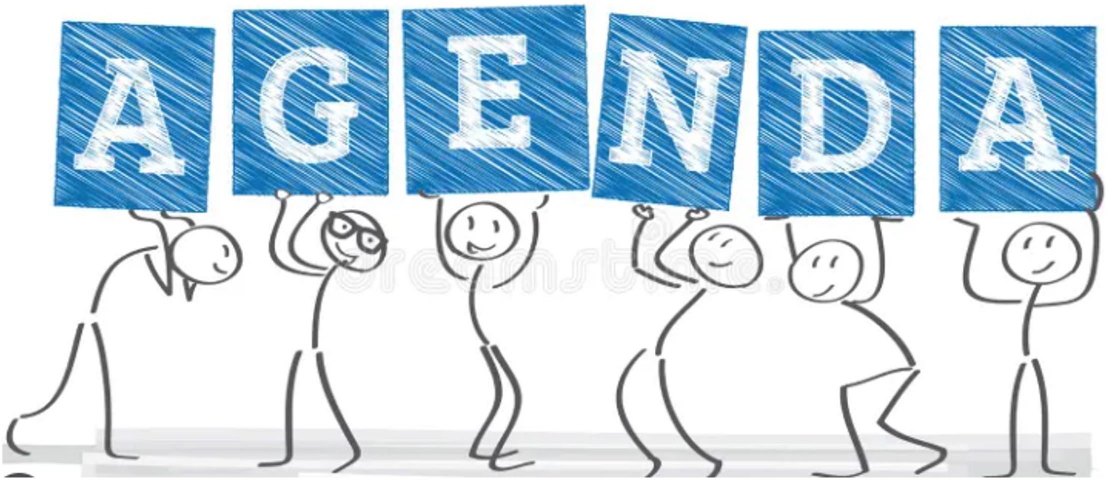
Operating Systems

Lecture 2.2

UNIX File System Architecture (Part-I)

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- Overview of UNIX File system Architecture
- What is Linux Virtual File System?
- How VFS work?
- VFS Data Structure and Operations
- Schematic View of Linux File System
- Understanding Superblock Inode, Dentry and File structure
- File System in Practice
- PPFDT and connection of Open Files
- IO redirection on the Shell
- The open-read-write-close paradigm

Linux Virtual File System
&
File System Architecture

What is a File System?

A file system is a software layer within the operating system that provides an abstraction for storing, organizing, and accessing data on storage devices. It allows users and programs to work with files and directories without needing to understand the underlying physical details of storage hardware, such as disk platters, heads, tracks, sectors, or cylinders.

- **Native Linux File Systems:**
  - **ext2** - Second Extended File System (legacy)
  - **ext3** - Third Extended File System (with journaling)
  - **ext4** - Fourth Extended File System (current standard)
  - **Btrfs** - B-tree File System (advanced features, snapshots)
  - **JFS** - IBM's Journaled File System
  - **ReiserFS** - Journaling file system (legacy)
  - **F2FS** - Flash-Friendly File System (for SSDs)

  Network File Systems:
  - **NFS** - Network File System
  - **AFS** - Andrew File System
  - **SMB/CIFS** - Server Message Block/Common Internet File System

- **Windows File Systems:**
  - **NTFS** - New Technology File System
  - **FAT12/FAT16/FAT32**
  - **exFAT** - Extended File Allocation Table

- **Virtual/Special File Systems:**
  - **procfs (/proc)** - Process information
  - **sysfs (/sys)** - System information
  - **devfs (/dev)** - Device files
  - **tmpfs** - Temporary file system in RAM
  - **ramfs** - RAM-based file system
  - **debugfs** - Kernel debugging
  - **configfs** - Kernel configuration

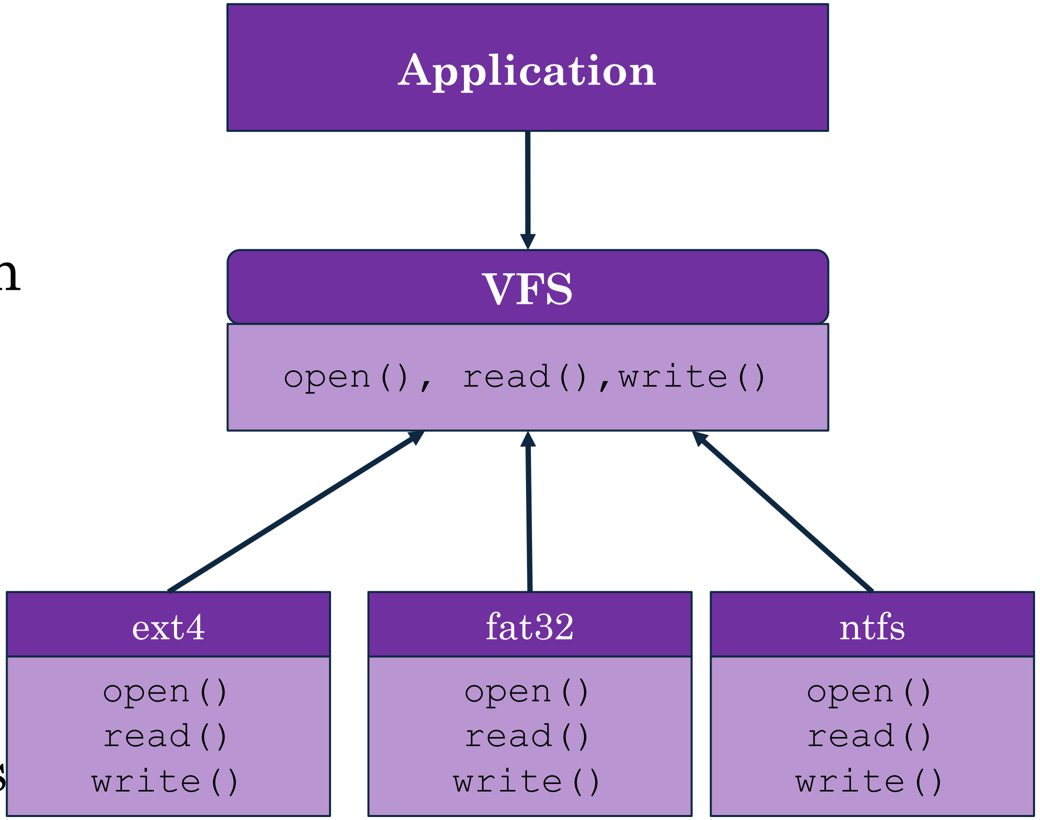
What is Linux Virtual File System

Virtual filesystem is the **magic abstraction** that makes the ***"everything is a file"*** philosophy of Linux **possible**

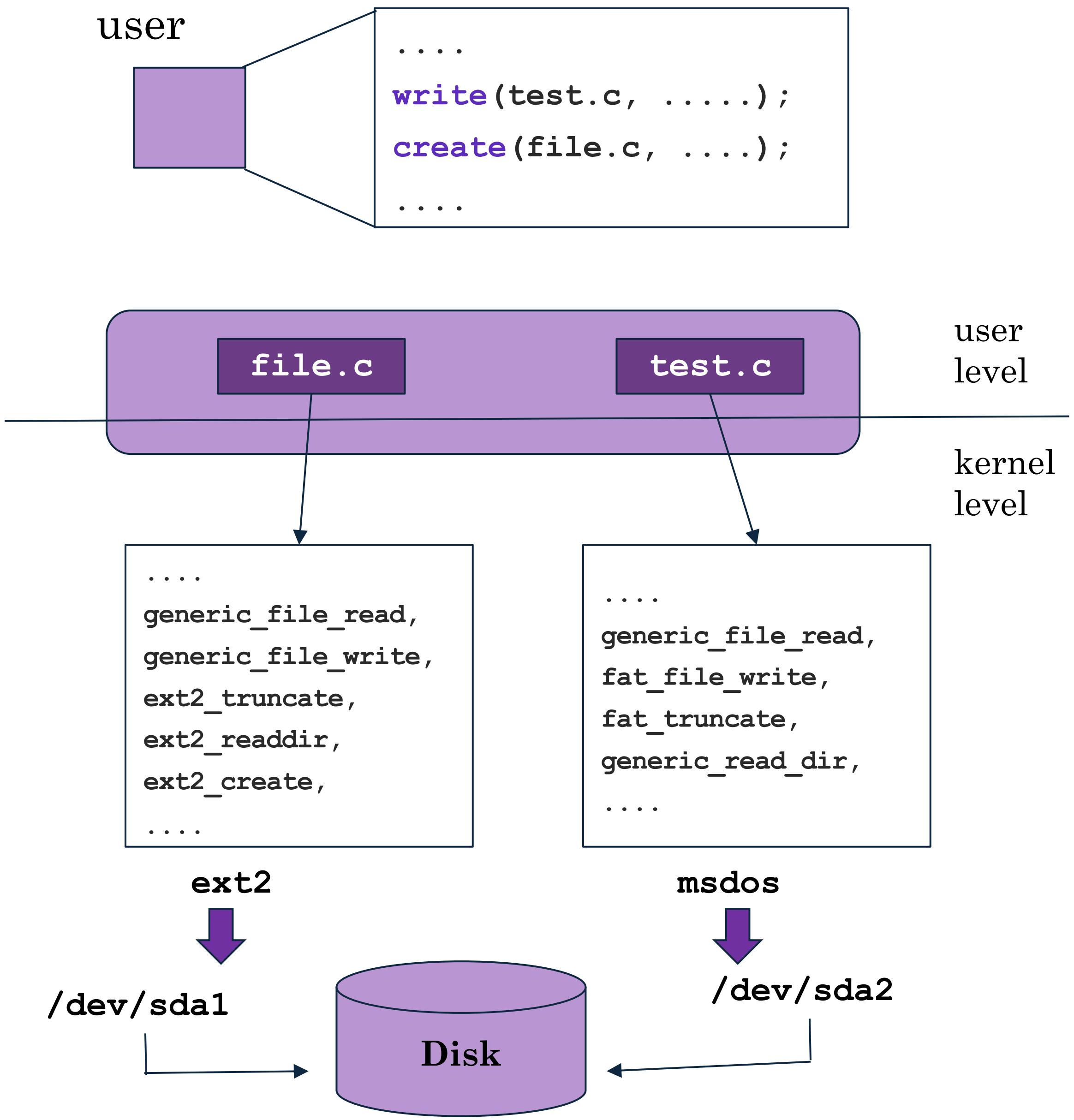
- In the early days of Unix-like systems, the kernel could only support one filesystem type at a time (e.g., `ext`).
- As networking, removable media, and virtual file systems emerged, there was a growing need for supporting multiple filesystem types. However, different filesystems (e.g., `ext`, `NFS`, `FAT`, `procfs`) had different internal implementations, making kernel code complex and unscalable.
- The Linux kernel introduced the Virtual File System (VFS) in the early 1990s as a unifying abstraction layer that allows the kernel to interact with all filesystems through a common interface, regardless of their on-disk format or purpose. VFS enables seamless data operations: e.g., copy data from ext4 via `read()` and write it to NFS via `write()`.
- This design made it easier to:
  - Add new filesystems (e.g., `ext4, Btrfs`)
  - Support remote/network filesystems (e.g., `NFS, SMB`)
  - Enable virtual filesystems like `/proc` and `/sys` with no physical storage

How VFS Works?

