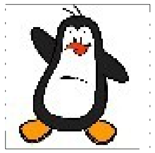


Video Lecture # 12
UNIX File System Architecture

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Agenda

- Recap
  - Disk geometry
  - Disk partitioning
  - File system mounting
- File System Architecture
- Data structures involved in FSA
- Connection to an opened file
- The `open-read-write-close` Paradigm

OS with Linux Lec#16
Hard Disk Geometry

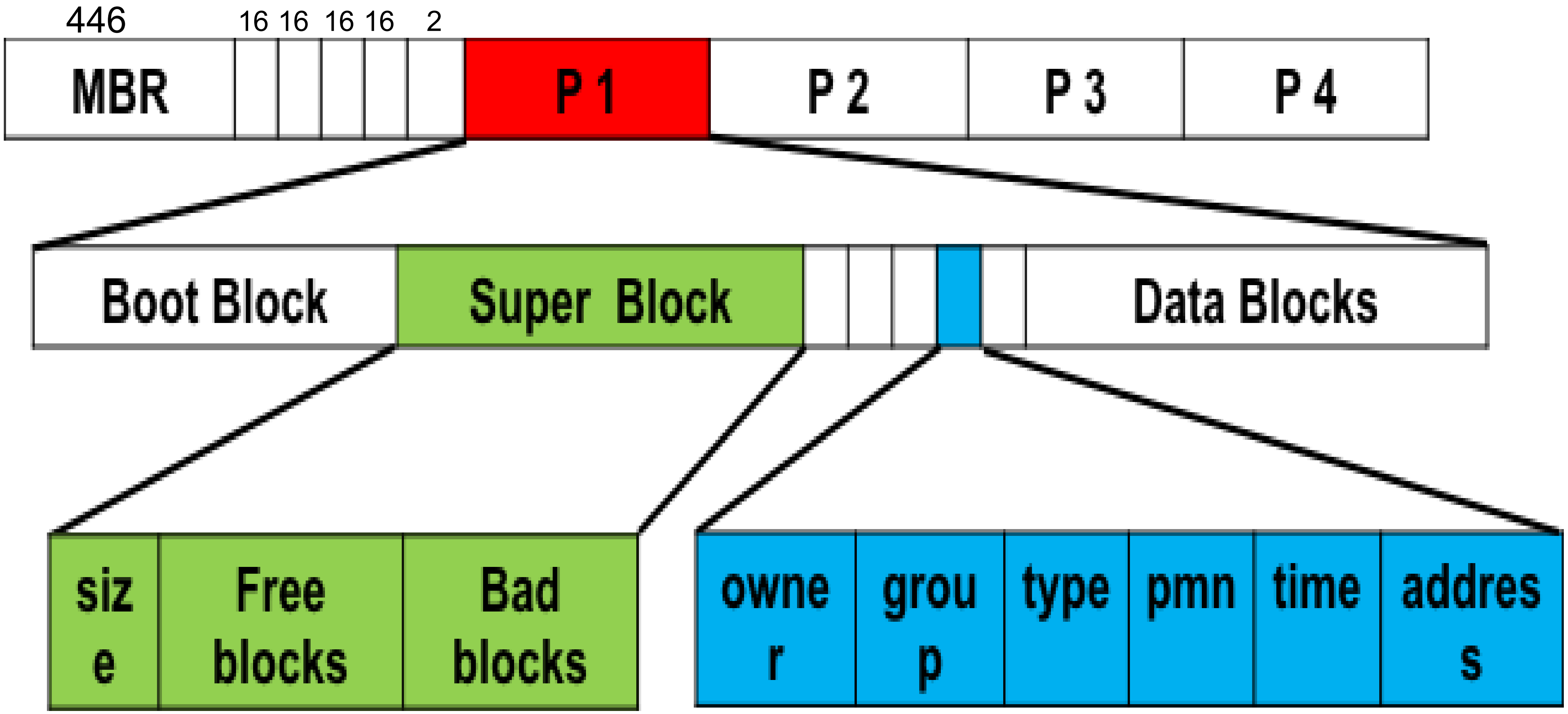
OS with Linux Lec#17
Partitioning a Hard Disk

