CS3.301 Operating Systems and Networks

Persistence: Files and Directories

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Software Engineering Research Centre

Acknowledgement

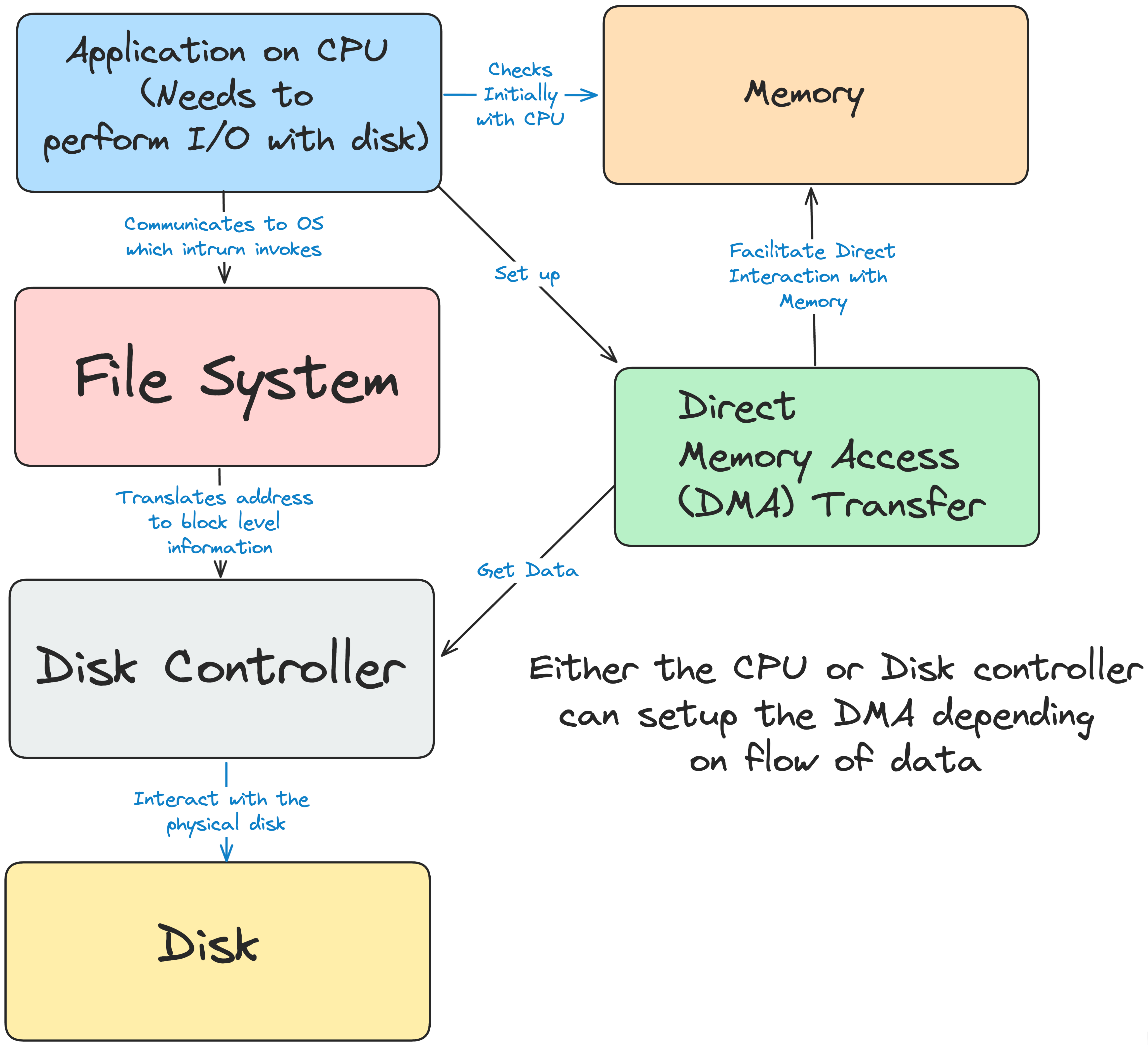
The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in Three Easy Pieces by Remzi et al.

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



So far!

- **Devices for Persistence**
  - Hard disk - Simple interface, store data in magnetic disks
  - RAIDs provide support for improved capacity, performance and reliability
- **What we still need!**
  - How to manage a persistence device?
  - What about the APIs?
  - What are some key implementation aspects!

Virtualization of Storage

Image source: Dalle-3

Files

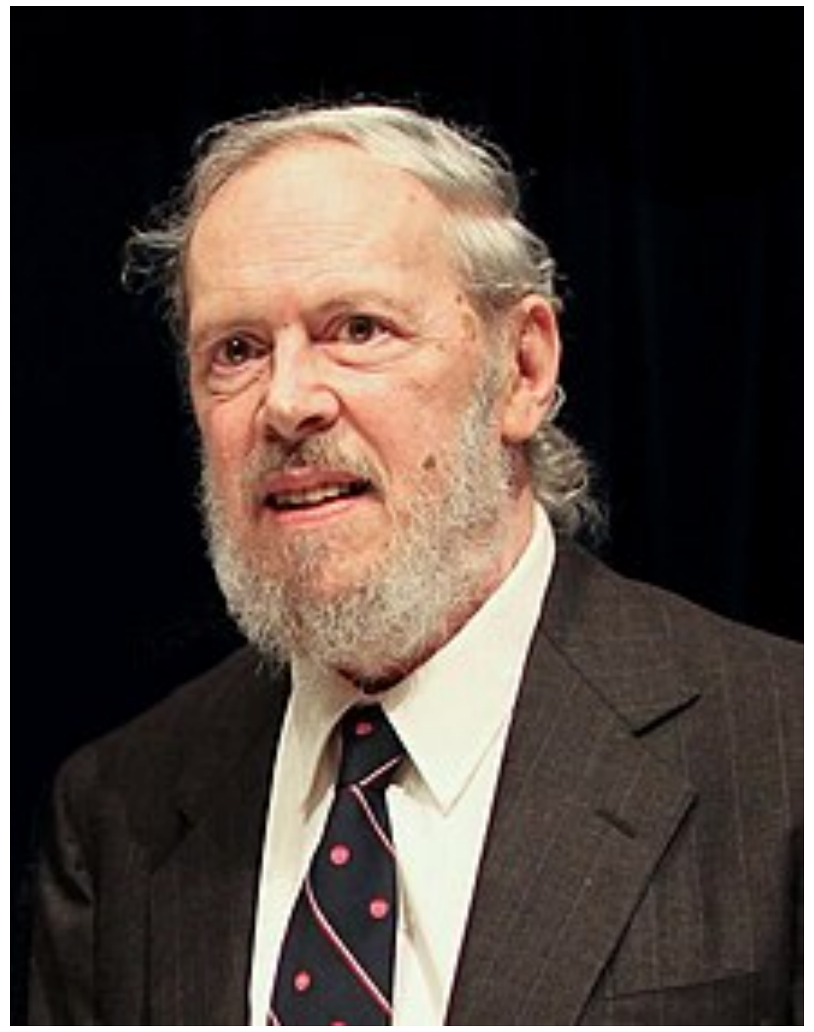
- Linear array of bytes each of which can be read or written
- Each file has a human-readable name - **“sample.pdf”**
- Each file has a unique low-level name (not user given, OS given) - **inode number (i-number)**
- Type of the file is not the concern of the OS (image, code, etc)
    - File system should ensure that data is stored persistently
    - Also ensures that data is retrieved when requested
- Applications can worry about extensions and reading file in the way needed

Directories

- A directory is just like a file
- It also has a low-level name: inode number
- Contains a list of pairs **(user readable file name, i-node number)**
    - Eg: consider a directory name **OSN**
        - (Lectures, 123) -> Directory
        - (OSN_L23.pdf, 326) -> File
- Basically directory is a special type of files with contents: files, directories and corresponding i-node numbers

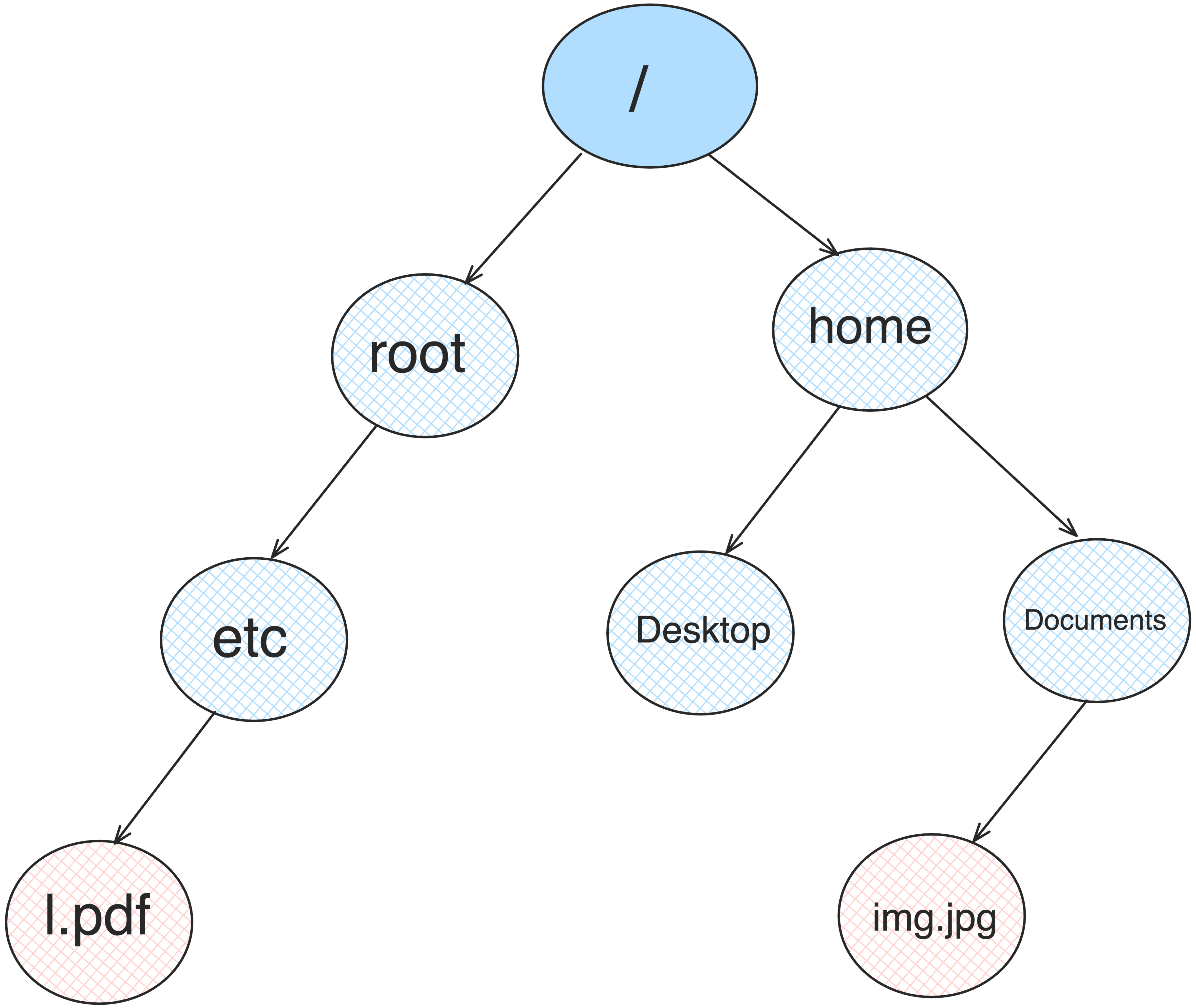
Inode Number - Truth!

“In truth, I don't know either. It was just a term that we started to use. ***‘Index’ is my best guess,*** *because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk…*



Dennis Ritchie

The Unix Directory Tree



- Files and directories arranged in a tree
- Directory hierarchy starts at root directory - referred to as /
- Uses a separator to name subsequent directories
- **Absolute pathname** can be used:
  - /home/Documents/img.jpg
- File has two parts:
  - Arbitrary name - “img”
  - Type - “.jpg”
- Everything is an abstraction by OS

File System Interface

- Everything in Unix is virtually a file
- Mainly the file system has to provide three interfaces
  - **Creation of files** - Support creating files, allocate space
  - **Accessing files** - Reading and writing files
  - **Deletion of files** - Delete files and clear space
- Internally everything is 1s and 0s in the disk so File system has a big responsibility!

Creation Interface

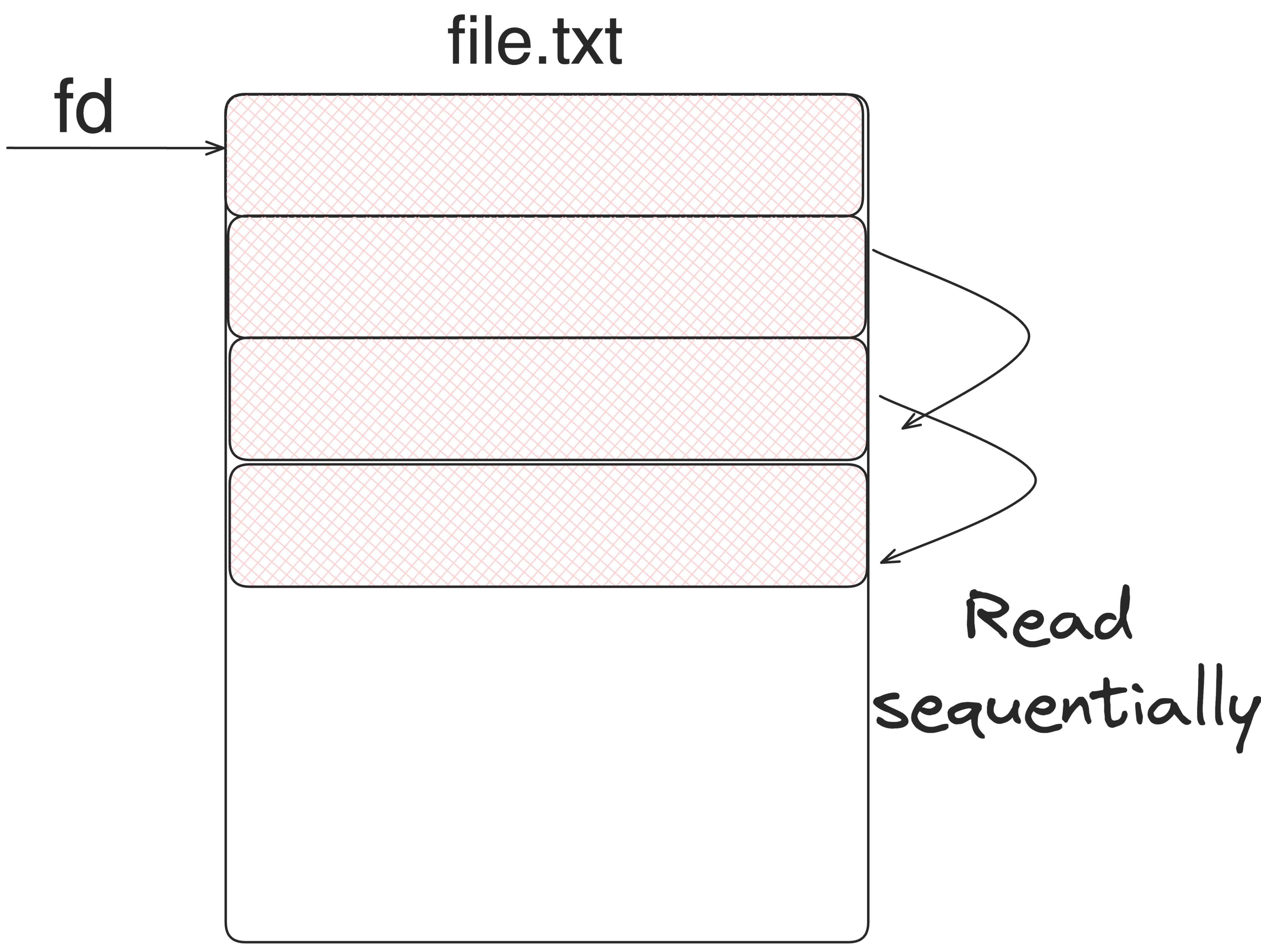
- **open()** system call with flag to create file

int fd = open("sample", O_CREAT | O_WRONLY, O_TRUNC, S_IRUSR | S_IWUSR);

  - O_CREAT: creates a file if it does not exist
  - O_WRONLY: file is write only
  - O_TRUNC: truncates file to zero bytes if it already exists
  - S_IRUSR or S_IWUSR: permissions - make file readable or writeable
- The call returns a number, **file descriptor:** operations on file uses the file descriptor
- Existing files must be opened before they can be read or written
- **close():** closes the file

Access Interface

- *read () / write ()* system calls: Reading/writing files
  - Three arguments: file descriptor, buffer with data, size
  - Buffer - where data will be placed and size - size of buffer
  - Reading and writing happens sequentially by default
  - Successive read/write calls fetches from the offset that is being used
- Every process has three files opened - stdin, stdout, stderr with fd 0, 1 and 2



Random Reading and Writing

- In general file is accessed sequentially
  - Read/write from beginning to end
- What if it needs to be randomly accessed for read/write?
  - *lseek()* system call - seek to random offset
  - Start reading and writing from random offset
  - ***off_t lseek(int flides, off_t offset, int whence);***
    - flides - file descriptor
    - off_t - moves pointer to a given offset,
    - Whence - determines how seek is performed (from an offset, from given + some offset or size of file + offset

lseek has nothing to do with disk seek!

A Simple Example - Normal Read

System calls	Return Code	Current Offset
fd = open(“file.txt”, O_RDONLY);	3	0
read (fd, buffer, 100);	100	100
read (fd, buffer, 100);	100	200
read (fd, buffer, 100);	100	300
read (fd, buffer, 100);	0	300
close(fd);	0	-

- Offset is initialised to 0 when opened
- For each read call, the offset is incremented fixed value - sequentially
- At the end, 0 denotes the read has been completed

A Simple Example - Seeking

System calls	Return Code	Current Offset
fd = open("file.txt", O_RDONLY);	3	0
lseek(fd, 200, SEEK_SET);	200	200
read (fd, buffer, 50);	50	250
close(fd);	0	-

- Offset is initialised to 0 when opened
- lseek sets the offset to 200
- Read call, reads the next 50 bytes and updates offset

There is a buffer - How to write immidiately?

- Regular writes, write() puts the data to buffer => some point it will be written to persistent storage
- This is done for performance enhancement (keep in buffer for 5 to 30 seconds)
- Some applications require more real-time guarantees
- System call: *fsync(int fd)*: returns 0, once write is complete
- Sometimes fsync has to be called on directory itself that contains the file
  - This ensures that file is on disk

fsync example

```c
#include <stdio.h>


int main ()
{
  int fd = open("sample", ...);
  assert (fd > -1);
  int rc = write (fd, buffer, size);
  assert (rc == size);
  rc = fsync(fd);
  assert(rc == 0);
  return 0;
}
```

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!

Interface for Directories

- Directories can also be accessed like files
    - Operations like create, open, read, close
- Create directory - **mkdir()** system call, when created its empty. It has two entries
    - “.” And “..” Itself and the parent directory respectively
- Listing all the directories - **ls** command (internally - opendir(), readdir() and closedir())
- What about **rm** * and **rm -rf** * ? - Powerful double-edged sword!
- Directory entry contains information such as name, I-node number,
- Deleting directory - **rmdir ()** - System call and command have same name

Hard Links

- Hard linking creates another file that points to the **same i-node number** (hence same underlying data)
- Assume a file, “file1” which just contains a string “test” - What if we need file2 linked to this?
- Another file that links to this can be created using **link()** call - **ln** command
- Essentially both files have same underlying data - just two different user-given names
- I-node maintains a **link count,** file deleted only when no further links to it
- One can only unlink file, OS decides when to delete

Hard links

```
prompt> echo hello > file1
prompt> cat file1
hello OSN Students
prompt> ln file1 file2
prompt> cat file2
hello OSN Students
```

Hard links

```
prompt> rm file1
removed "file1"
prompt> cat file2
hello OSN Students
```

Symbolic Links or Soft Links

- Another way to create link - This time in much simpler
    - Hard links are limited - link to directory not possible
    - Hard link to files in other disk not possible
    - I-node is unique within a file system
- **Symbolic link or soft link** creates a file by itself
    - The name can be different
    - **i-node number will be different**
    - If the main file is deleted, link points to an invalid entry: **dangling reference**

Soft Links

```
prompt> echo "Hello OSN" > file1
prompt> cat file1
Hello OSN
prompt> ln -s file1 file2
prompt> cat file2
Hello OSN
prompt> rm file1
prompt> cat file2
cat: file2: No such file or directory
```

Beyond Files and Directories

- Mounting a file system connects the files to specific point in the directory tree

mount -t ext3 /dev/sda1 /home/users

ls /home/users

- Assembling directory tree from underlying file system
  - Accomplished by mounting the file system
  - Two tasks: **making the file system** and **mounting**
- Several devices and file systems are mounted on a typical machine
  - Can be accessed with mount command

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

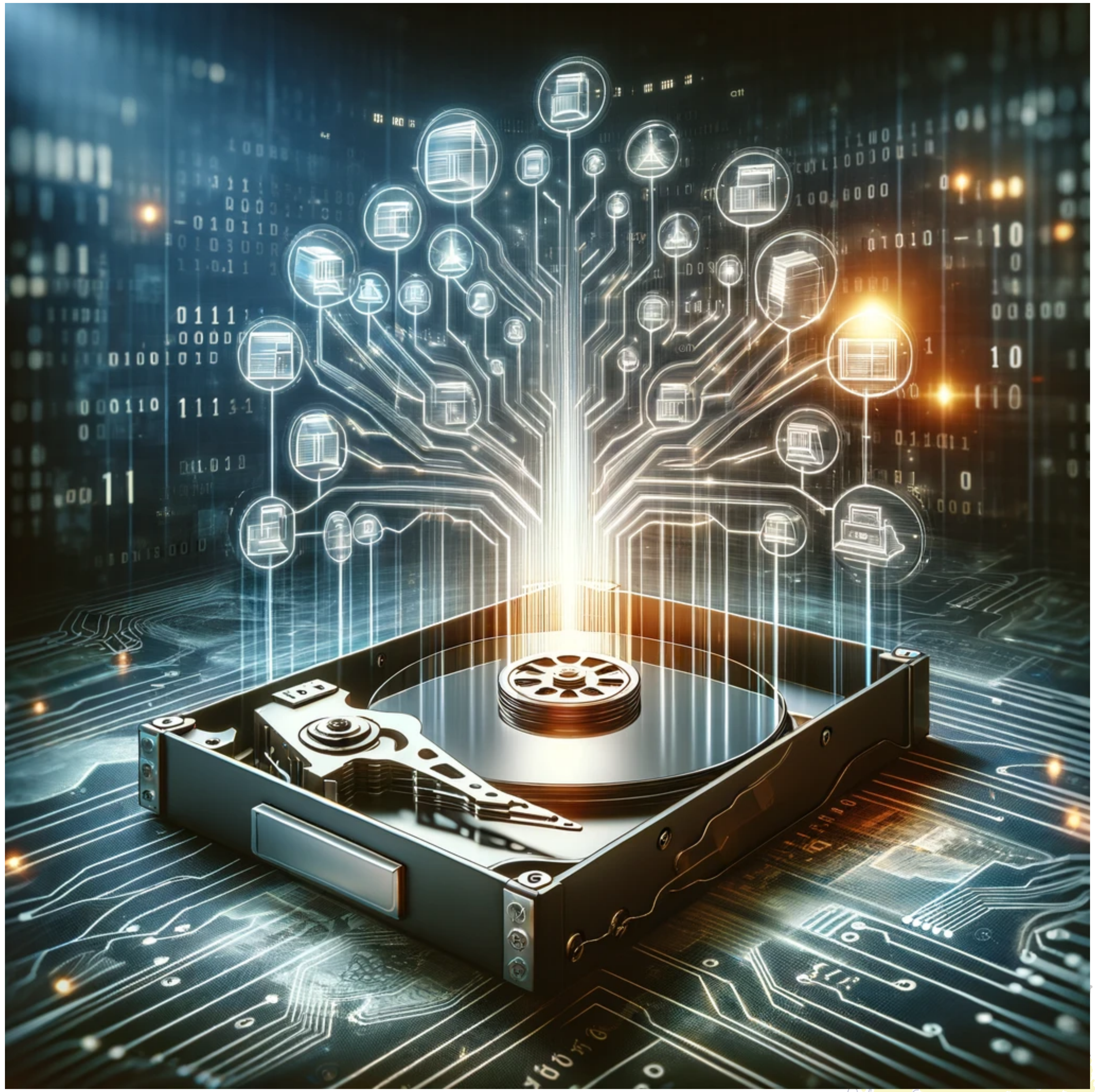


Image credits: Dalle-3

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)

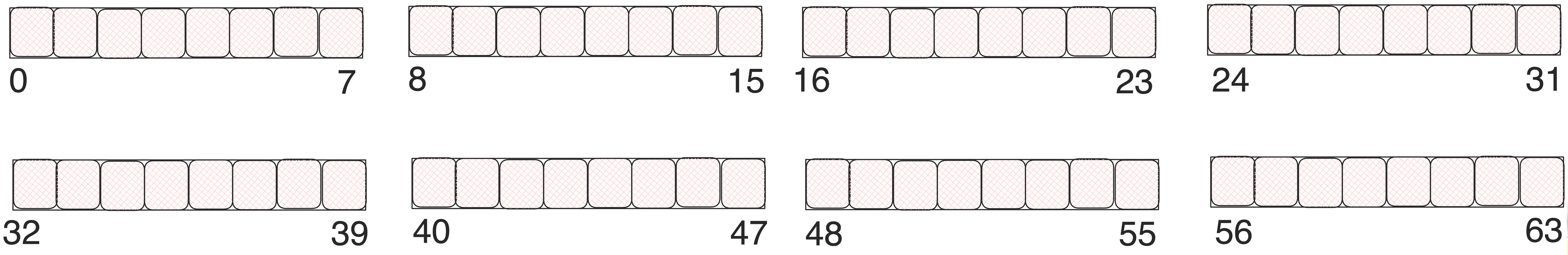
Access Methods

- 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

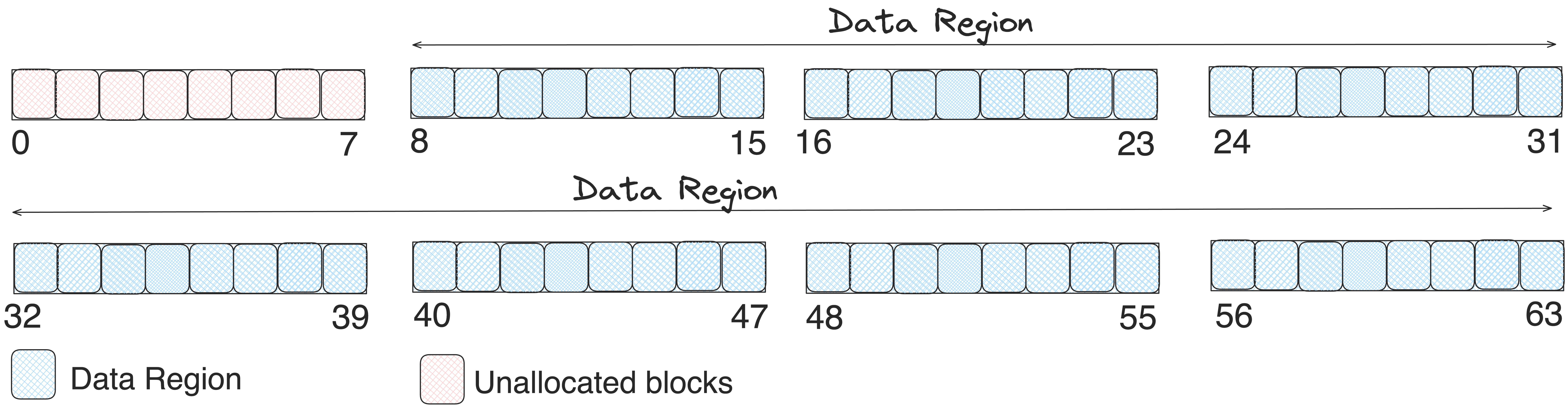
On-disk organisation of VSFS

- 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

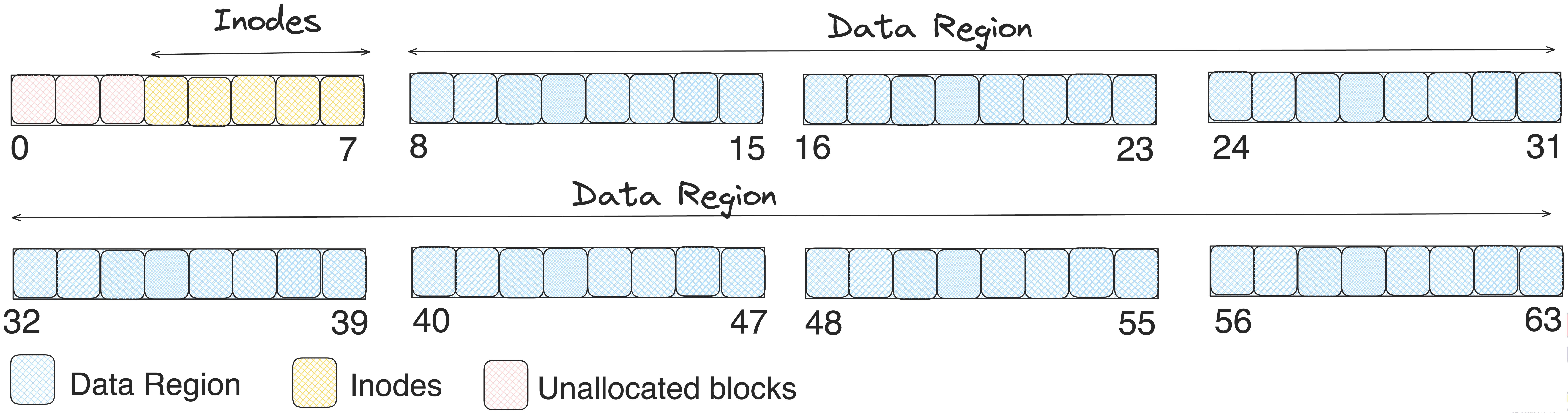
Some blocks needs to be reserved for storing data - **data region**



- 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!

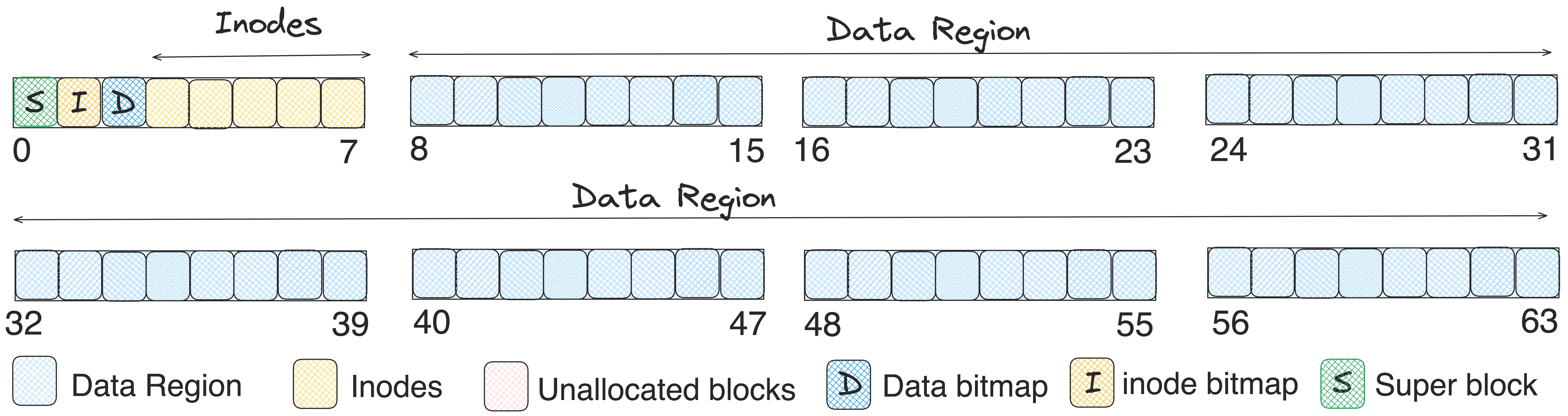
- Dedicate some space for inode table
  - This can hold an array of on-disk inodes
  - Consider each inode takes 256 bytes and 5 blocks are dedicated
  - Each block can hold 16 inodes => file system can hold 80 files



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

SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere