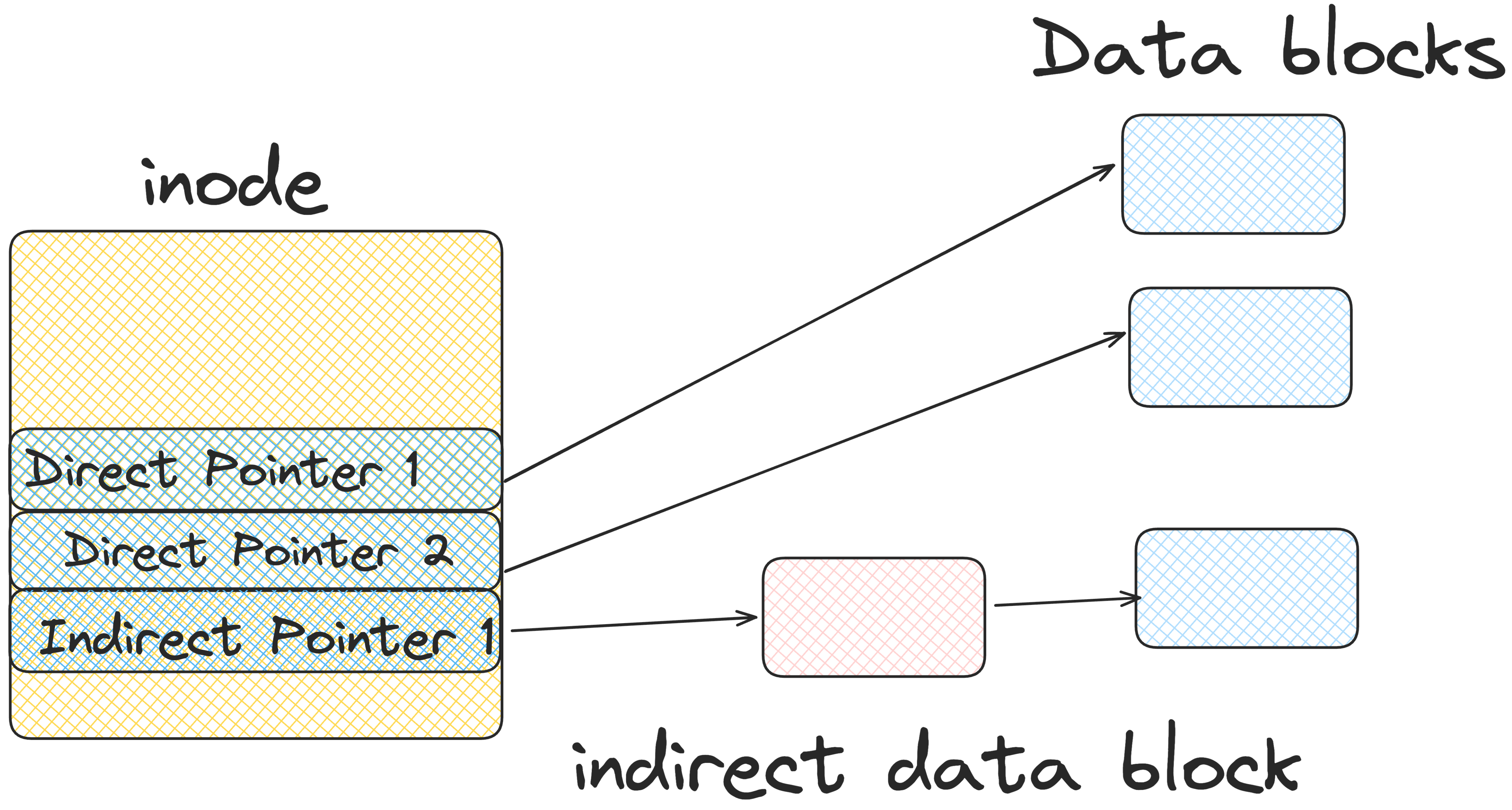
- Each inode needs to track disk block numbers of a file
- File data is not stored contiguously on disk
  - How to track multiple block numbers of a file?
  - Store pointer to the block inside the inode
  - Numbers of first few blocks are stored in inode itself
  - Each pointer can point to the location in the disk block - **direct pointers**
  - **What if the file size is large?** - How many block numbers can i-node store?
    - Need for better mechanism



Size of one block is 4 KB here!