OS with Linux Lec#18 Formatting a Hard Disk

OS with Linux Lec#19 Mounting a File System

OS with Linux Lec#20
File System Architecture

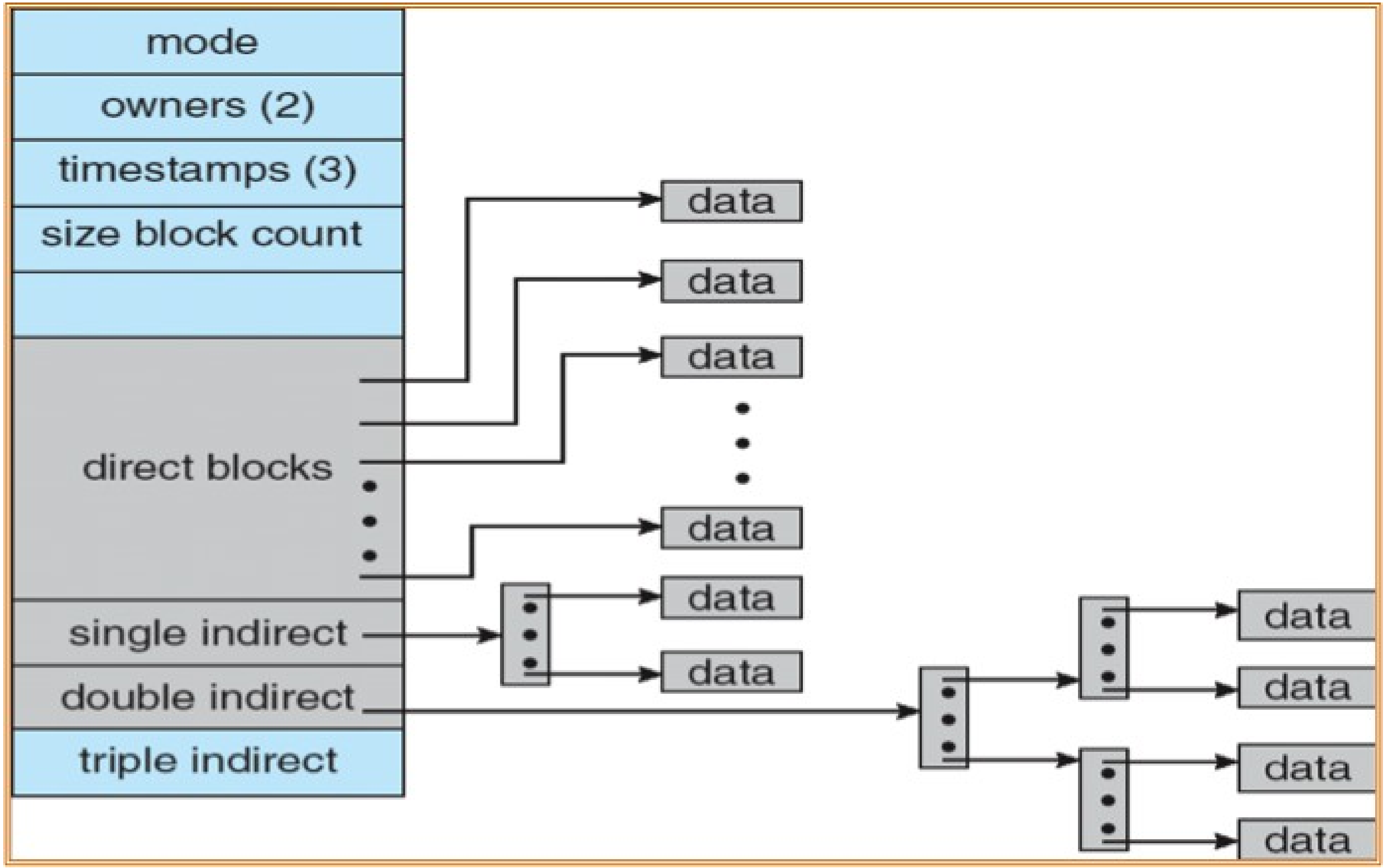
Schematic Structure of a Unix File System



1.FS type of this partition
2.Data block size
3.Total blocks
4.Info about free and allocated blocks

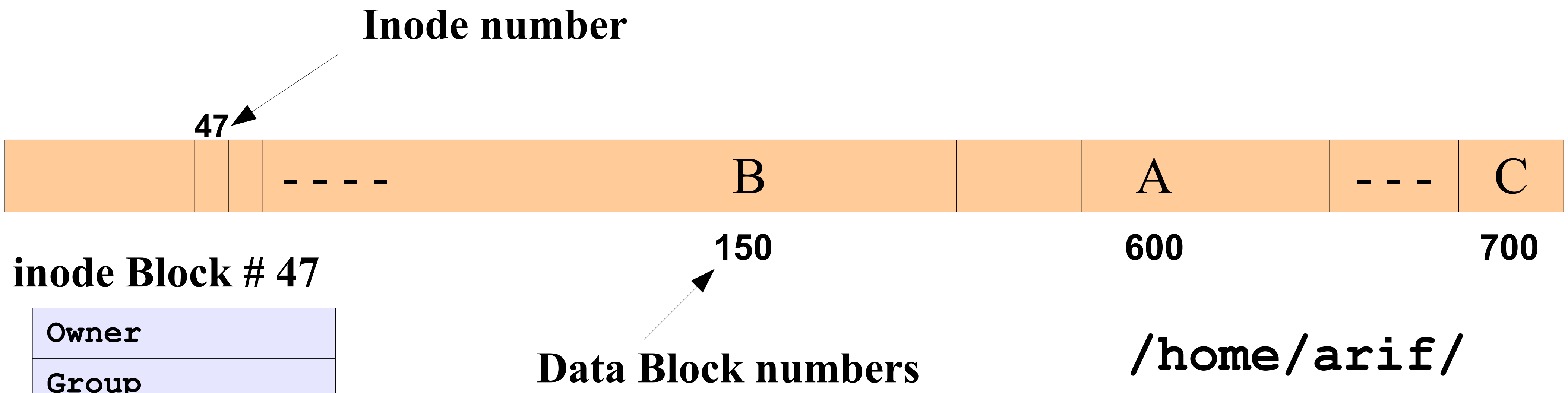
```
sudo tune2fs -l /dev/sda1 | less
df -i /dev/sda1
```

Structure of UNIX Inode



File System in Practice (Creating a file)

```
$ echo "This is text......" 1> /home/arif/f1.txt
```