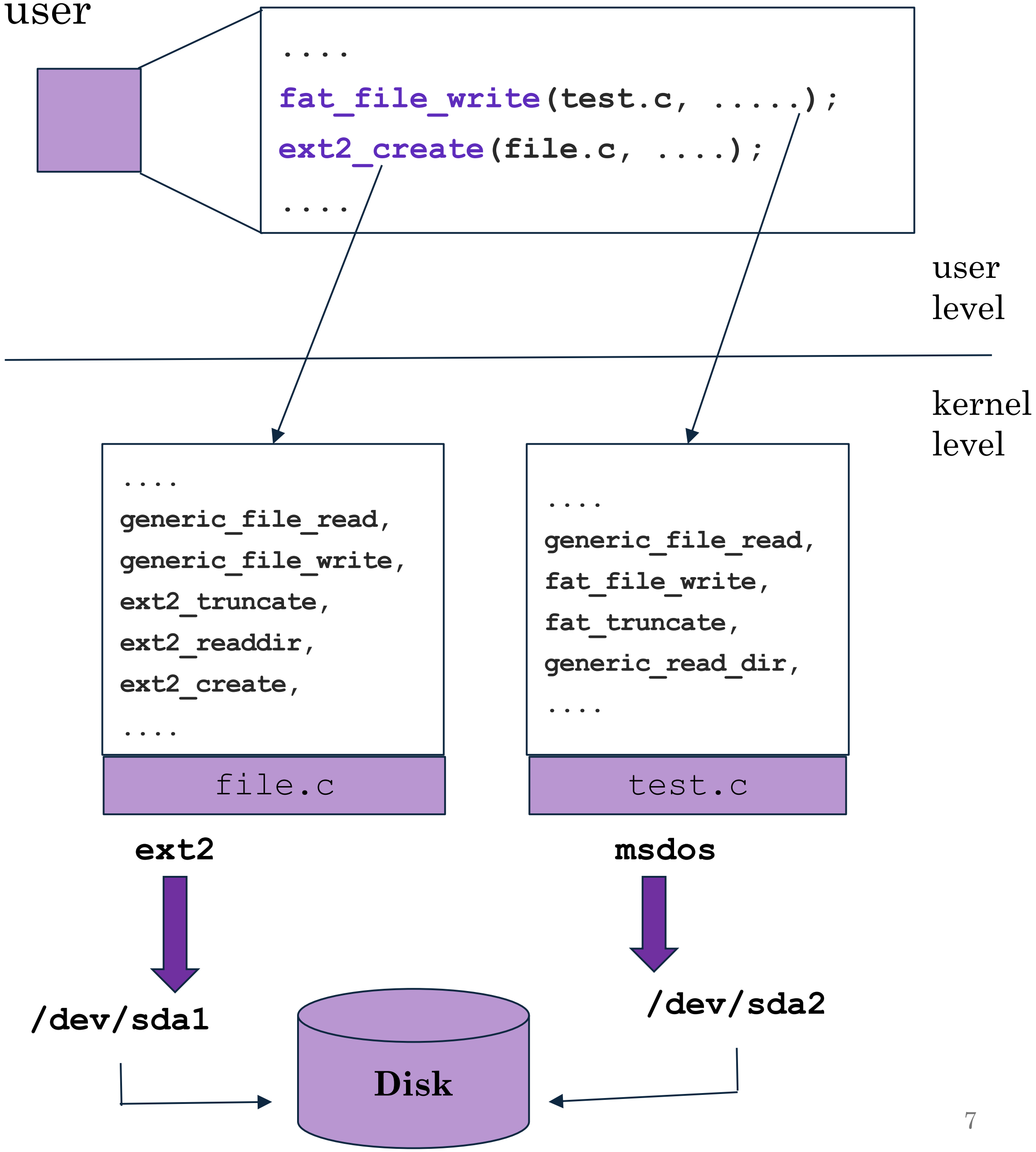
- VFS defines a common interface of file operations (e.g., `open, read, write, mkdir`) through `file_operations`.
- Each filesystem implements these operations through its own `file_operations` structure.
- When a system call is made (e.g., `open()`), VFS:
  - Resolves the file path using `dentries`
  - Finds the file metadata using the `inode`
  - Creates a file object to represent the open file
  - Returns a file descriptor to the user-space process
- VFS acts as a dispatcher, routing operations to the correct filesystem implementation.
- Uses internal caching (`dentry, inode, page`) to boost performance and reduce disk I/O.



With VFS



Without VFS



VFS Data Structures and Operations

VFS uses four main in-kernel object types:

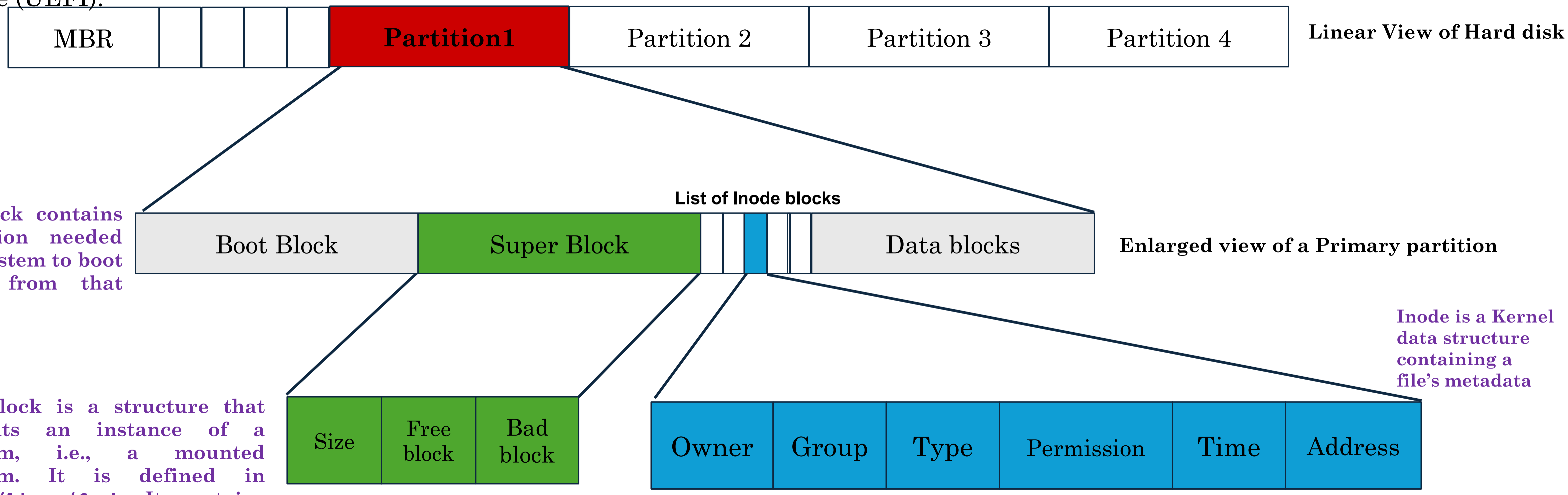
- **Superblock**
- **Inode**
- **Dentry**
- **File**

Elixir Cross Referencer is a web-based tool for browsing and cross-referencing large C codebases like the Linux kernel. It helps developers quickly find definitions, usages, and relationships between functions, structs, and macros. Developed by Bootlin, it offers a fast, searchable, and intuitive interface to explore complex code like vfs, drivers, and syscalls. Let us explort the fs/ directory that implements all supported file systems and the VFS (Virtual File System) layer.

https://elixir.bootlin.com/linux/v6.16/source

Schematic View of Linux Filesystem

A hard disk is divided into partitions, and each partition can have an independent file system. Master Boot Record (MBR) supports 2TiB disk size and four primary partitions. It resides in sector 0 (512 bytes) containing ~446 bytes of boot code, 64 bytes of partition table (four entries of 16 bytes each defining up to 4 primary partitions and boot signature (0x55AA) that mark its validity. Globally Unique Identifier Partition Table (GPT) has superseded MBR that support 9.4 ZiB disk size and 128 partitions. MBR uses BIOS while GPT uses Unified Extensible Firmware Interface (UEFI).



Boot Block contains information needed by the system to boot an OS from that partition

Super block is a structure that represents an instance of a filesystem, i.e., a mounted filesystem. It is defined in `include/linux/fs.h`. It contains metdata for filesystem, like total and free block/inode count, block size and other filesystem features

```
$ sudo dd if=/dev/sda bs=512 count=1 | hexdump -C    [To view the contents of boot block]
$ sudo tune2fs -l /dev/sda1 | less                   [To view and modify ext filesystems]
$ sudo dumpe2fs -h /dev/sda1                         [To view superblock details]
$ sudo file -s /dev/sda1                             [To view superblock details]
```

Superblock

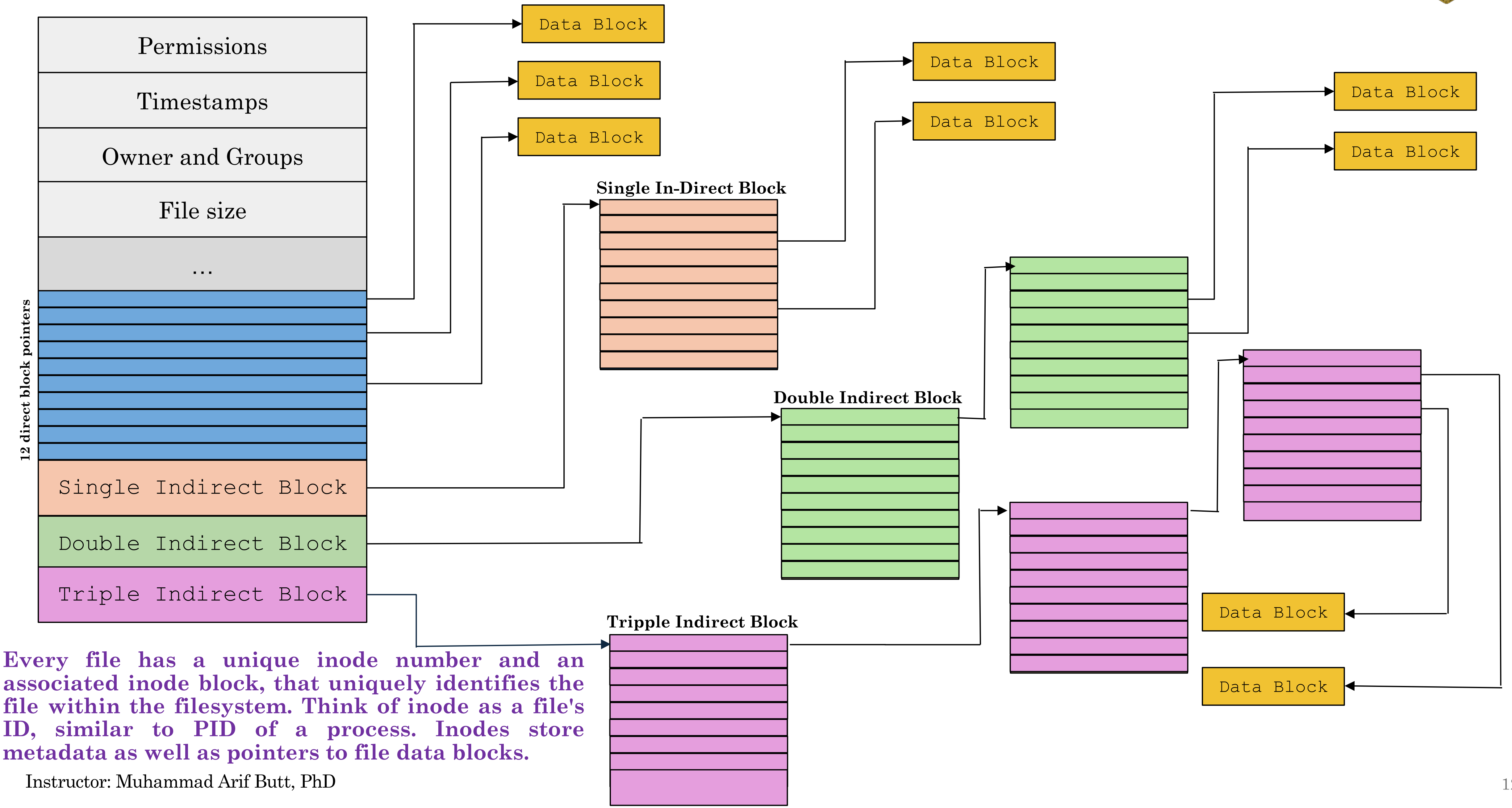
- Superblock is a structure that represents a mounted instance of a filesystem and *exists both on disk and in memory*.
- It contains essential filesystem metadata such as block size, total/free block counts, max file size, and a pointer to the root inode.
- In Linux, superblock is defined in `include/linux/fs.h` as `struct super_block` and includes a pointer to a table of superblock operations (`struct super_operations`)
- The `struct super_operations` define filesystem-wide methods such as inode allocation, syncing and unmounting e.g., `alloc_inode()`, `destroy_inode()`, `write_inode()` etc. These are invoked on the superblock and provide hooks for per-filesystem resource management.