Indirect Pointers

- To support large files, few direct pointers may not suffice!
- Use a special pointer - **indirect pointer**
  - Point to a block that contains more pointers - **indirect data block**
  - Each of the pointer can further point to data blocks
  - The indirect block is allocated from the data region
  - Inode array may have 12 direct pointers and one indirect pointer

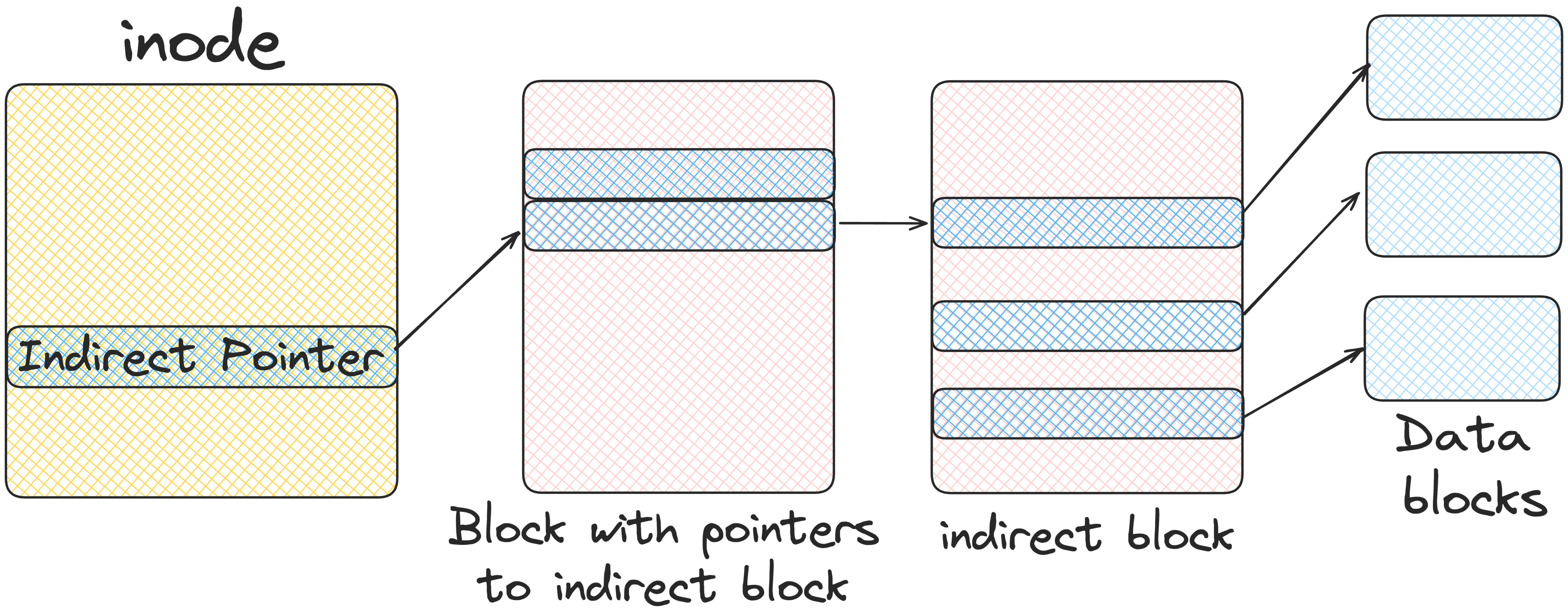


How much files can be supported?

Having one indirect pointer

- Each block is 4 KB
- Each inode can contain 12 direct pointers => 12*4 = 48 KB of file can be addressed
- 1 indirect pointer points to a block of size 4 KB
  - Each address takes around 4 bytes
  - Indirect blocks can have around 1024 pointers (4 KB / 4)
- Total size of file that can be addressed = (12 + 1024) * 4K = 4144 KB
- What if the file is even larger? How can the inode capture all the blocks?

The Multi-Level Index



- **Double indirect pointer**: Points to a block with pointers to indirect block
  - Each of the pointers in indirect block points to data blocks
  - Size now that can be supported is 1024*1024*4 ~ 4GB
- For more even **triple indirect pointers** can be sought of

Why this direct and indirect pointers?

- One finding over many years of research: most of files are small
- Thus with small number of direct pointers, inode can point to 48 KB of data
- All that is needed is one or few indirect blocks

Most files are small	˜2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contain lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

"A Five-Year Study of File-System Metadata" by Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch. FAST '07, San Jose, California, February 2007.

What about Directories?