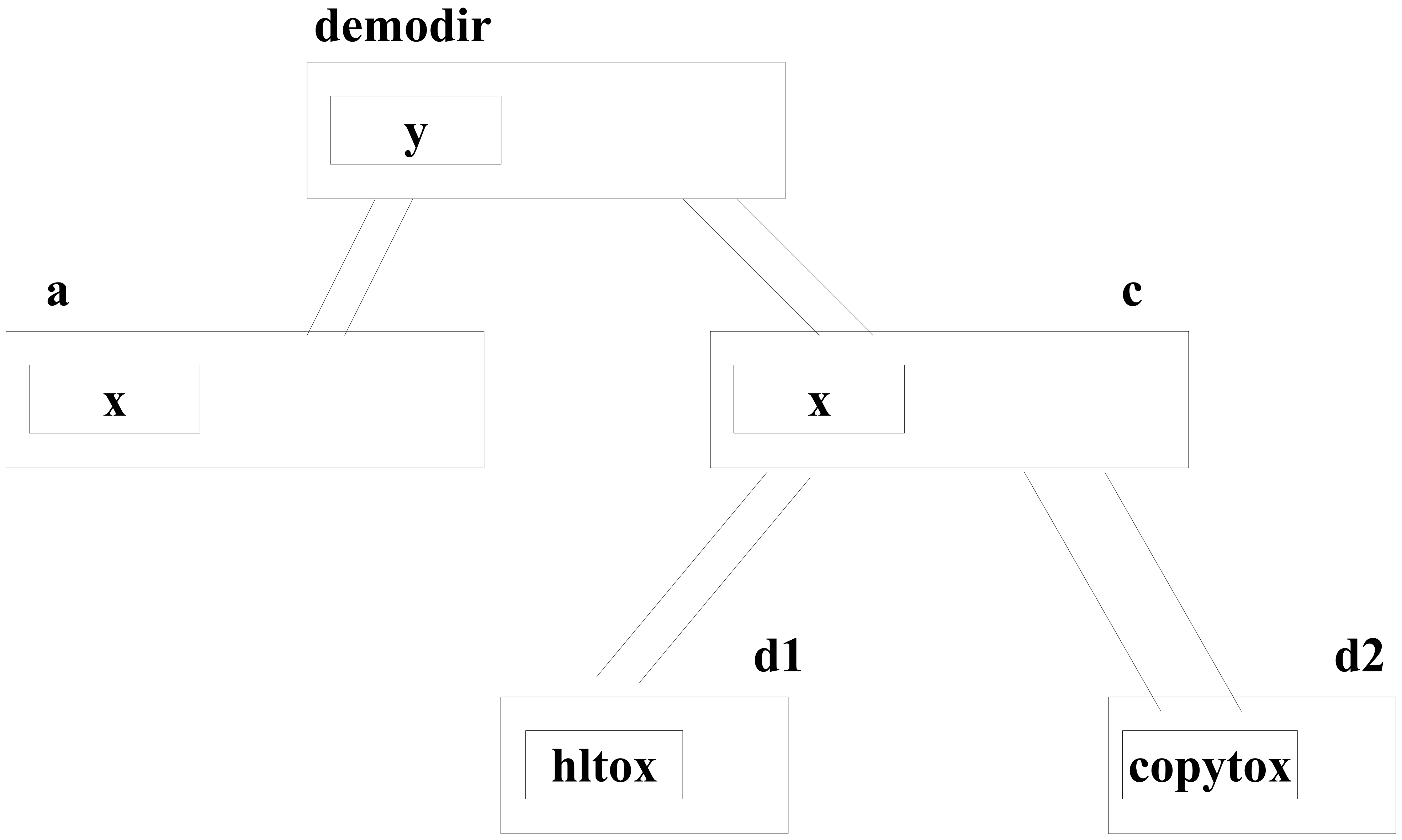
inode Block # 47

Owner
Group
Time Stamps
File Size
Permissions
10 x Dir Ptrs ... ...
Single I.D ptr
Double I.D ptr
Triple I.D ptr

/home/arif/

54	.
6	..
47	f1.txt
125	f2.txt
34	dir1

File System in Practice (Understanding directories)

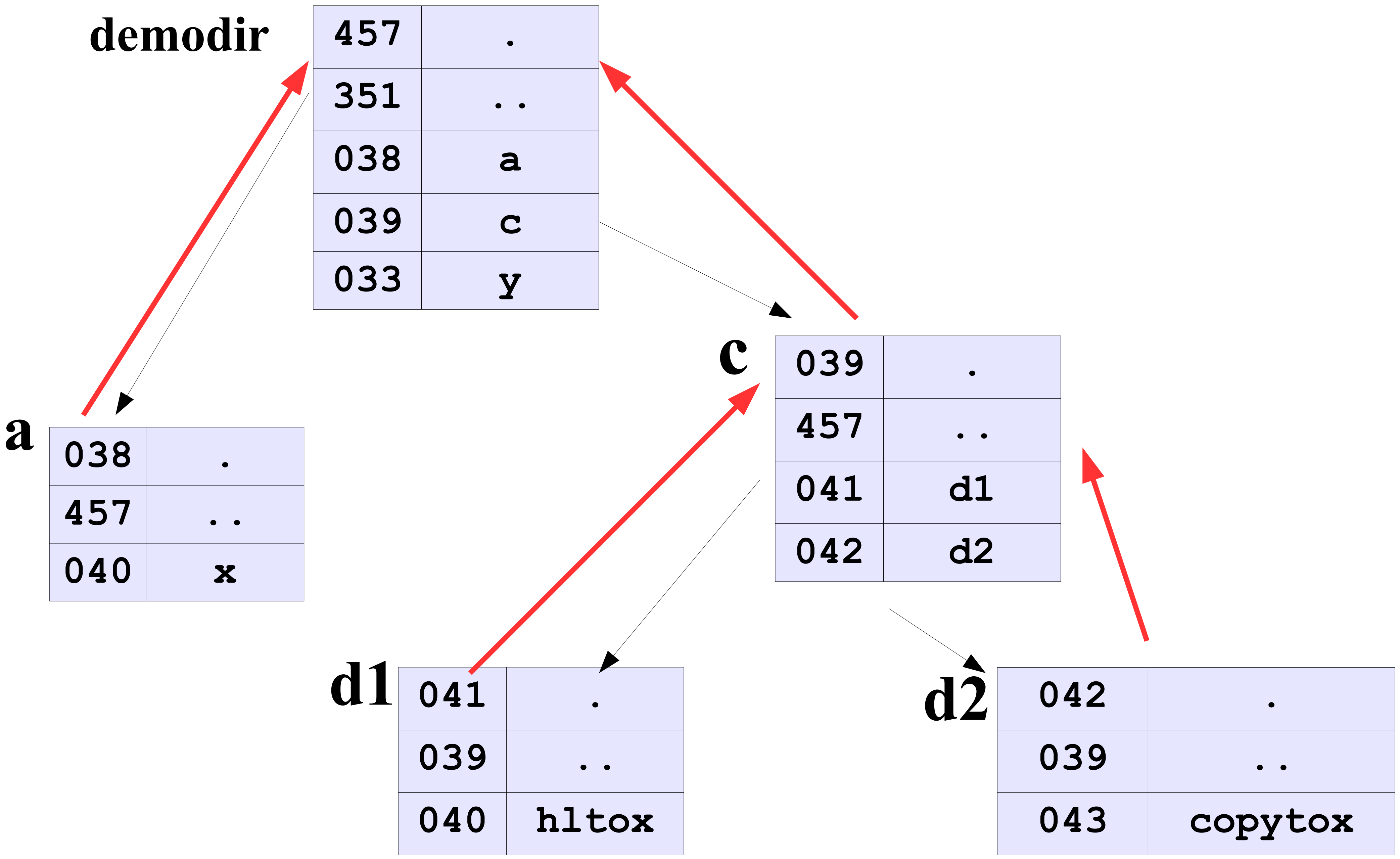


File System in Practice (Understanding directories)

```
$ls -iaR demodir/
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
```

File System in Practice (Understanding directories)



demodir

457	.
351	..
038	a
039	c
033	y

a

038	.
457	..
040	x

c

039	.
457	..
041	d1
042	d2

d1

041	.
039	..
040	hltox

d2

042	.
039	..
043	copytox

File System in Practice (Accessing a file)

```
$ cat /home/arif/file1
```

root directory

1	.
1	..
4	Bin
7	Dev
6	home

Block 190 is /home directory

6	.
1	..
21	rauf
54	arif
30	jamil

Block 535 is /home/arif directory

54	.
6	..
91	mydata
32	file1
28	os

6	mode	190	time	- - -

54	size	535	time	- - -

32	size	555	time	- - -

- Searches directories for file name
- Locate and read inode 32
- Checks for permissions. (userID vs file owner/gp/others)
- Go to the data blocks one by one, the first 10 block addresses are in inode block. Next in single, double and tripple indirect blocks

Review
Connection of an Opened File

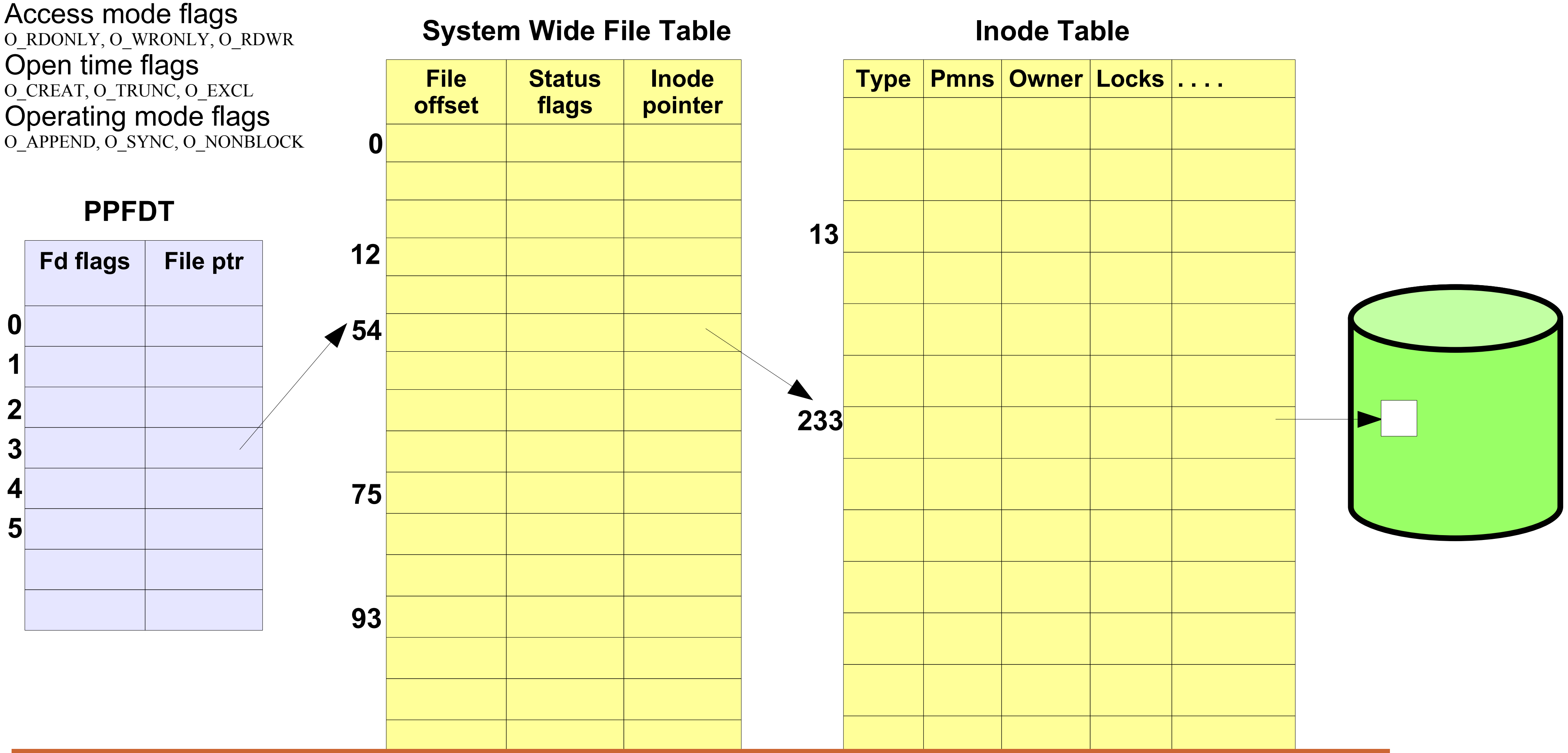
File Descriptor to File Contents

File Status Flags

Access mode flags
O_RDONLY, O_WRONLY, O_RDWR

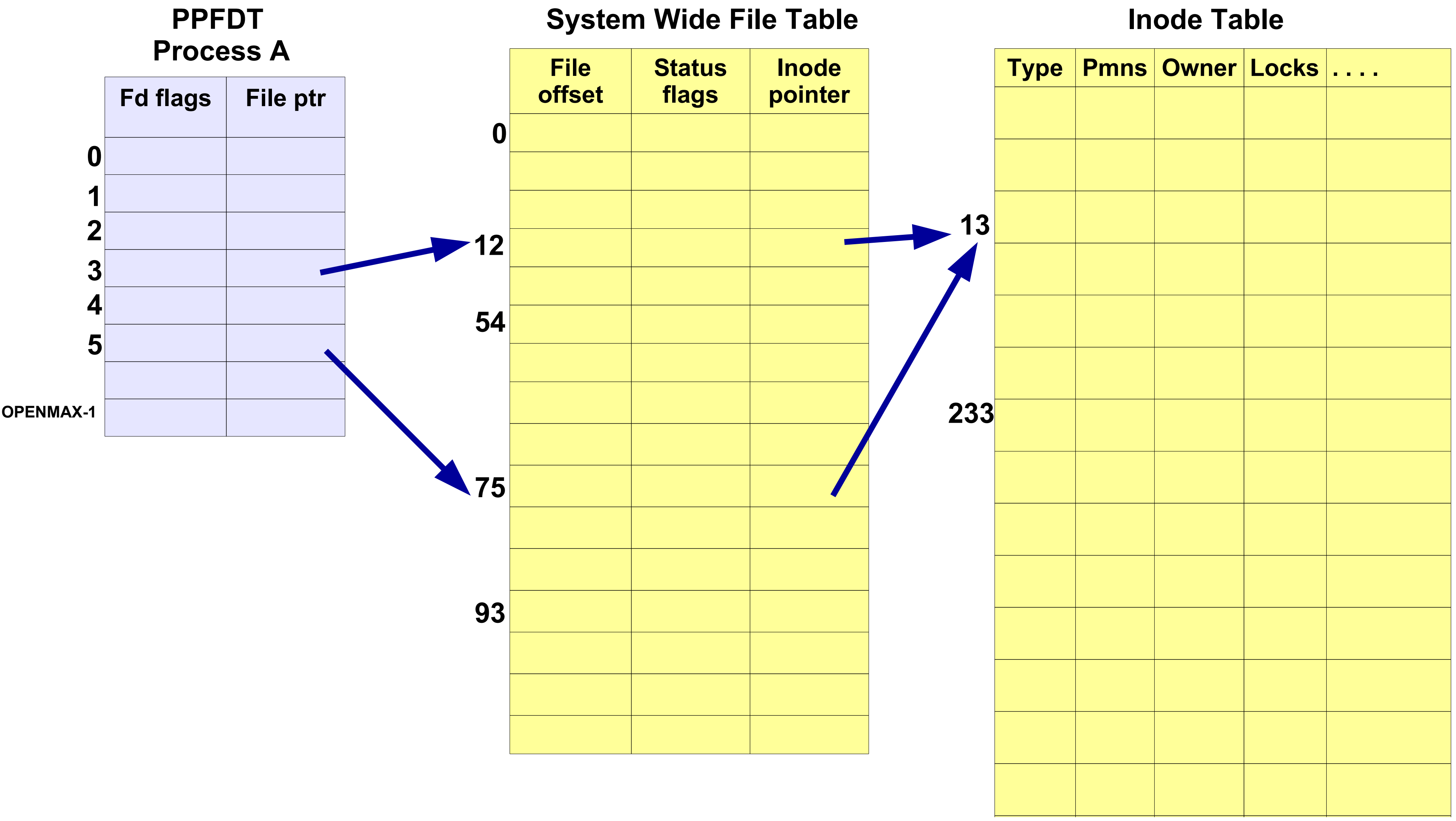
Open time flags
O_CREAT, O_TRUNC, O_EXCL

Operating mode flags
O_APPEND, O_SYNC, O_NONBLOCK



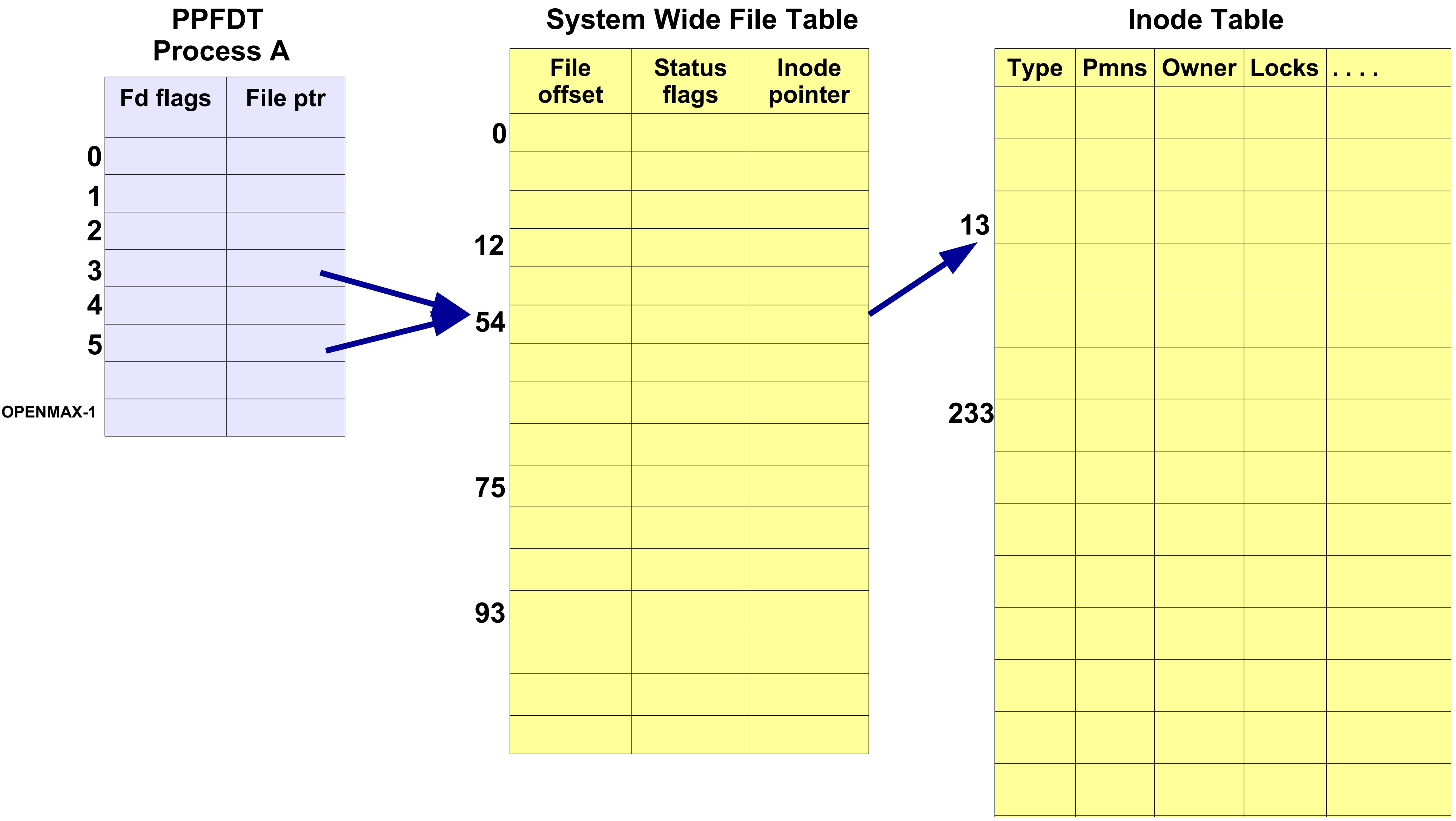
File Descriptor	Purpose	POSIX Name	stdio Stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

Relationship between fd and Open files



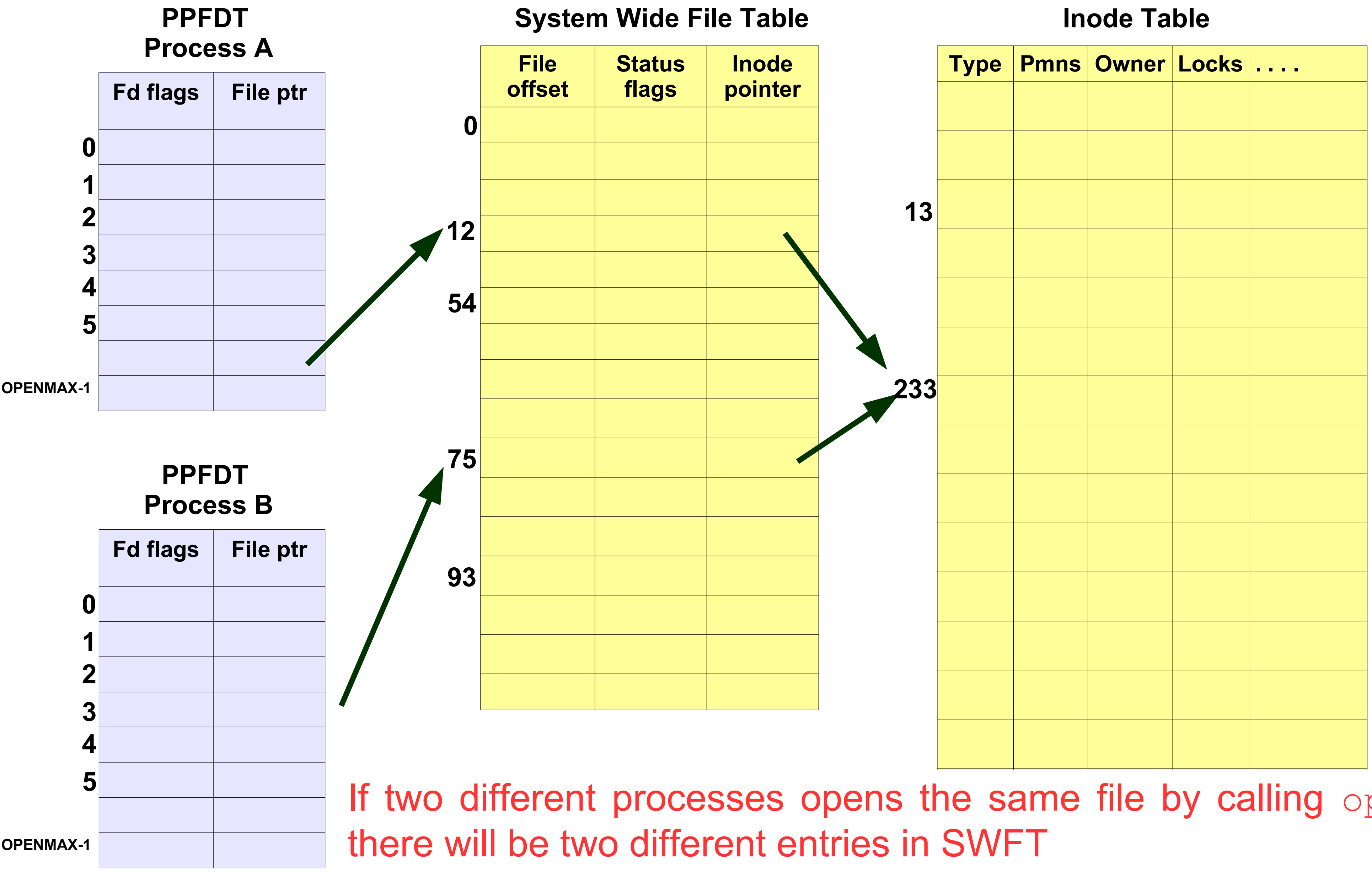
A process can open a file twice. If this is done by calling `open()` twice, then there will be two different entries in PPFTD as well as in SWFT for that single file

Relationship between fd and Open files



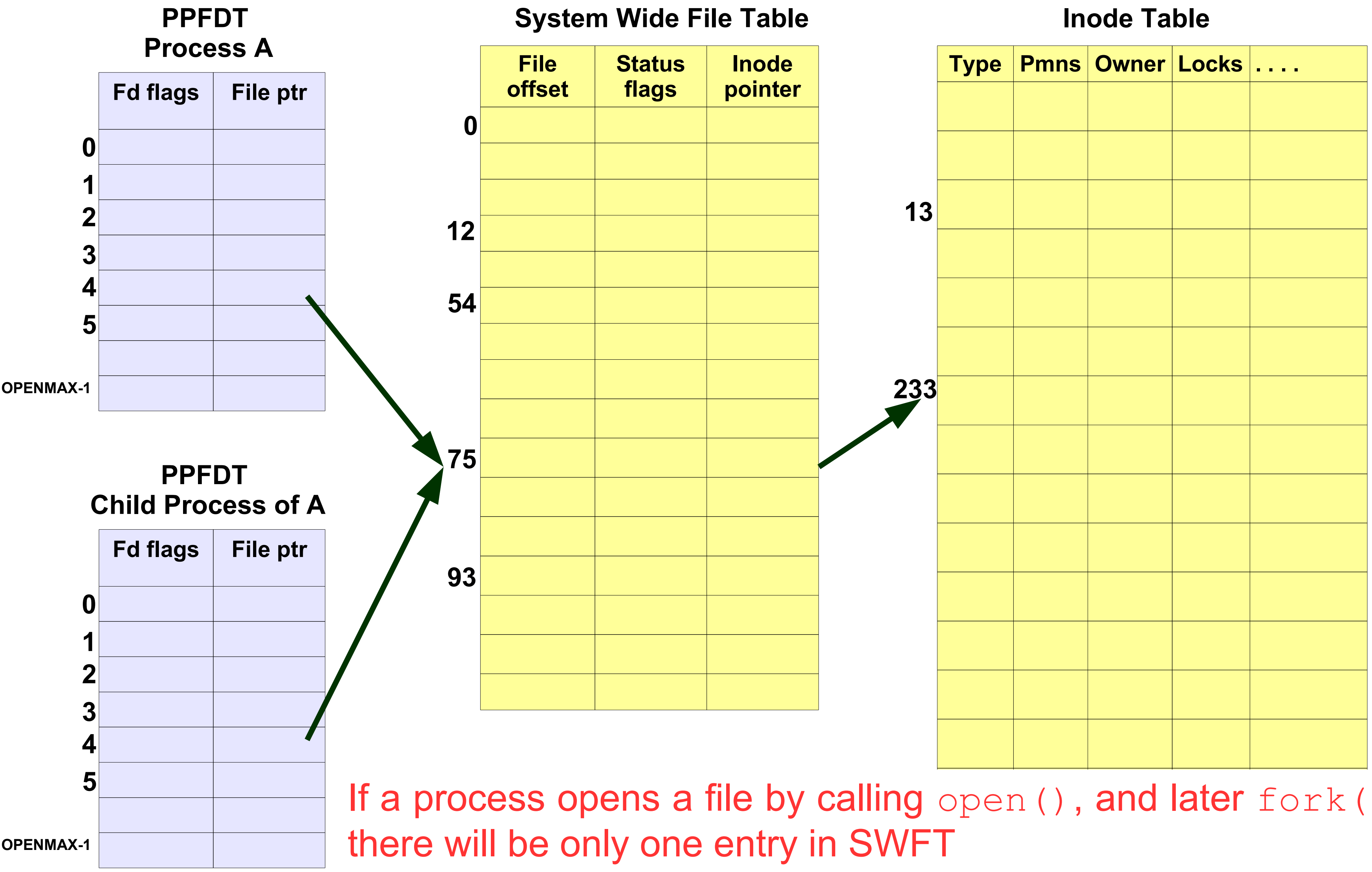
A process can open a file twice. If this is done by calling `dup()`, then there will be two entries in PPFDT but only one entry in SWFT

Relationship between fd and Open files



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

Relationship between fd and Open files



If a process opens a file by calling `open()`, and later `fork()`, then there will be only one entry in SWFT

Universal I/O Modal

open(), read(), write(), close() paradigm

Following are the four key system calls for performing file I/O (programming languages and software packages typically employ these calls indirectly via I/O libraries):

- **`fd = open(pathname, flags, mode)`** opens the file identified by pathname, returning a file descriptor used to refer to the open file in subsequent calls. If the file doesn't exist, `open()` may create it, depending on the settings of the flags bit- mask argument. The flags argument also specifies whether the file is to be opened for reading, writing, or both. The mode argument specifies the permissions to be placed on the file if it is created by this call. If the `open()` call is not being used to create a file, this argument is ignored and can be omitted.
- **`numread = read(fd, buffer, count)`** reads at most count bytes from the open file referred to by fd and stores them in buffer. The `read()` call returns the number of bytes actually read. If no further bytes could be read (i.e., end-of-file was encountered), `read()` returns 0.
- **`numwritten = write(fd, buffer, count)`** writes up to count bytes from buffer to the open file referred to by fd. The `write()` call returns the number of bytes actually written, which may be less than count.
- **`status = close(fd)`** is called after all I/O has been completed, in order to release the file descriptor fd and its associated kernel resources.

`read()` System call

```
#include<unistd.h>
ssize_t read(int fd,void *buf,size_t count);
```

- Attempts to read upto **`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
- 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 `fifos` or `sockets` this is not guaranteed
- On failure, returns -1 and set `errno`. Check reasons in man page
- A return of zero indicates end-of-file

pread() System call

```
#include<unistd.h>
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

```
#include<unistd.h>
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 f le 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 `fifos` or `sockets` this is not guaranteed
- On failure; Return -1 and `errno` is set appropriately. Check reasons in man page
- Return 0 indicates nothing is written

pwrite() System call

```
#include<unistd.h>
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

`close()` System call

```
#include<unistd.h>
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

Restarting a System call

- Once performing blocking I/O using a `read()` or `write()` system calls, if the call is interrupted during its execution we need to restart the system call. A `read()` on a keyboard normally blocks if the user has not entered anything. Similarly if a `read()` is trying to read an empty pipe it blocks
- In such scenarios, most modern UNIX implementations restart such system calls automatically. However, if you are not sure whether your code would be running on such a system, you need to write code to explicitly handle the restarting of an interrupted system call

```
repeat:
  if((rv = read(fd, buff, SIZE)) == -1){
    switch(errno){
      case EINTR: goto repeat;
      ……….....
    }
  }
```

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 used by open ( )

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 bit-mask 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(STDIN_FILENO);
fd = open(pathname, O_RDONLY);
```

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

`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, although it may still be seen in older programs
- 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()`

Things To Do

If you have problems visit me in counseling hours. . . .