Inode

- Inode is a is a structure that uniquely identifies a file or directory within a filesystem and *exists both on disk and in memory*.
- It contains metadata about the file, such as its size, permissions, ownership, timestamps, the type of file, and pointers to data blocks.
- The inode does not store the filename or directory path, which are managed by dentries.
- In Linux, it is defined in `include/linux/fs.h` as `struct inode`, and each inode also includes a pointer to table of inode operations (`struct inode_operations`), allowing filesystem-specific handling of operations like create, lookup, or unlink.
- The `struct inode_operations` defines operations related to file and directory metadata such as `create()`, `mkdir()`, `link()`, `unlink()`, `lookup()` etc. These are invoked on inodes and control how the filesystem handles structural changes.

struct inode: https://elixir.bootlin.com/linux/v6.0/source/include/linux/fs.h#L593

Structure of an Inode block in Linux ext Filesystem



Every file has a unique inode number and an associated inode block, that uniquely identifies the file within the filesystem. Think of inode as a file's ID, similar to PID of a process. Inodes store metadata as well as pointers to file data blocks.

Dentry

- **Dentry** (short for directory entry) is a structure used to map file names to their corresponding inodes and *exists only in memory*.
- It plays a critical role in pathname resolution by breaking full paths (e.g., `/home/user/file.txt`) into individual components, each represented by a `dentry`.
- The dentry structure caches name-to-inode mappings to speed up repeated lookups and reduce disk I/O. It does not contain file metadata itself, but rather acts as the glue between a file name and its inode.
- In Linux, it is defined in `include/linux/dcache.h` as `struct dentry`, and it may include a pointer to a table of `dentry_operations` table for filesystem-specific behaviors like name comparison and validation.
- The `struct dentry_operations`, defines operations related to path lookup and name validation such as `d_hash()`, `d_compare()`, `d_delete()` etc, used during pathname resolution and dentry cache management.

Inode#	Filename
54	.
6	..
47	**f1.txt**
49	f2.txt
34	dir1
35	dir2

struct dentry_operations: https://elixir.bootlin.com/linux/v6.0/source/include/linux/dcache.h#L127

File

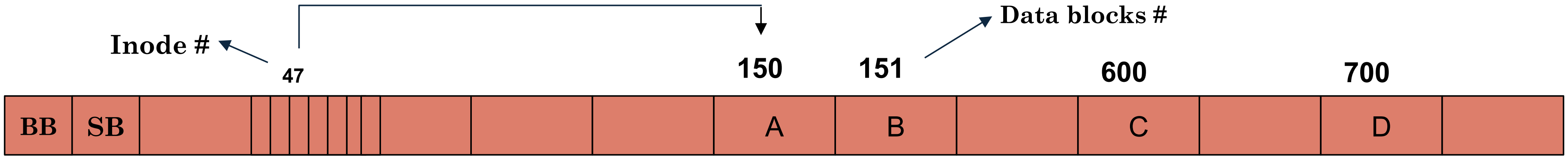
- **File** is a structure that represents an open file instance in the kernel and *exists only in memory* for the duration of the file access.
- It contains runtime information such as the current file offset, access mode (read/write), and flags set during opening (e.g., O_APPEND).
- Each process has a file descriptor table that points to these file structures, allowing multiple processes or threads to share open file instances.
- In Linux, it is defined in `include/linux/fs.h` as `struct file`, and it also contains a pointer to a table of `struct file_operations`, enabling filesystem-specific implementations of operations like read(), write(), and llseek().
- The `struct file_operations` defines file-specific operations performed on open files like `read()`, `write()`, `lseek()`, `ioctl()` etc. These operations are tied to the file structure and govern runtime file I/O behavior.

`struct file`: https://elixir.bootlin.com/linux/v6.0/source/include/linux/fs.h#L940

File System in Practice

File system in action: File creation

```
$ echo "Learning is fun with Arif Butt" 1> /home/arif/f1.txt
```



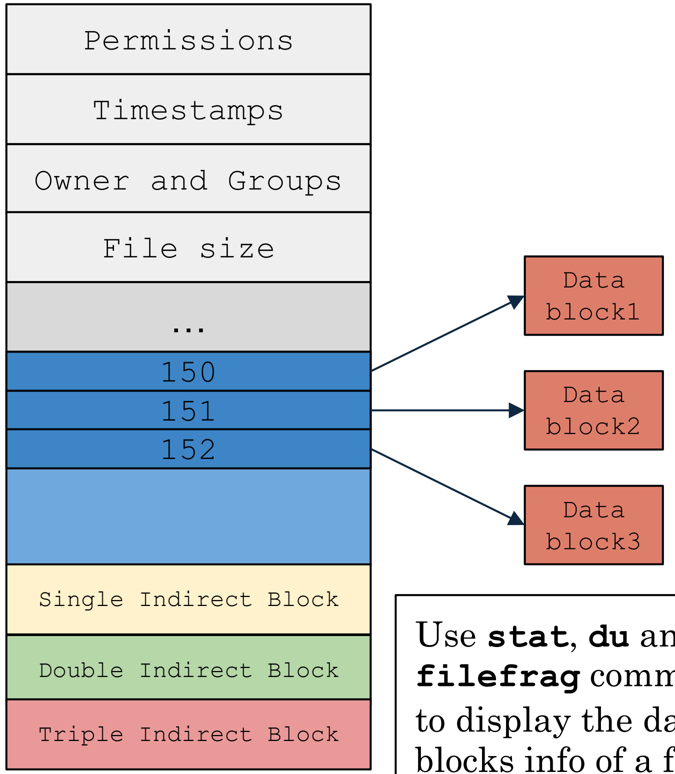
When a user creates a new file, the kernel performs the following steps:

1. **Allocates a free inode** to store the file's metadata (permissions, ownership, timestamps, etc.).
2. **Allocates free data blocks** to store the actual contents of the file.
3. **Updates the inode** to record the pointers of the allocated data blocks.
4. **Adds an entry to the directory** to link the filename to its corresponding inode number.

Directory entries for `/home/arif`

Inode#	Filename
54	.
6	..
47	**`f1.txt`**
49	`f2.txt`
34	`dir1`
35	`dir2`

`Inode block #47`

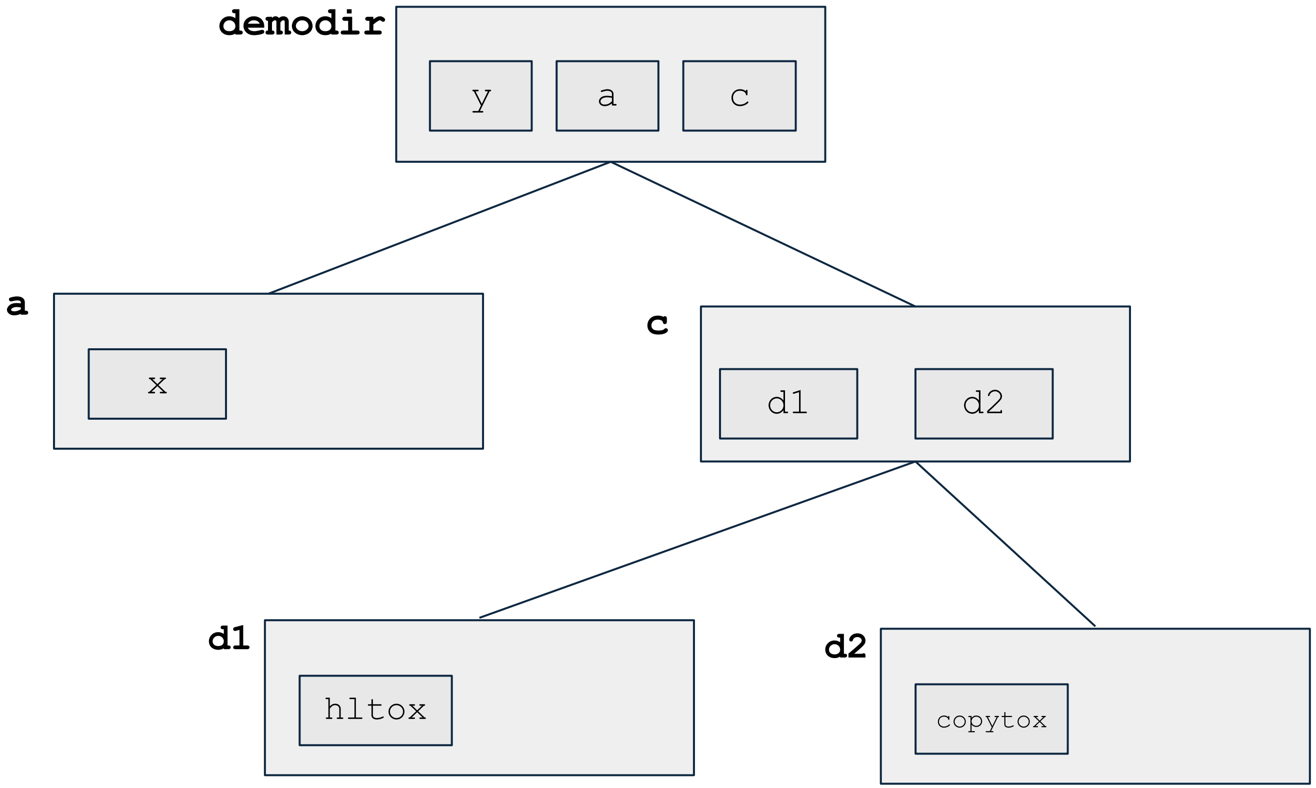


Use **`stat`**, **`du`** and **`filefrag`** commands to display the data blocks info of a file

File system in action: Understanding directories

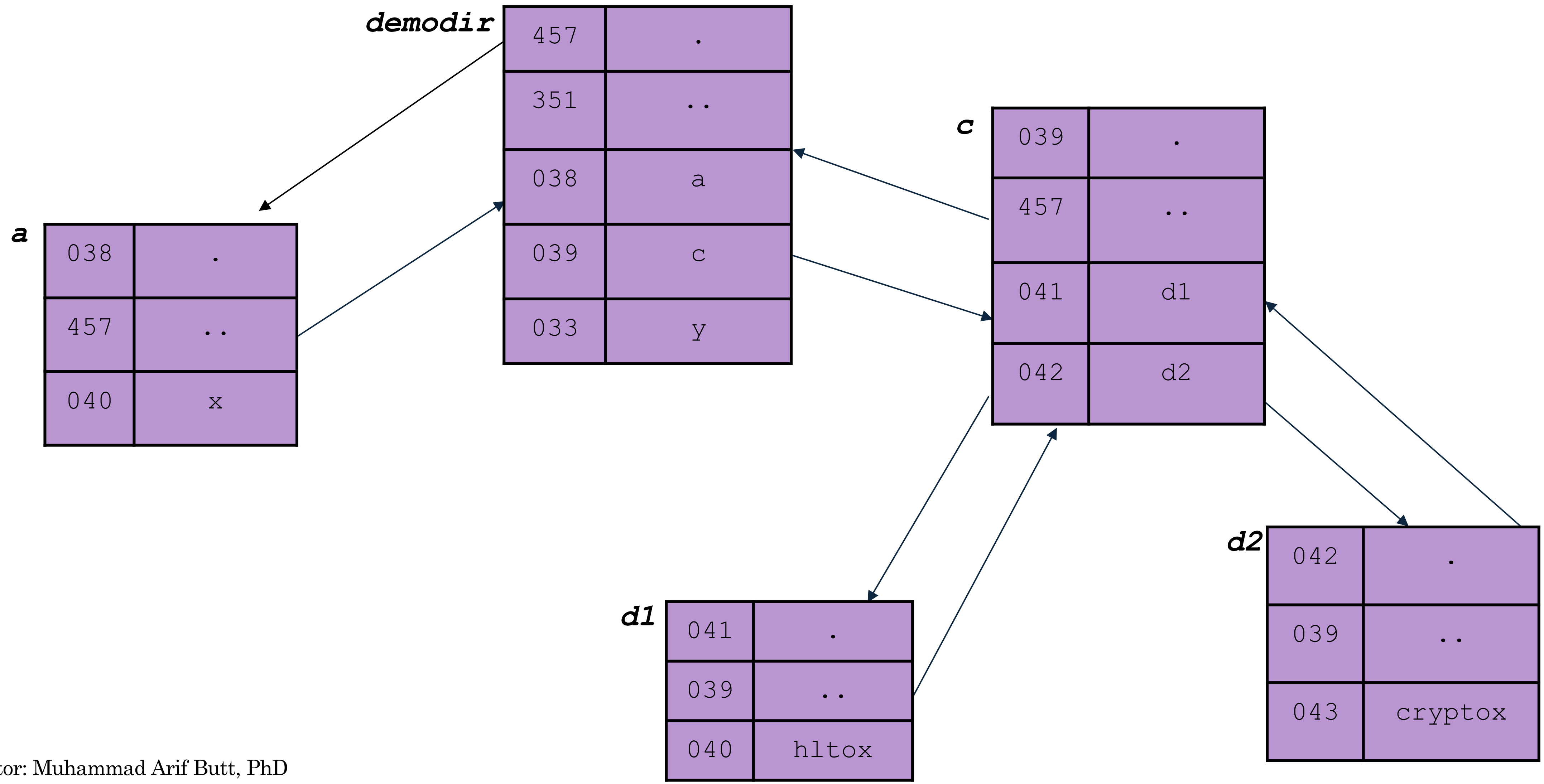
```
$ ls -iaR

demodir/:
2621457 .   2629351 ..   2627038 a   2627039 c   2627033 y

demodir/a:
2627038 .   2621457 ..   2627040 x

demodir/c:
2627039 .   2621457 ..   2627041 d1   2627042 d2

demodir/c/d1:
2627041 .   2627039 ..   2627040 hltox

demodir/c/d2:
2627042 .   2627039 ..   2627043 copytox
```

User View of Directory Structure



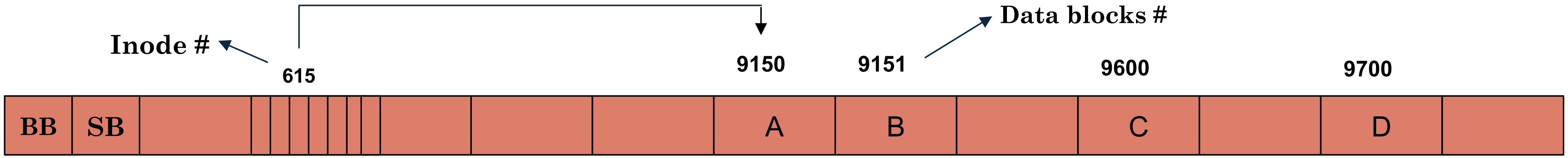
File system in action: Understanding directories

System View of Directory Structure



File system in action: Accessing a file

```
$ cat /home/arif/f1.txt
```



When a user tries to access a file (**`f1.txt`**), the kernel performs the following steps:

1. Searches the directory structure to find the given filename.
2. Retrieves the associated inode, e.g., inode 615.
3. Performs a permission check by comparing the process's user ID with the file's owner, group, and others.
4. Accesses the file's data blocks. First 12 data block addresses are stored directly in the inode. Additional data blocks are accessed via Single indirect block, Double indirect block, and Triple indirect block

Inode # 2

/

2	.
2	..
561	**home/**
	bin
34	etc
35	var

Inode # 561

home/

561	.
2	..
533	**arif/**
534	rauf

Inode # 533

arif/

533	.
561	..
615	**f1.txt**
619	file2.txt

Inode # 615

f1.txt

Learning is fun with Arif Butt

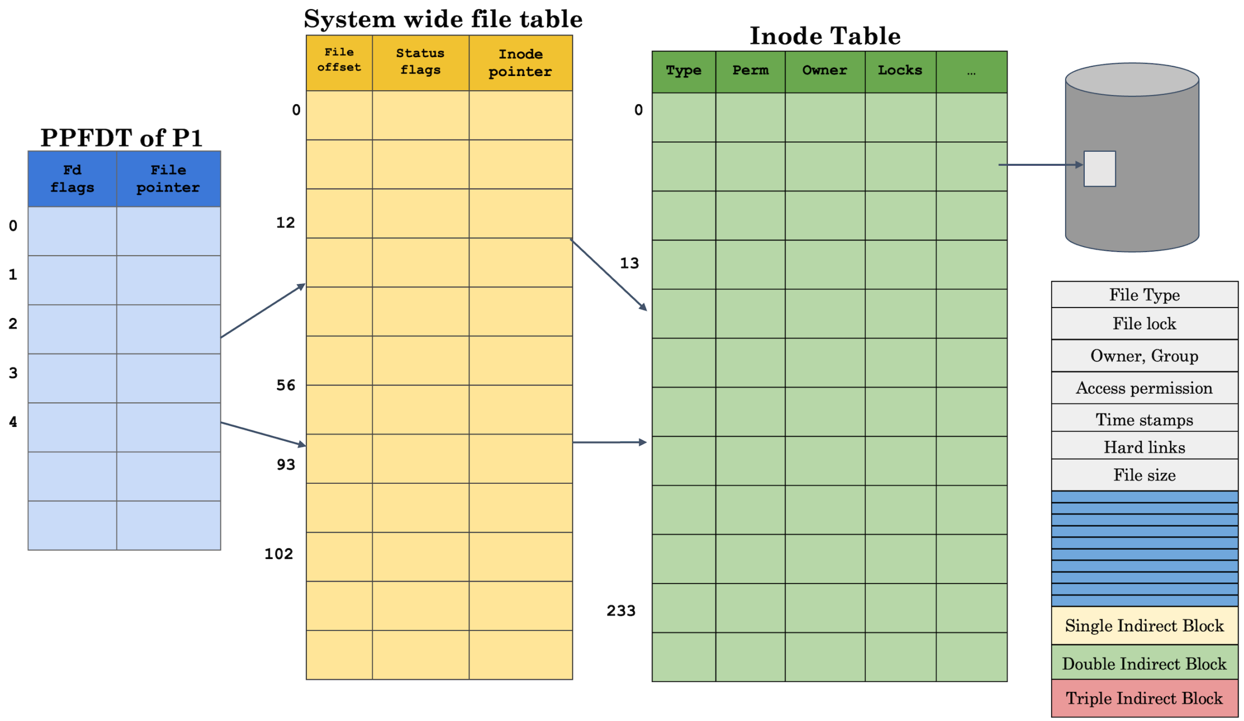
<EOF>

Connection between fd and Open Files

Relation between fd and opened files - PPFDT

Per Process File Descriptor Table:

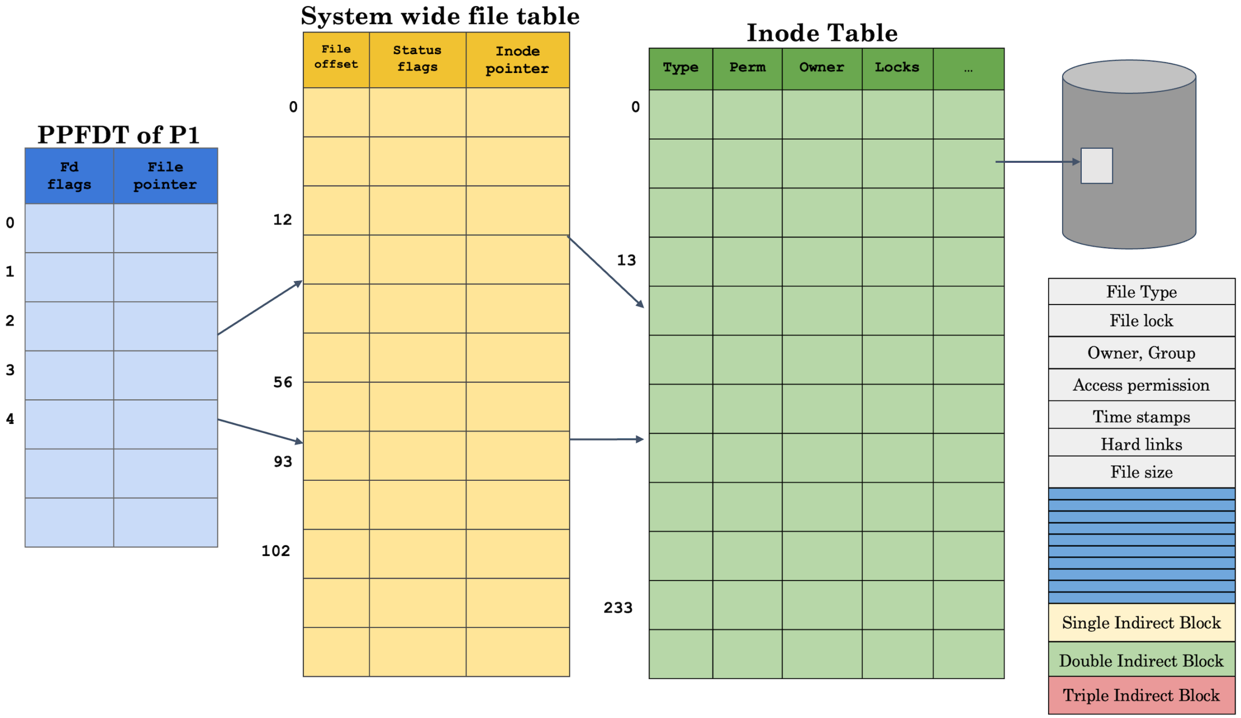
- The kernel maintains a dedicated file descriptor table for each process to track all files that the process has opened.
- By default three files are opened at descriptor 0, 1, and 2 called `stdin`, `stdout` and `stderr`.
- Total number of entries in this table is kernel variable `OPEN_MAX`, that specifies the max number of files that a process can open at a time. Traditional limit is 1024 file descriptors per process (configurable via `ulimit` command)
- Each entry stores a set of flags controlling file descriptor operations, and a reference pointer to the global system file table
- In order to view all files opened by a specific process, you can use the `lsof -p <pid>` command.
- In order to identify which processes have opened a particular file, you can use the `fuser <filename>` command.



Relation between fd and opened files - SWFT

System Wide File Table:

- The kernel maintains a global file table that tracks all files currently opened by any process in the system.
- Each entry in SWFT include following information:
  - Current file offset used by read/write operations to track the position within the file.
  - Status flags that were specified when opening the file, categorized into three types:
    - ✓ Access mode flags: O_RDONLY, O_WRONLY, O_RDWR
    - ✓ Open time flags: O_TRUNC, O_CREAT, O_EXCL
    - ✓ Operating mode flags: O_APPEND, O_SYNC, O_NONBLOCK

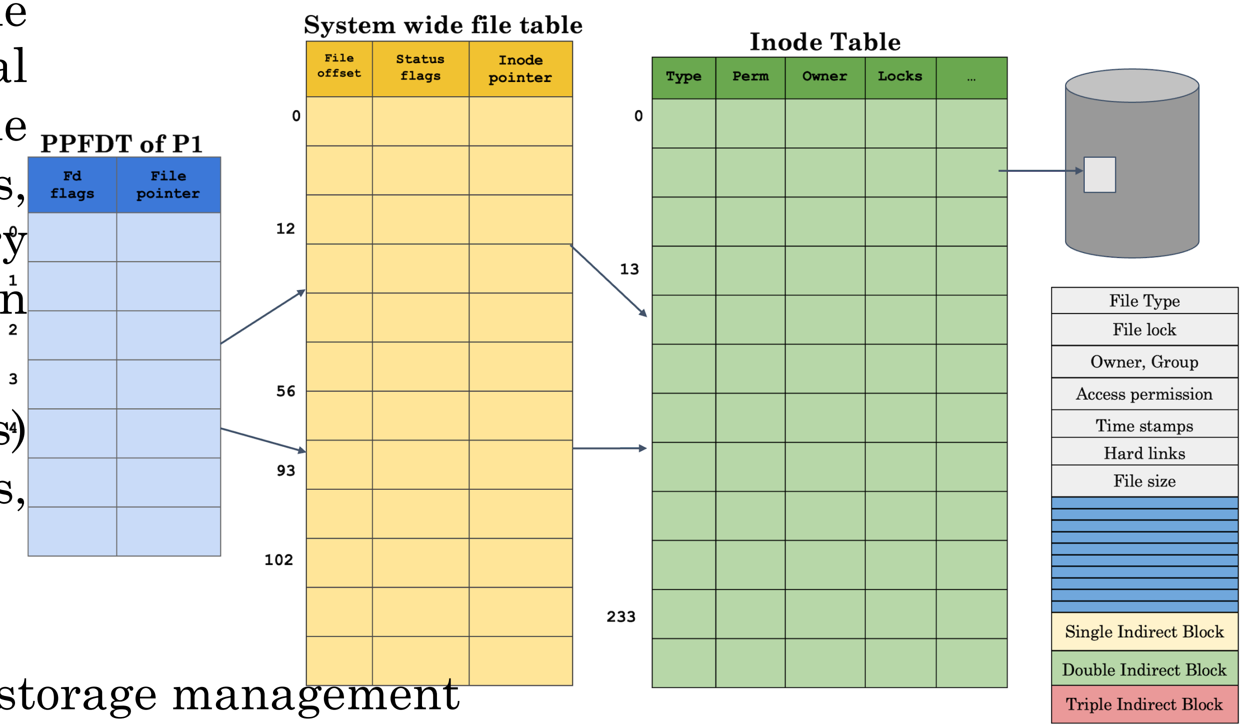


- Maximum number of entries in SWFT represents the maximum number of files that can be opened system-wide at any instant of time.
- Each entry maintains a reference pointer to the corresponding inode object in the inode table.
- When multiple processes open the same file, separate entries are created in the SWFT, but they may point to the same inode in the inode table.

Relation between fd and opened files – inode Table

Inode Table and Inode Block:

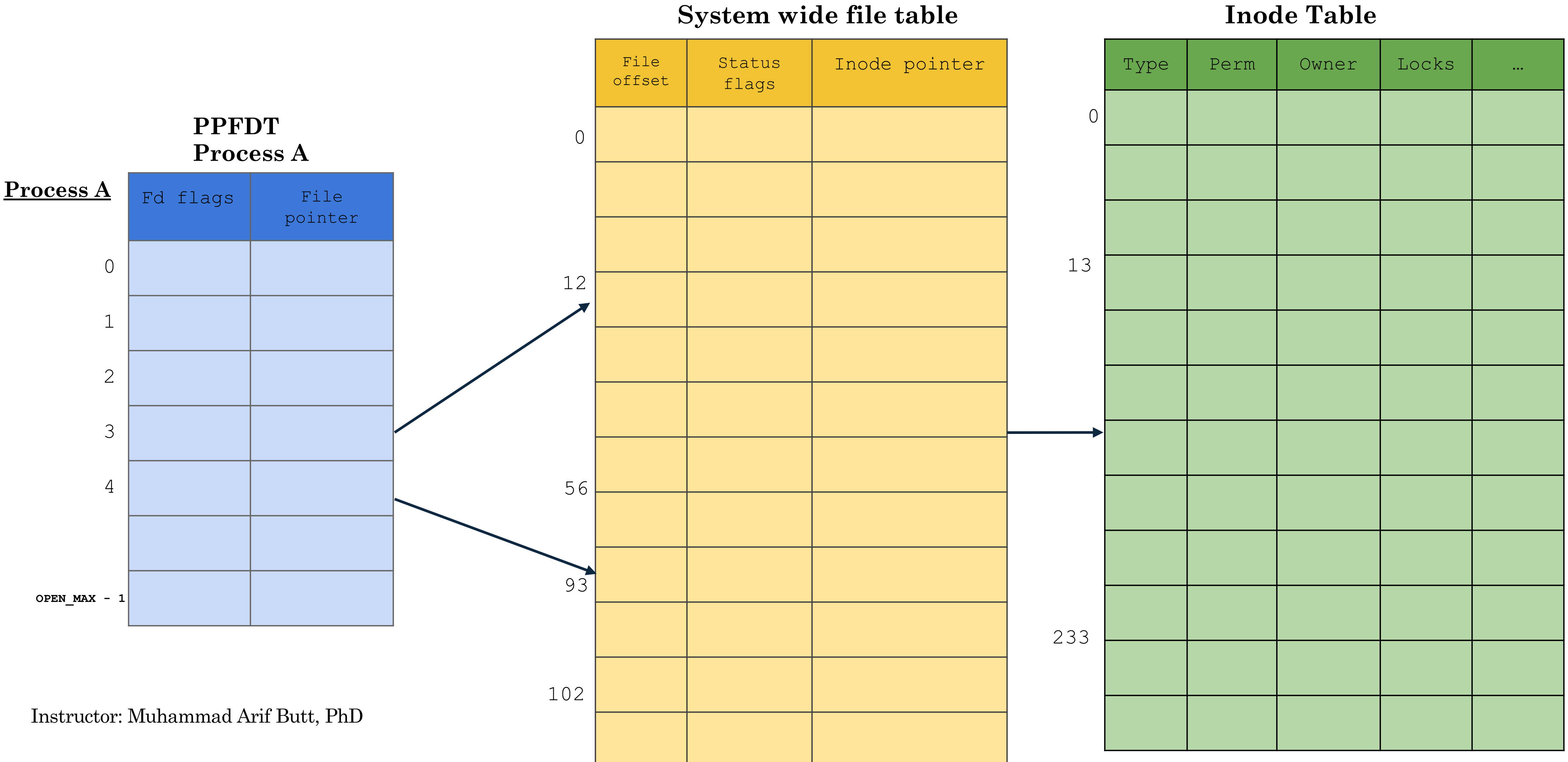
Each file system maintains a table of inodes for all the files residing in that filesystem, serving as the central repository of file metadata. Each inode acts as the unique identifier for files on disk, much like a PID for processes, and holds essential information about the file. Each entry in the inode table stores comprehensive file information including:



- **File type:** Seven different types (-, d, l, p, c, b, s) representing regular files, directories, links, pipes, character devices, block devices, and sockets.
- **File lock:** Information on file locks applied to the file.
- **File size:** Stored in both bytes and blocks for efficient storage management
- **Ownership and Access permissions:** Read, write and execute permissions for owner, group and others
- **Time stamps:** Modification time (ls -l), Access time (ls -lu), Status change time (ls -lc)
- **Link Management:** Number of hard links pointing to this inode
- **Data Block Pointers:** Contains a total of 15 pointers. Twelve direct pointers that directly point to data blocks, one single indirect pointer, one double indirect pointer, and one triple indirect pointer.

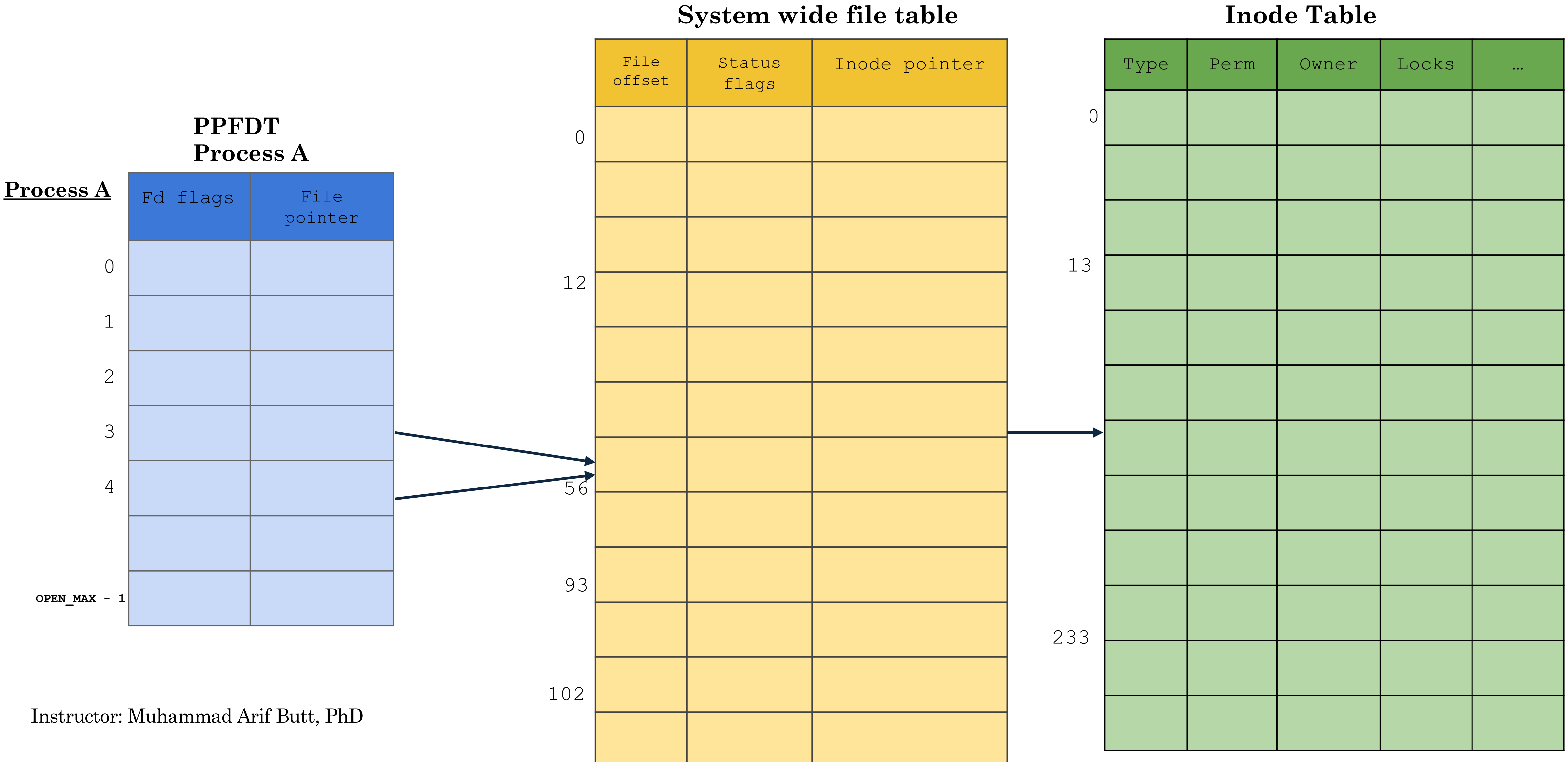
File Descriptor to File Contents

If a process call `open()` twice on a file, there will be two entries in PPFDT, two entries in SWFT and one entry in inode table.



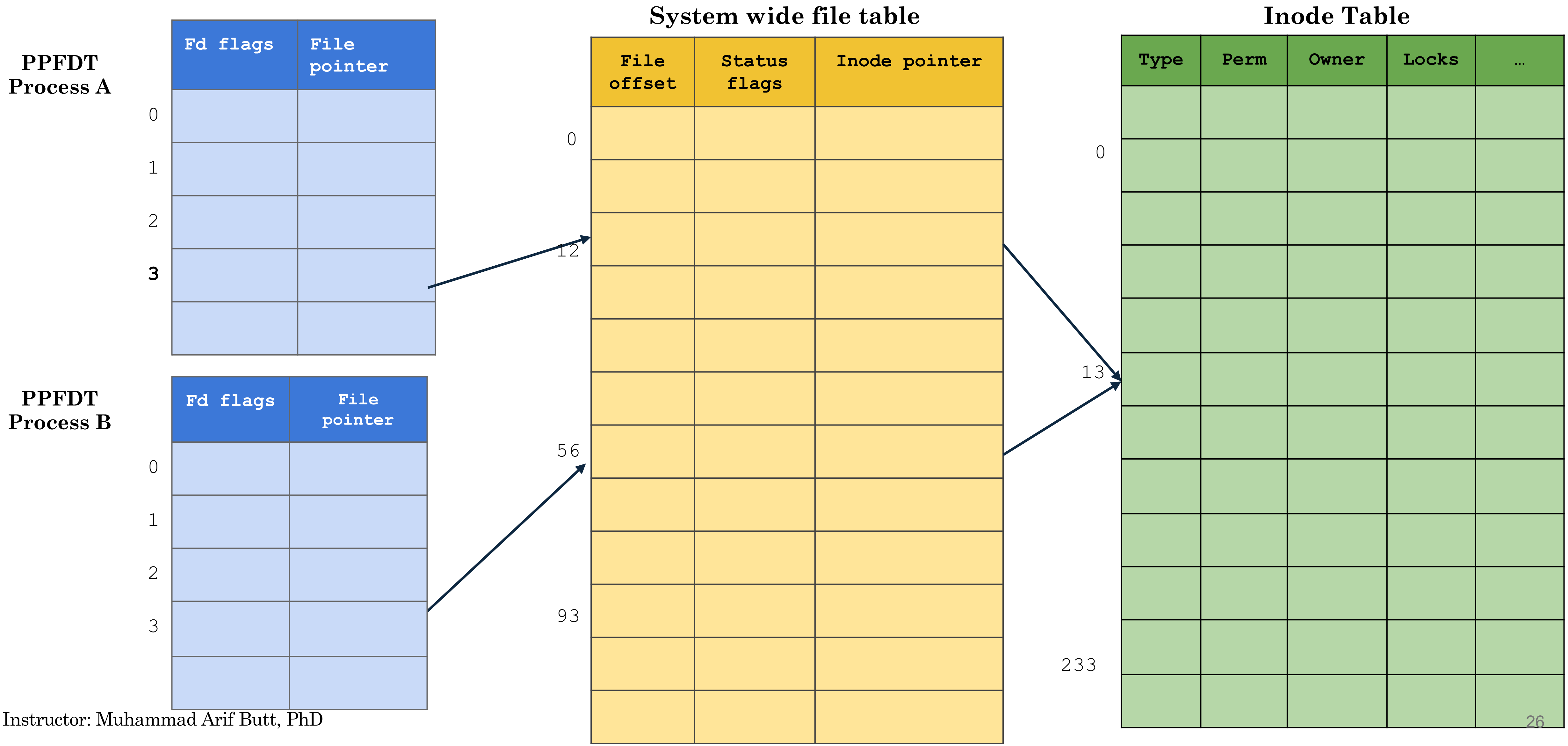
File Descriptor to File Contents

If a process call `open()` on a file and then dup(), there will be two entries in PPFDT, one entry in SWFT and one entry in inode table.



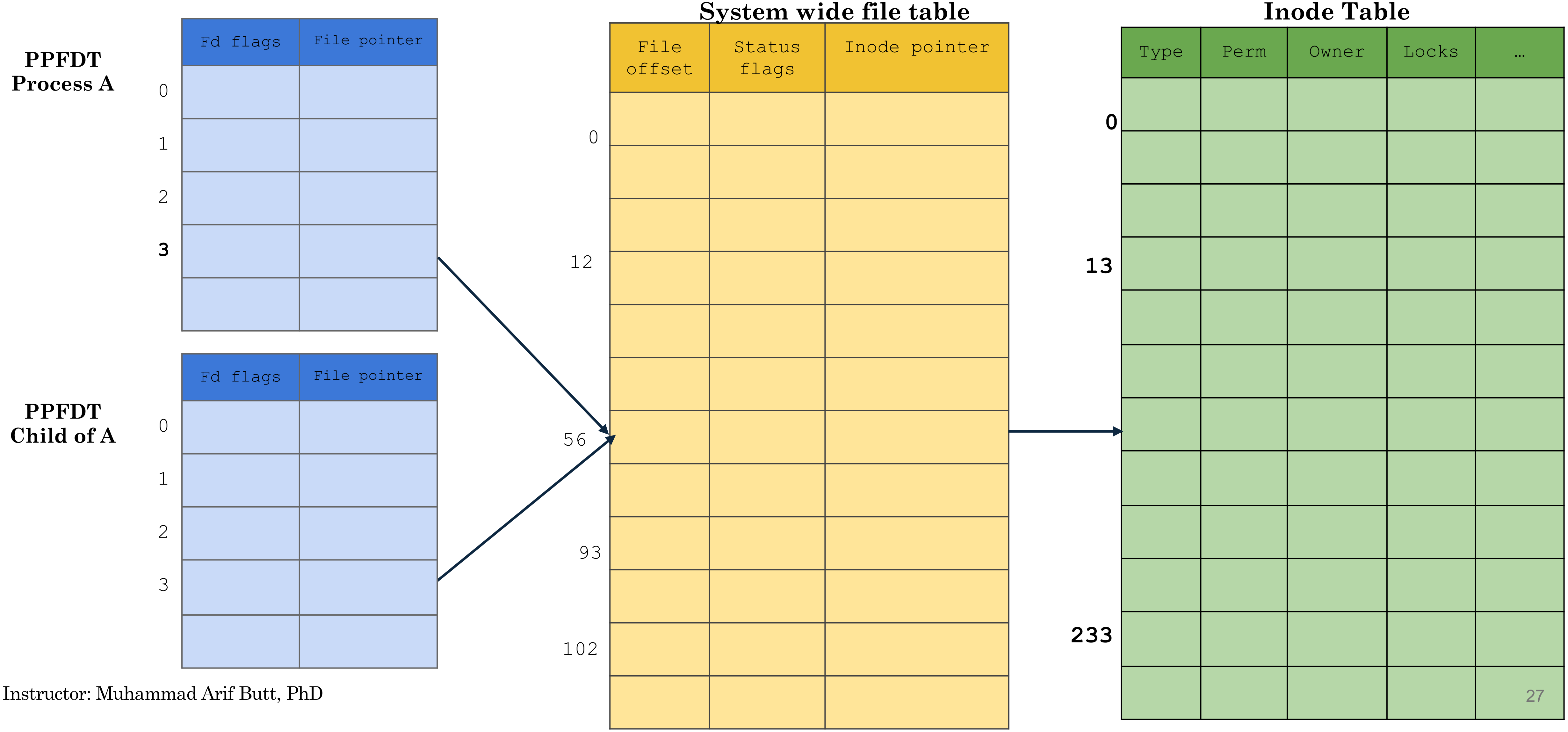
File Descriptor to File Contents

If two different processes opens the same file by calling `open()`, there will be two different entries in SWFT



File Descriptor to File Contents

If a process opens a file with `open()` and then calls `fork()` the parent and child processes will share the same SWFT entry, and both refer to the same inode table entry.



I/O Redirection on the Shell

Standard file descriptors

- All system calls for performing I/O refer to open files using a file descriptor, a (usually small) nonnegative integer. File descriptors are used to refer to all types of open files, including regular files, directories, terminals, devices, pipes, and sockets. Symbolic links, however, are not normally "opened" for reading/writing in the same sense. They're followed (resolved) by the kernel to another file, unless you explicitly open them with O_NOFOLLOW flag . In most cases, you don't get a file descriptor that directly refers to the symlink's contents. Each process has its own set of file descriptors.

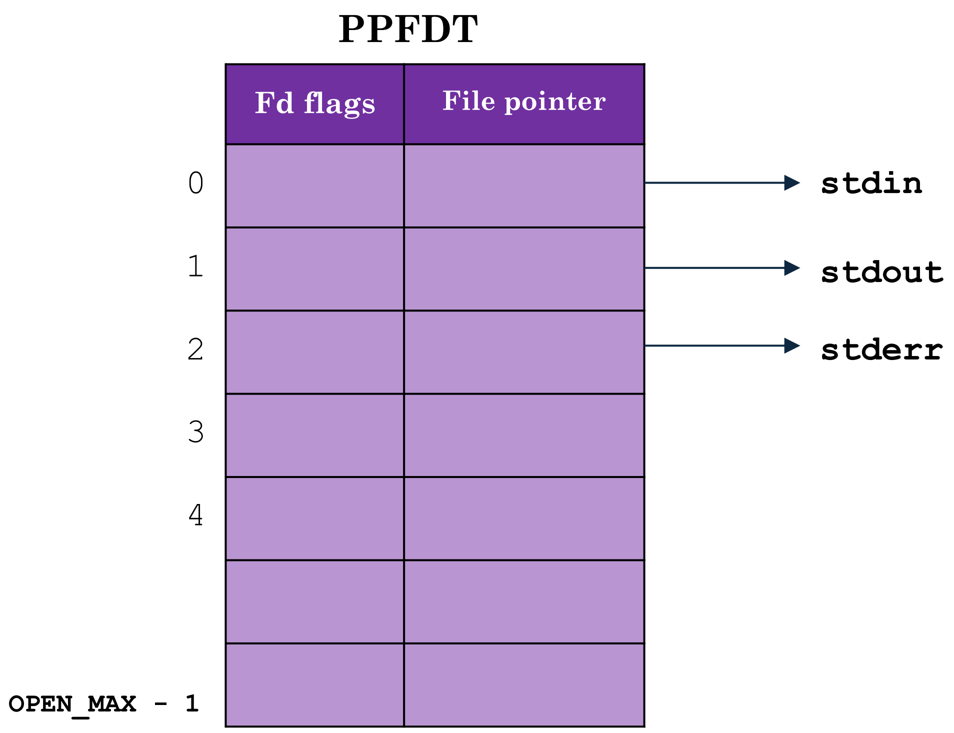
- By convention, most programs expect to be able to use the three standard file descriptors listed below. These three descriptors are opened on the program's behalf by the shell, before the program is started. Or, more precisely, the program inherits copies of the shell's file descriptors, and the shell normally operates with three file descriptors always open as mentioned in the table below:

File descriptors	Purpose	POSIX Name	stdio stream
0	Standard Input	STDIN_FILENO	stdin
1	Standard Output	STDOUT_FILENO	stdout
2	Standard Error	STDERR_FILENO	stderr

stdin and stdout for Shell Commands

```
$ cat

This is Great
This is Great
<Ctrl + D>
```

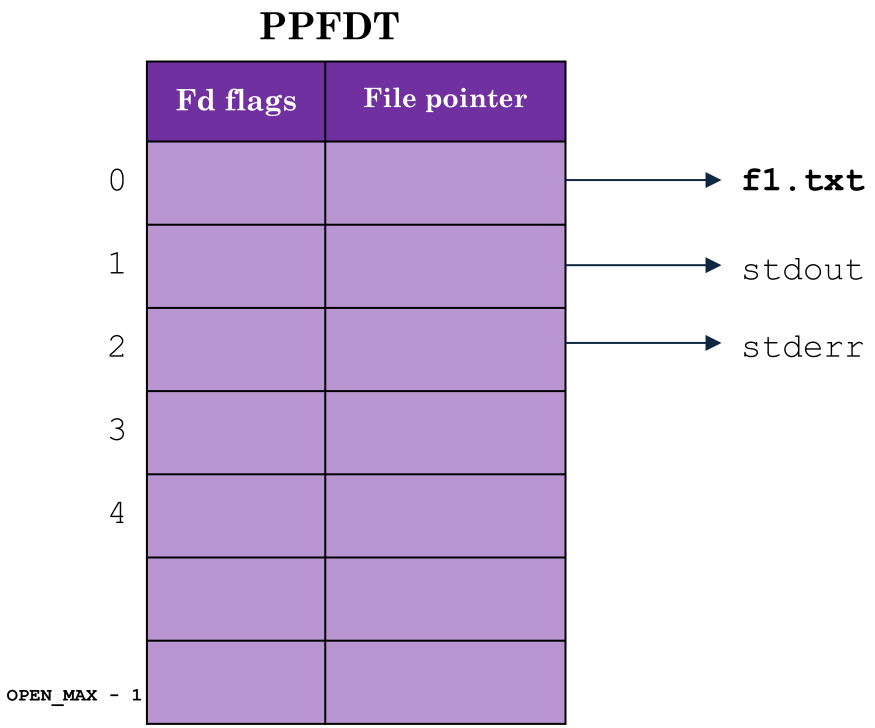
```
$ sort rauf arif kamal

rauf
aif
kamal
<Ctrl + D>
```



Redirecting Input (0<)

- By default, `cat` and `sort` commands takes their input form the standard input, i.e. keyboard. We can detach the keyboard from `stdin` and attach some file to it.
- After input of a process is redirected, it will read from this file and not from the keyboard

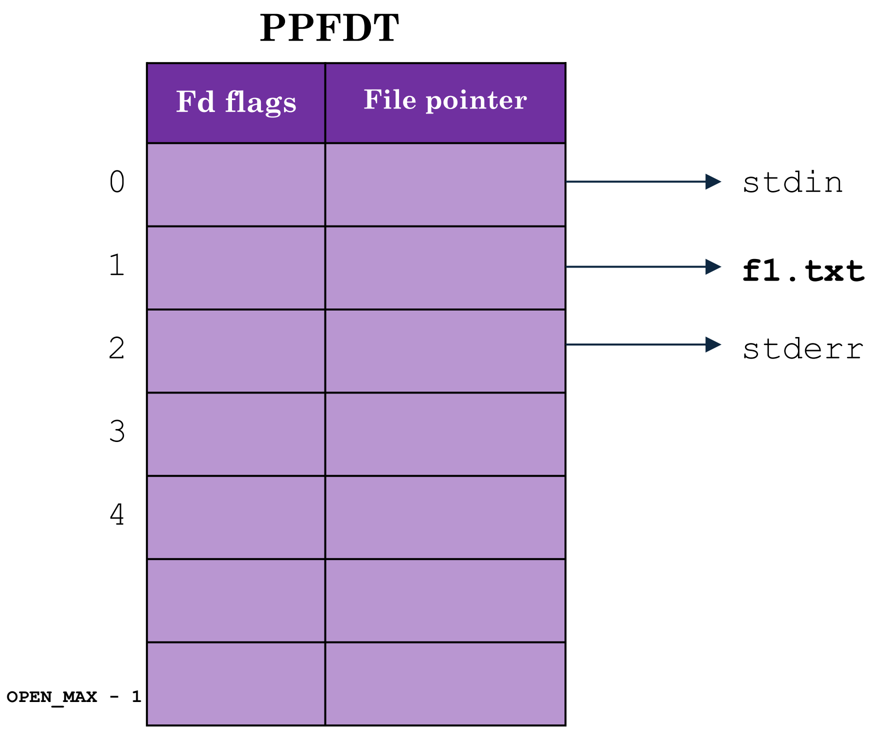
```
$ cat 0< f1.txt

$ sort 0< f1.txt
```



Redirecting Output (1>)

Similarly, by default `cat` and `sort` commands sends their outputs to user terminal. We can detach the display screen from `stdout` and attach a file to it; i.e. `cat` command will now write its output to this file and not to the display screen.

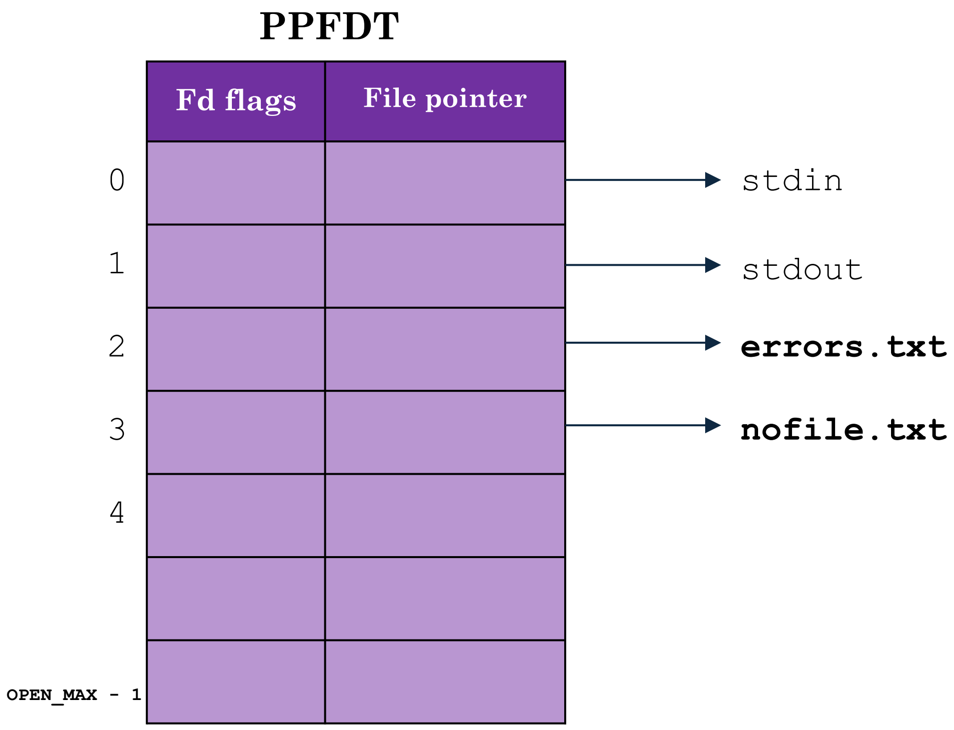
```
$ cat 1> f1.txt
```



Redirecting Error (2>)

Similarly, by default `cat` and `sort` commands sends their outputs to user terminal. We can detach the `stderr` and attach a file to it; i.e. `cat` command will now write its errors to this file and not to the display screen

```
$ cat nofile.txt 2> errors.txt
```

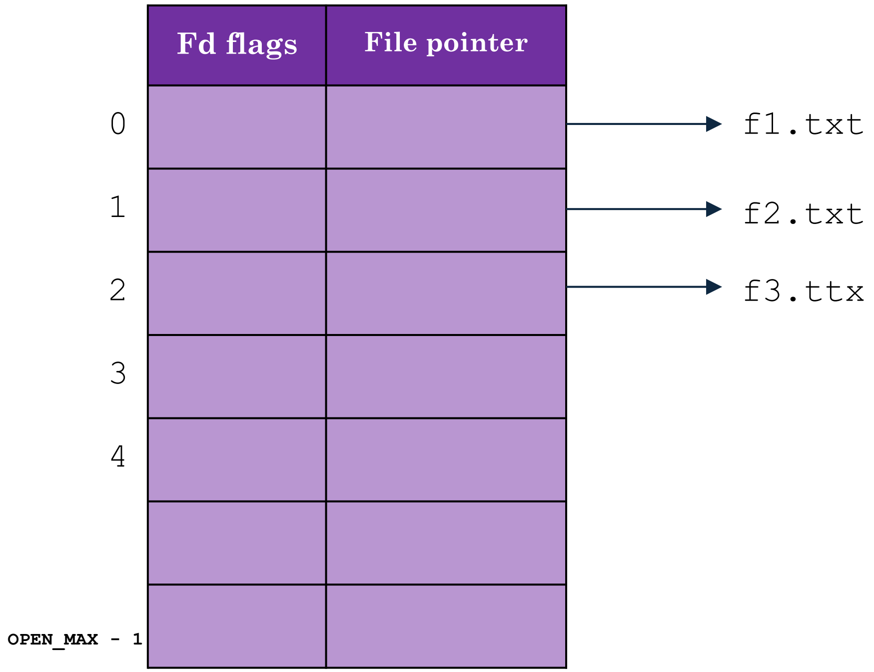


Redirecting Input, Output and Error

We can also redirect the input, output and error in a single shell command as shown below:

```
$ cat 0< f1.txt 1> f2.txt 2> f3.txt
```

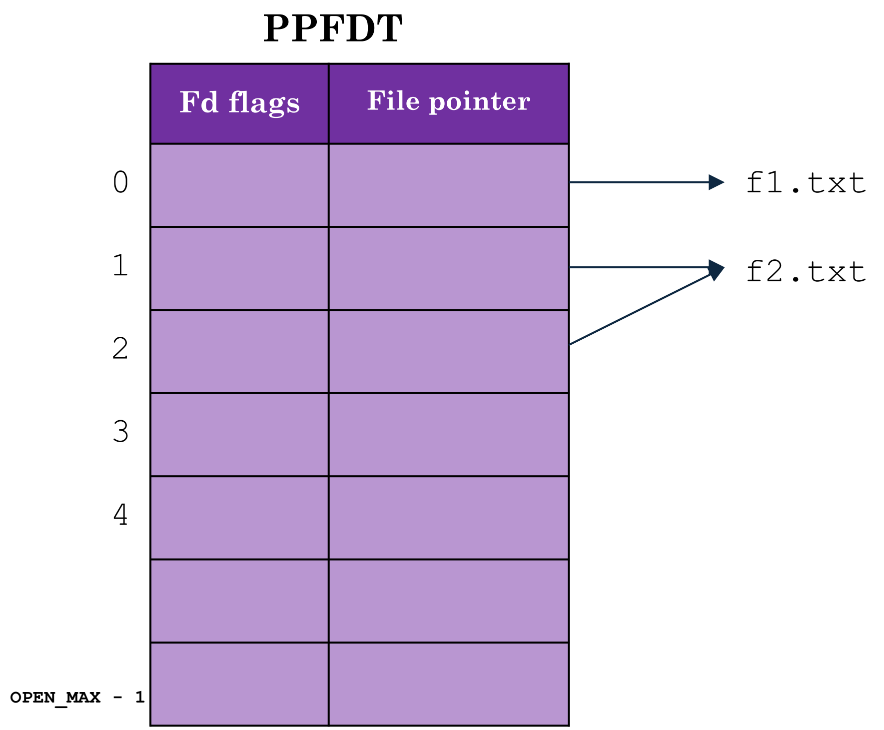
PPFDT



Duplicating a File Descriptor

We can use the syntax `2>&1` that informs the shell (make 2 a copy of 1), i.e., the standard error is redirected to the same place to which `stdout` is pointing at that moment.

```
$ cat 0< f1.txt 1> f2.txt 2>&1
```



Redirection happens from Left to Right

Consider the following commands, and understand their execution, if the input file don't exist.

Command	`f2.txt` created?	Error goes to?	Why?
`cat 0< f1.txt 1> f2.txt 2>&1`	No	Terminal	`0< f1.txt` fails first, so command & other redirections are never processed
`cat 2>&1 1> f2.txt 0< f1.txt`	Yes	Terminal	`2>&1` processed first (points to terminal), then `1> f2.txt` (creates file), then `0< f1.txt` fails
`cat 1> f2.txt 2>&1 0< f1.txt`	Yes	`f2.txt`	`1>` and `2>&1` redirect both output and error to `f2.txt`, then `0< f1.txt` fails, so error goes to `f2.txt`

$100 QUESTION

How many command line arguments are passed to the `cat` program and why?

Redirection happens from Left to Right

Consider the following commands, where `f1.txt` is passed as a command line argument and understand their execution:

- The shell does not open the file itself, rather the `cat` process does.
- So, all redirections are processed first, regardless of whether `f1.txt` exist or not.
- Then `cat` is executed and it fails inside the process if the file does not exist.

Command	**f2.txt created?**	**Error goes to?**	**Why?**
`cat f1.txt 1> f2.txt 2>&1`	Yes	`f2.txt`	Redirections processed before cat runs; both stdout and stderr go to `f2.txt`
`cat f1.txt 2>&1 1> f2.txt`	Yes	Terminal	`2>&1` points stderr to terminal, then `1> f2.txt` changes `stdout` to f2.txt. Error goes to terminal
`cat f1.txt 1> f2.txt 2>&1`	Yes	`f2.txt`	Both stdout and stderr points to f2.txt, and error goes to `f2.txt`

Universal I/O model

`open()` System Call

```
int open(char *pathname, int flags);
int open(char *pathname, int flags, mode_t mode);
```

- The file to be opened is identified by the pathname argument. If pathname is a symbolic link, it is dereferenced
- On success, `open()` returns a file descriptor that is used to refer to the file in subsequent system calls
- On error, `open()` returns –1 and `errno` is set accordingly
- The file status flags argument is a bit mask that:
  a. Must include one of the three file access modes `(O_RDONLY, O_WRONLY, O_RDWR)`
  b. Zero or more file open time flags, `(O_CREAT, O_TRUNC, O_EXCL)`
  c. Zero or more file operating mode flags `(O_APPEND, O_SYNC, O_NONBLOCK)`

Flags argument of open ( ) System call

Flags	Description
O_RDONLY	Open file in read only mode
O_WRONLY	Open file in write only mode
O_RDWR	Open file in read write mode
O_CREAT	If file does not already exist , it makes a new file. If we specify O_CREAT, then we must supply a mode argument in the open() call; otherwise, the permissions of the new file will be set to some random value from the stack
O_APPEND	Writes are always appended to the end of the file
O_TRUNC	If the file already exists and is a regular file, then truncate it to zero length, destroying any existing data
O_EXCL	This flag is used in conjunction with O_CREAT to indicate that if the file already exists, it should not be opened; instead, open() should fail, with errno set to EEXIST
O_CLOEXEC	Enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor. By default, the file descriptor will remain open across an execve(). Normally used in multithreaded programs to avoid the race conditions

Mode argument of open() System call

- When `open()` is used to create a new file, the mode bit-mask argument specifies the permissions to be placed on the file. If the `open()` call doesn't specify `O_CREAT`, mode can be omitted.
- Mode argument can be specified as a number (typically in octal) or, preferably, by ORing (|) together zero or more of the bitmask constants. These constants are:

S_IRWXU	0700	S_IRWXG	0070	S_IRWXO	0007
S_IRUSR	0400	S_IRGRP	0040	S_IROTH	0004
S_IWUSR	0200	S_IWGRP	0020	S_IWOTH	0002
S_IXUSR	0100	S_IXGRP	0010	S_IXOTH	0001

- Permissions actually placed on a new file depend not just on the mode argument, but also on the process `umask` and can be computed as:

```
mode & ~umask
```

- This mode only applies to future accesses of the newly created file.

File Descriptor returned by `open()`

- SUSv3 specifies that if `open()` succeeds, it is guaranteed to use the lowest-numbered unused file descriptor for the process. We can use this feature to ensure that a file is opened using a particular file descriptor.

- For example, the following sequence ensures that a file is opened using standard input (file descriptor 0)

```
close(0);
fd = open(pathname, O_RDONLY);
```

- Since file descriptor 0 is unused, `open()` is guaranteed to open the file using that descriptor.

read() System Call

```
ssize_t read(int fd, void *buf, size_t count);
```

- Attempts to read up to `count` number of bytes from the file descriptor `fd` into the buffer starting at memory address `buf`.
- If `count` is 0 then `read()` return 0. If count is greater than `SSIZE_MAX` then the result is unspecified. `SSIZE_MAX` is the largest positive value that can fit in `ssize_t` (e.g., typically 9223372036854775807 on 64-bit systems).
- On success, returns number of bytes read, which can be less than count if `EOF` is encountered. Before a successful return the current file offset is incremented by the number of bytes actually read.
- In case of regular file having more than `count` bytes, it is guaranteed that `read` will read `count` bytes and then will return. However, in case of FIFO or socket this is not guaranteed.
- On failure, returns -1 and set `errno`
- A return of zero indicates end-of-file.

pread() System Call

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

- This function read `count` number of bytes from the file descriptor `fd` at offset `offset` into the buffer starting at memory address `buf`.
- On success; Number of bytes read is returned and **current file offset is not advanced to new location**.
- On failure; Return -1 and `errno` is set to indicate the error.
- A return value of 0 means nothing is read.

`write()` System Call

```
ssize_t write(int fd,void *buf,size_t count);
```

- Attempts to write up to `count` number of bytes to the file referenced by file descriptor `fd` from the buffer starting at memory address `buf`. The data is written starting with the current location of current file offset.
- On success; Number of bytes written is returned which may be less than count. Current file offset is advanced to new location.
- In case of regular file, the call guarantees writing `count` bytes, if the disk is not full or the file size has not exceeded the maximum file size supported by system. However, in case of FIFO or socket this is not guaranteed.
- On failure; Return -1 and `errno` is set appropriately.
- Return 0 indicates nothing is written.

`pwrite()` System Call

```
ssize_t pwrite(int fd, void *buf,size_t count, off_t offset);
```

- This function write `count` number of bytes from memory address pointed to by `buf` to the file referenced by file descriptor `fd` at offset `offset`.
- On success; Number of bytes written is returned and **current file offset is not advanced to new location**.
- On failure; Return -1 and `errno` is set to indicate the error.
- A return value of 0 indicates nothing is written.

`creat()` System Call

```
int creat(char *pathname, mode_t mode);
```

- In early UNIX implementations, `open()` had only two arguments and could not be used to create a new file. Instead, the `creat()` system call was used to create and open a new file.
- The `creat()` system call creates and opens a new file with the given pathname, or if the file already exists, opens the file and truncates it to zero length.
- On success, `creat()` returns a file descriptor that can be used in subsequent system calls. Calling `creat()` is equivalent to the following `open()` call:

  ```
  fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
  ```

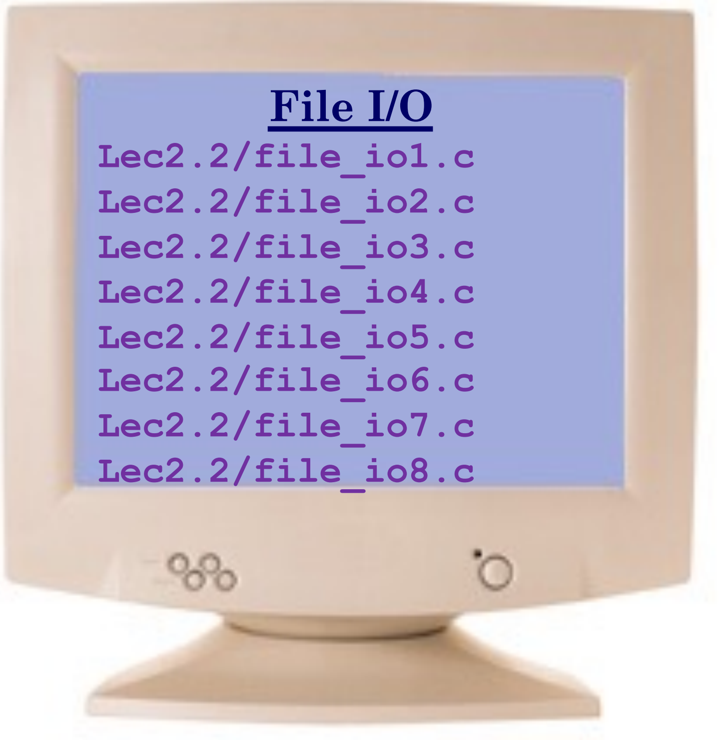
- Because the `open()` flags argument provides greater control over how the file is opened (e.g., we can specify `O_RDWR` instead of `O_WRONLY`), `creat()` is now obsolete.
- So, using `creat()`, a file is opened only for writing. If we were creating a temporary file that we wanted to write and then read back, we had to call `creat()`, `close()` and then `open()`

`close()` System Call

```
int close(int fd);
```

- Close a file descriptor `fd` so that it is no longer referenced in the PPFDT and may be reused to a later call of `open()`, or `dup()`.
- Closing a file also releases any record locks that a process may have on file.
- When a process terminates, all open files are automatically closed by kernel.
- On Success; Return 0
- On failure; Return -1 and `errno` is set appropriately.

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Watch OS video on I/O Redirection on the Shell:
https://www.youtube.com/watch?v=ik6TvPquVk8&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=9

- Watch OS video on File system architecture:
https://www.youtube.com/watch?v=58WJZbcNj2E&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=21

- Watch SP video on File system architecture:
https://www.youtube.com/watch?v=x_bu6De71KY&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=12

- Watch SP video on File related system calls:
https://www.youtube.com/watch?v=DZQkyoXgkMs&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=13

Coming to office hours does NOT mean that you are academically weak!