- Directory stores the mapping of file names and their inode numbers
- Each directory has two extra files
    - "." for current directory and ".." for parent directory
    - Assume that a directory "OSN" has three files (l01, l02, lect03)
- Directory is a special type of file and has inode and data blocks (stores file records)

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	l01
13	12	4	l02
24	36	7	lect03

inum - inode number
reclen - total bytes for name
strlen - length of the name
name - actual name

Free Space Management

- FS has to keep track of which inodes and data blocks are free
- Multiple methods can be used and many design choices exist. Eg:
  - Use **bitmaps** for inodes and data blocks, store one bit per block to indicate free or not
  - **Free list:** Super block can store pointer to first free block which can then point to next free block and so on.
- Eg: Linux FS such as ext2 and ext3 checks for sequence of blocks on new file creation
  - Sequence of data blocks are allocated contiguously for performance
  - Pre-allocation policy is commonly used heuristic when allocating data blocks

Access: Reading File From Disks

- FS also needs better ways of managing access to file (apart from data structure)
- Eg: FS has been mounted and read issued to */OSN/l01* - open, read, close
- Assume that file size is 12 KB (3 blocks in size)
    - sys call open("/OSN/l01", O_RDONLY)
- Intuitively: FS must traverse the pathname and locate the file
    - What will be the process to achieve this?

Opening Files

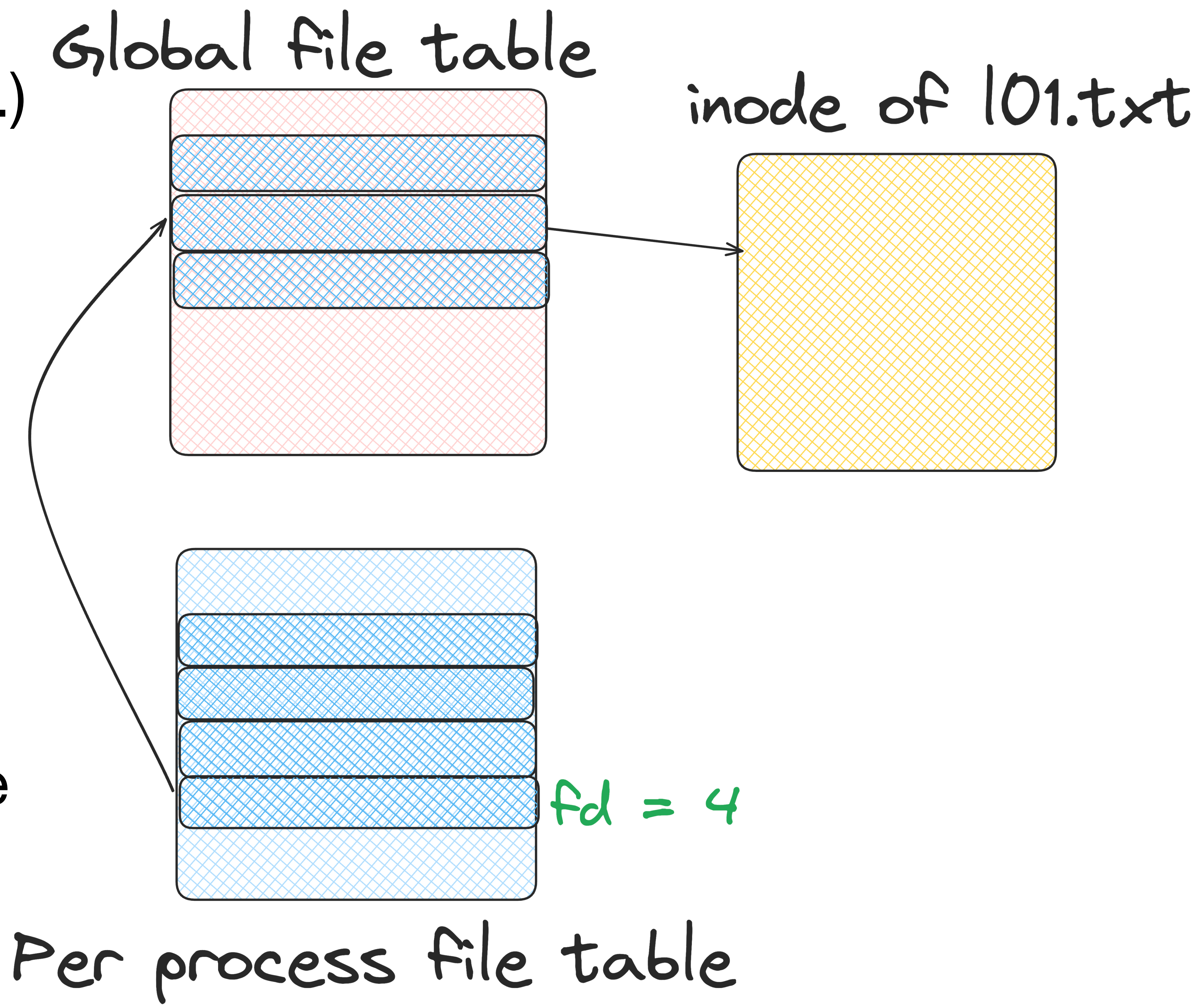
- First part of read is always open sys call - Why?
  - Take the inode and load it in the memory for future operations
  - Open returns file descriptor which points to in-memory I-node
  - Reads and writes can access file data from I-node
- Assume a sys call *open(“/OSN/lectures/l01.txt”, O_RDONLY)*
  - Traverse the path name and then locate desired inode
  - Begin at the root of the FS (/), root inode number is 2 in Unix FS (mostly)
  - FS reads the block that contains inode number 2

Opening Files

- Recursively: Read the data blocks of root directory, find the name “lectures” and get its inode number
  - Get inode of lectures -> get inode number of “l01.txt” -> get inode
  - Keep repeating the process until the end of the path
- Read inode of “l01.txt” into memory, make final permission check
- Allocate file descriptor for this process and return file descriptor to user
  - Allocation will be done in the in-memory **open file table.** It will be updated for each read - offset
- In the case of new file, new inode and data blocks will be allocated using bitmap and update directory entry

Open File Table

- Kernel uses a set of data structures to track all open files
- **Global open file table**
  - One entry for every open file (stores also sockets, pipes, etc.)
  - Entry points to the in-memory inode of the file (remember opening of file)
- **Per-process open file table**
  - Array of all the files that the process has opened
  - File descriptor is index into the array
  - Per process file entry -> global file table entry -> inode of file
  - Every process has three files (stdin, stdout, err) open by default
- Open system call creates entries in both table and returns file descriptor number



Reading a File

- Make a call read() to read from file
  - Read in the first data block of the file with help of inode
  - Update the inode with last accessed time
  - Update in-memory open file table for file descriptor, file offset
  - Repeat the process for reading each block of data
- Once file is closed
  - Just the file descriptor should be deallocated - No disk I/O

Reading a File From Disk

Timeline	data bitmap	inode bitmap	root inode	lectures inode	101 inode	root data	lecture data	101 data [0]	101 data [1]
open ()			read						
						read			
				read					
							read		
					read				
read ()					read				
								read	
					write				
read ()					read				
								read	
					write				

Writes to a File

- Make a call write() to write into the file on the disk
- Data block may have to be allocated (if not overwriting)
  - Need to update data bitmap and data block
  - Total of five I/O:
    - One to read data bitmap
    - Write to data bitmap
    - Two more to read and write the inode
    - Write to the actual block itself
- In case of creation of new file, number of I/Os can go really high!

Writing a File To Disk

Timeline	data bitmap	inode bitmap	root inode	lectures inode	101 inode	root data	lecture data	101 data [0]	101 data [1]
create () /lecture/101			read						
						read			
				read					
							read		
		read write							
							write		
					read write				
				write					
write ()					read				
	read write								
								write	
					write				

Can we do something about performance?

- Reading and writing files are expensive
- Imagine opening and reading a file by providing a long path
  - Each inode needs to be fetched, corresponding data then read of files
  - Can go upto 100s of I/Os
- Use the concept of caching and buffering
  - Use system memory to cache important blocks - **Minimise overheads!**
  - Early FS, used **fixed-size cache** -> store popular blocks (10% at boot time)
  - Use strategies like LRU to evict blocks

Caching and Buffering

- Static partitioning of memory is not always useful - **Wastages!**
- Modern systems employ dynamic partitioning approach
  - Integrate virtual memory pages and FS pages into unified page cache
  - First open may generate lot of I/O but subsequent will be in cache!
- Writes is little tricky as at some point the disk has to be accessed to store
  - **Write buffering** - Delay writes to disk, perform batch I/O
  - Schedule I/Os in a particular order for performance gain
  - Writes can be avoided totally - file is created and deleted in few seconds! **(Don't write)**

Caching and Buffering

- Applications like DB avoids caching altogether - direct I/O
  - System calls like fsync() allows writes to be pushed immidiately
  - Unexpected data loss may happen since data is in memory
  - Has impact on overall system performance
- At the end its all about trade-off’s
  - Durability vs Performance tradeoff
  - Has big dependance on the application
    - Browser vs Transactional database!

SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn

Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere