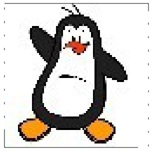


Lecture # 2.1
UNIX File System Architecture

Course: Advanced Operating System

Instructor: Arif Butt

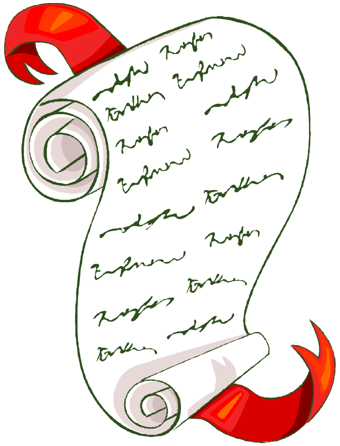
Punjab University College of Information Technology (PUCIT)
University of the Punjab

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

Agenda

- Repositioning current file offset using `lseek()`
- Creating and deleting hard and soft links to a file using `link()`, `symlink()`, `unlink()`, and `remove()`
- Changing ownership of a file using `chown()`, and `fchown()`
- Changing file mode creation mask and permissions on a file using `umask()`, `chmod()`, and `fchmod()`
- Checking permissions on a file using `access()`
- I/O redirection using `dup()`, and `dup2()`
- What all we can do with `fcntl()`



OS with Linux Lec#16
Hard Disk Geometry

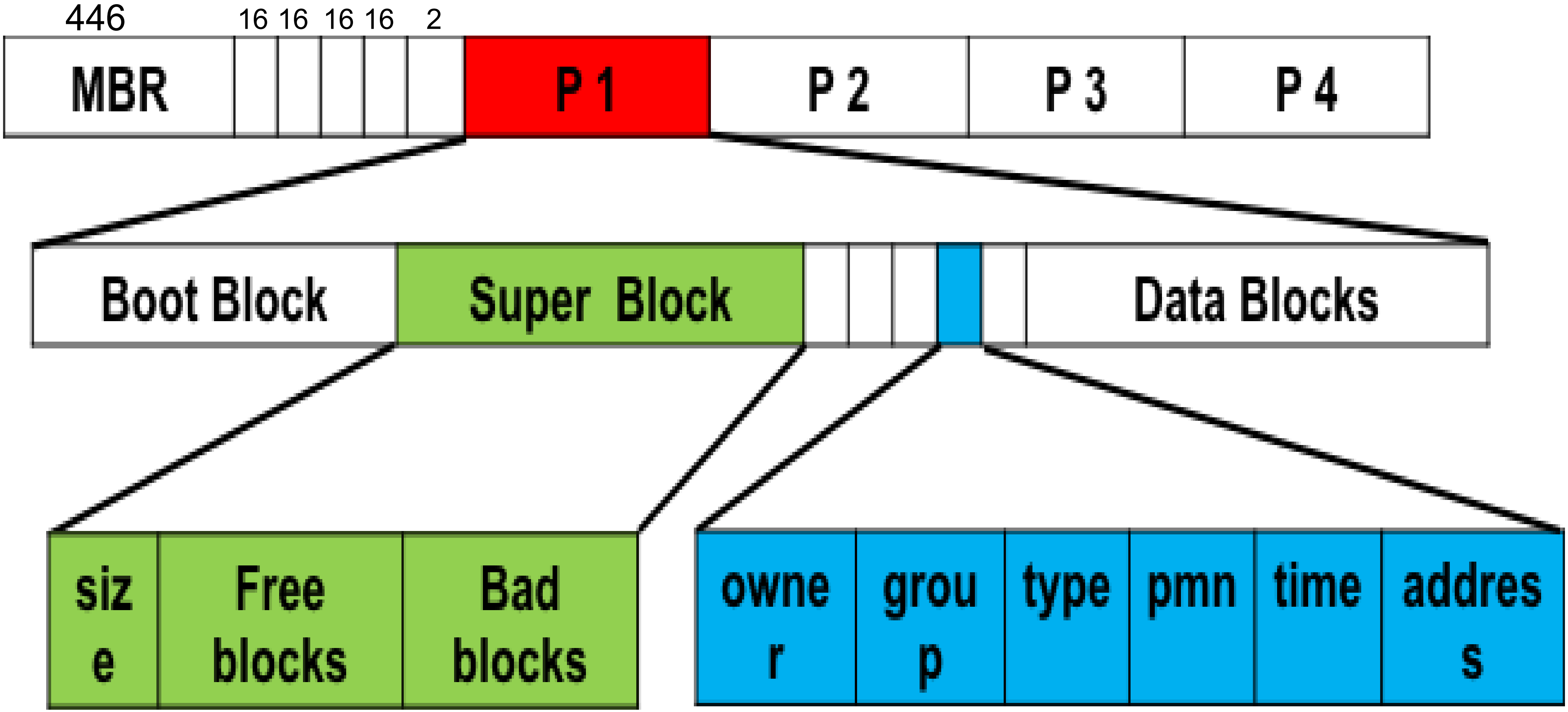
OS with Linux Lec#17
Partitioning a Hard Disk

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

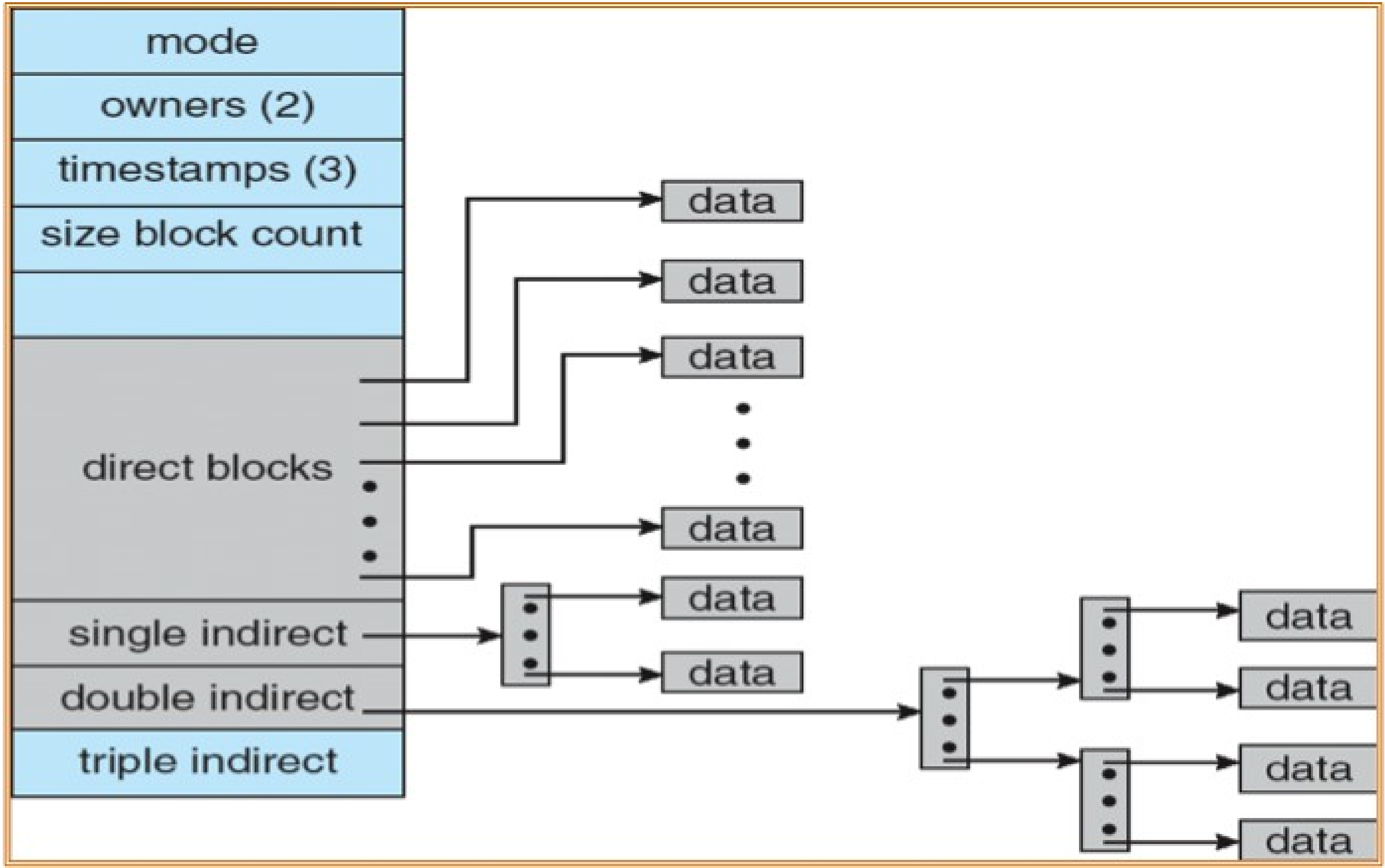
Schematic Structure of a Unix File System



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

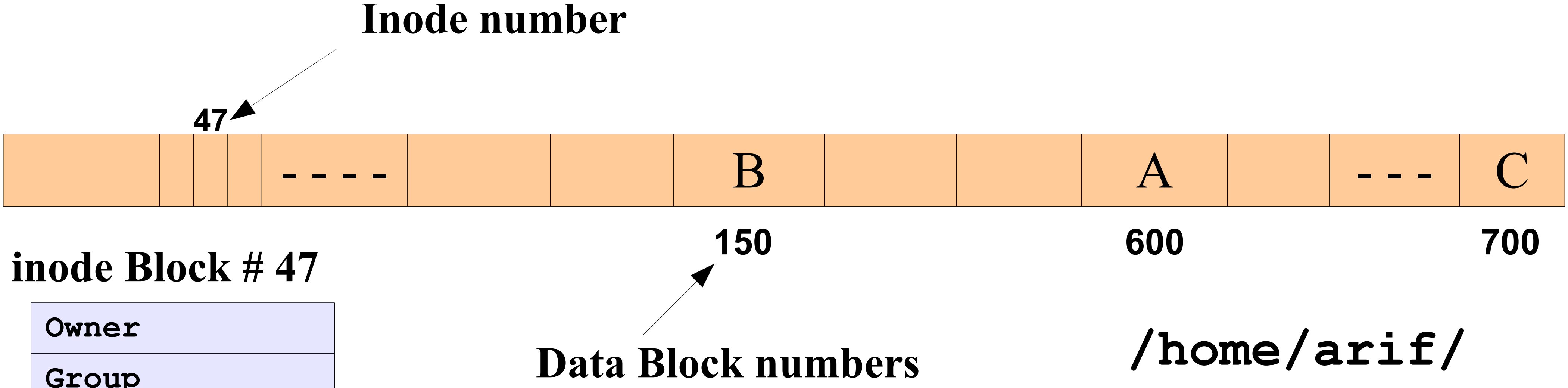
```
sudo tune2fs -l /dev/sda1 | less
df -i /dev/sda1
```

Structure of UNIX Inode



File System in Practice (Creating a file)

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



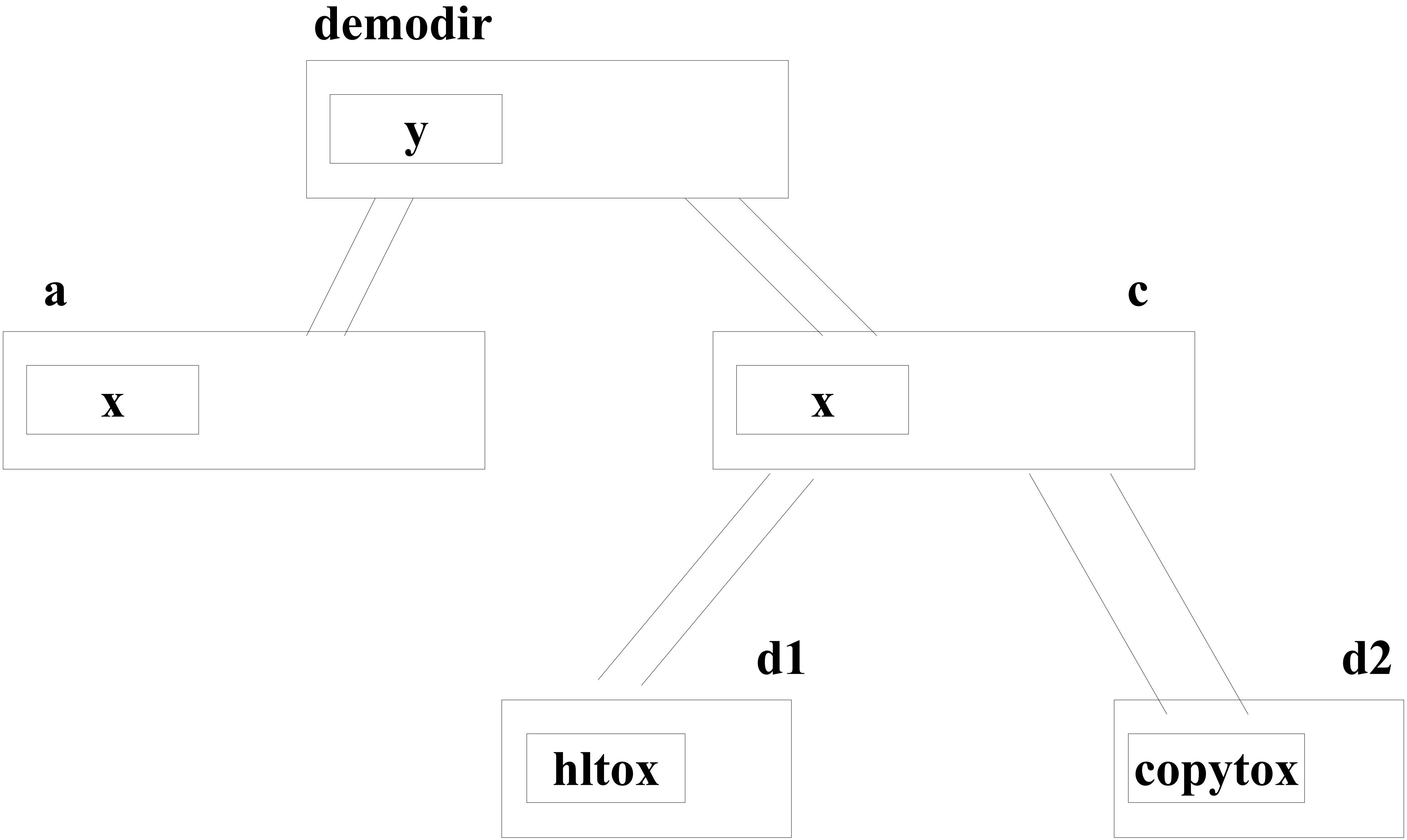
inode Block # 47

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)

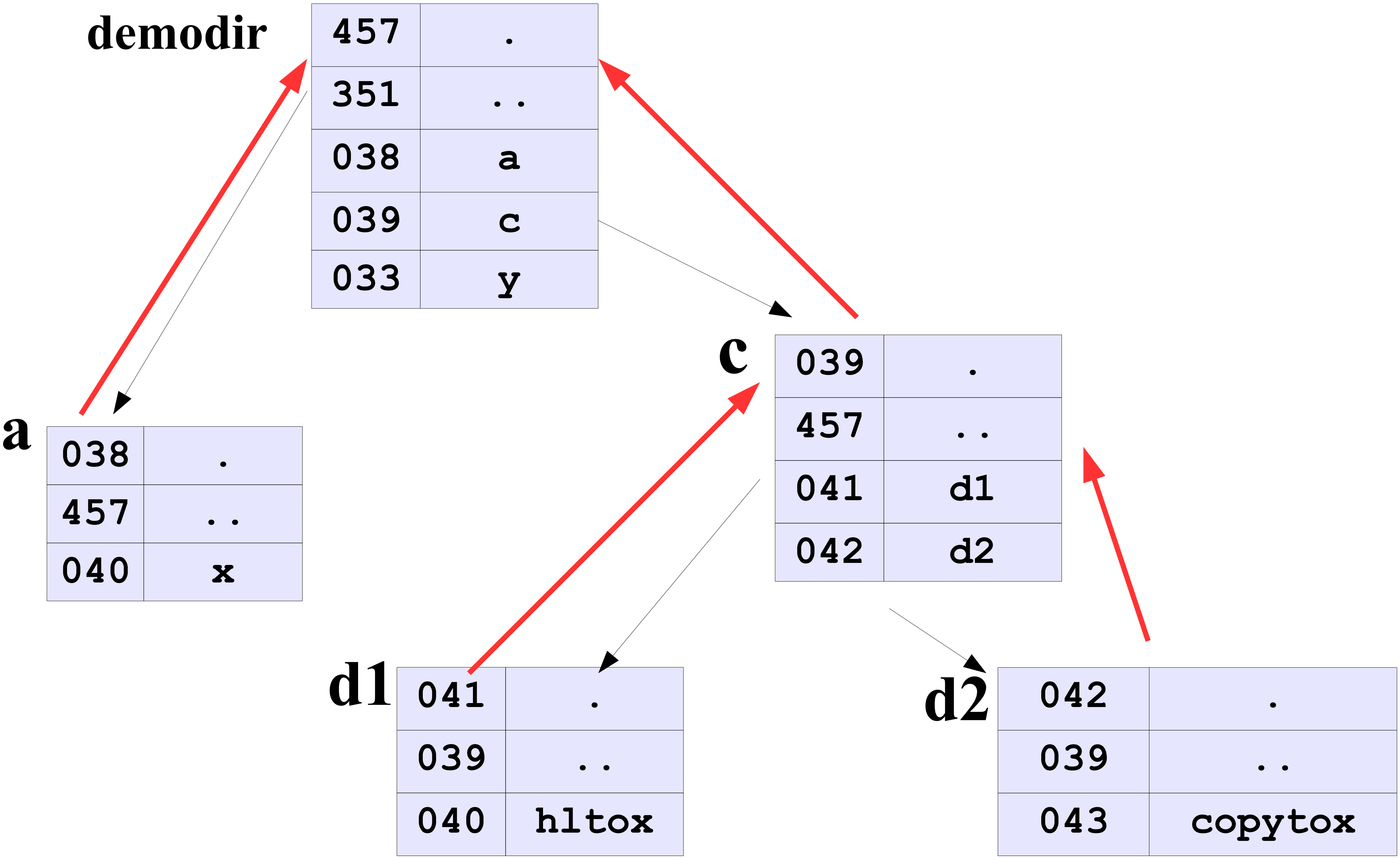


File System in Practice (Understanding directories)

```
$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

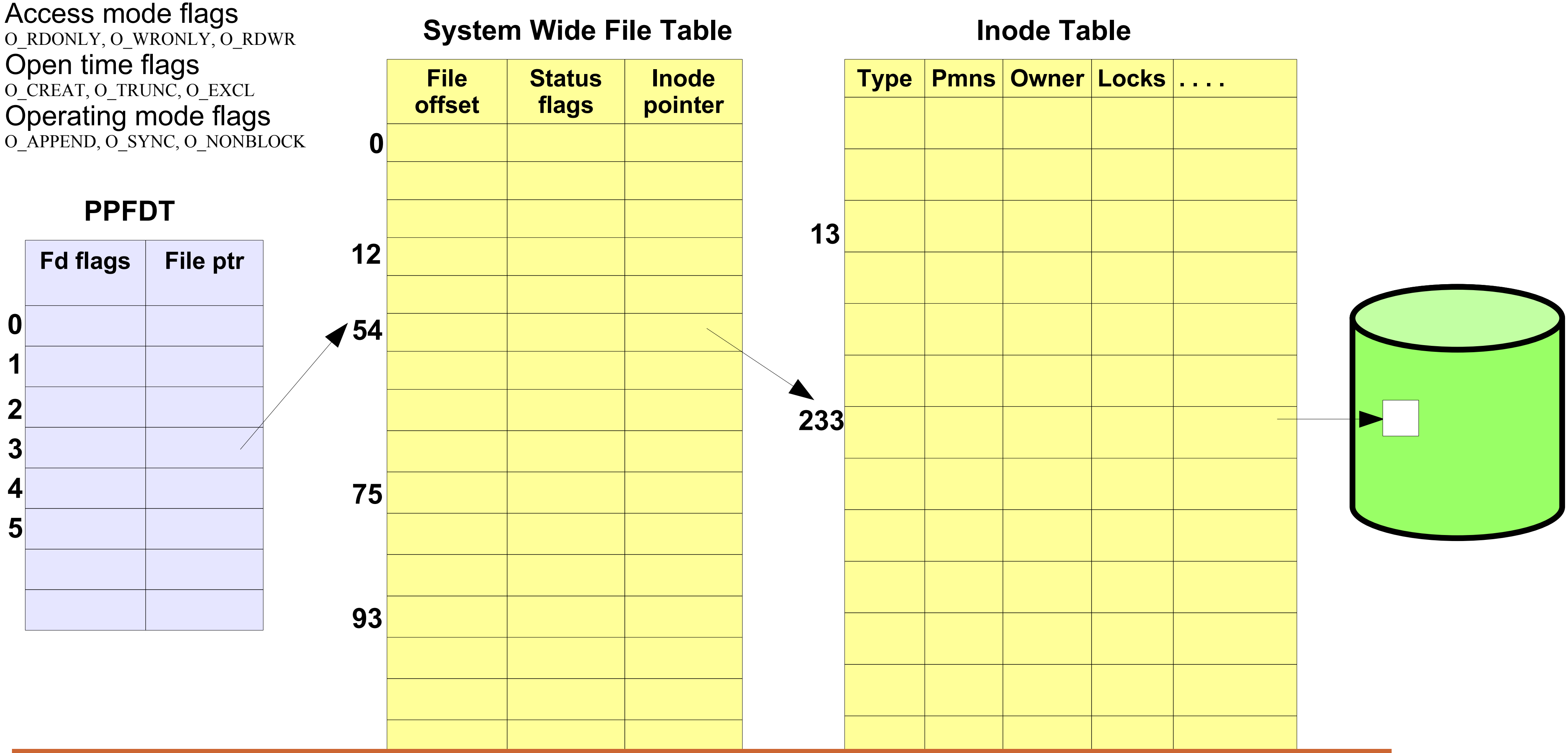
File Descriptor to File Contents

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



PPFDT

Fd flags | File ptr

0 1 2 3 4 5

System Wide File Table

File offset | Status flags | Inode pointer

0 12 54 75 93

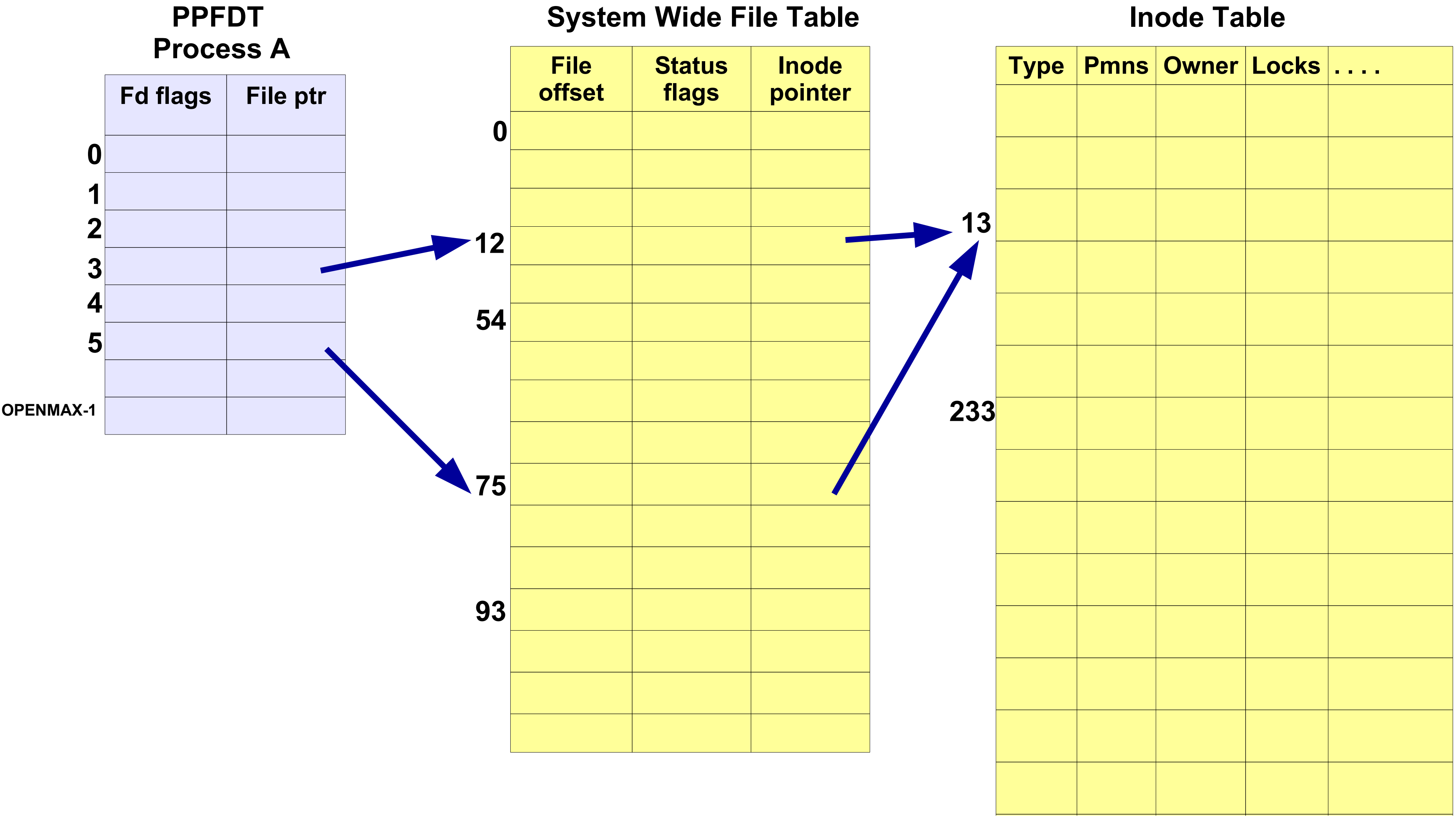
Inode Table

Type | Pmns | Owner | Locks | . . . .

13 233

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



PPFDT
Process A

Fd flags | File ptr

0
1
2
3
4
5
OPENMAX-1

System Wide File Table

File offset | Status flags | Inode pointer

0
12
54
75
93

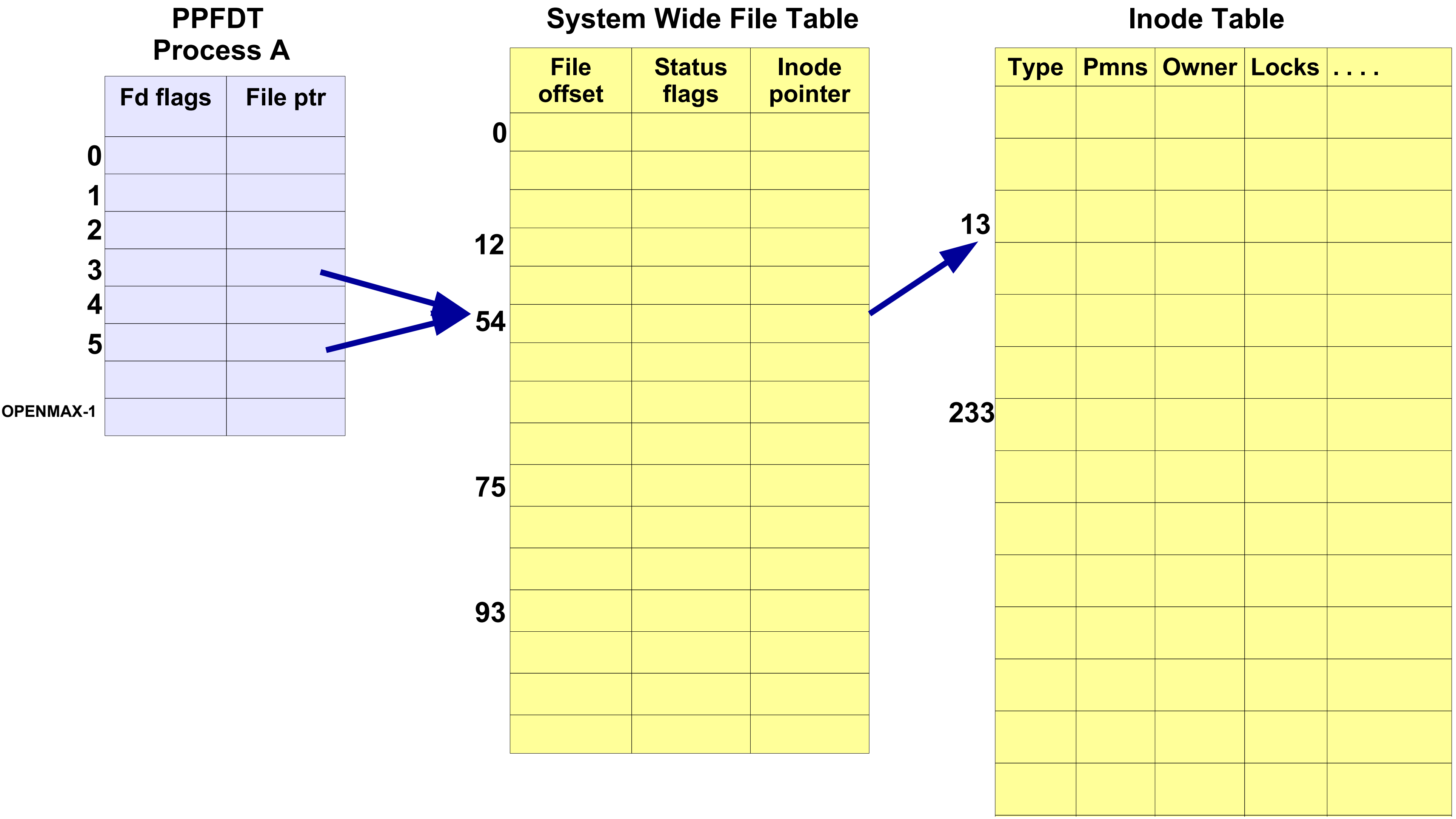
Inode Table

Type | Pmns | Owner | Locks | . . . .

13
233

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



PPFDT
Process A

Fd flags | File ptr

0
1
2
3
4
5
OPENMAX-1

System Wide File Table

File offset | Status flags | Inode pointer

0
12
54
75
93

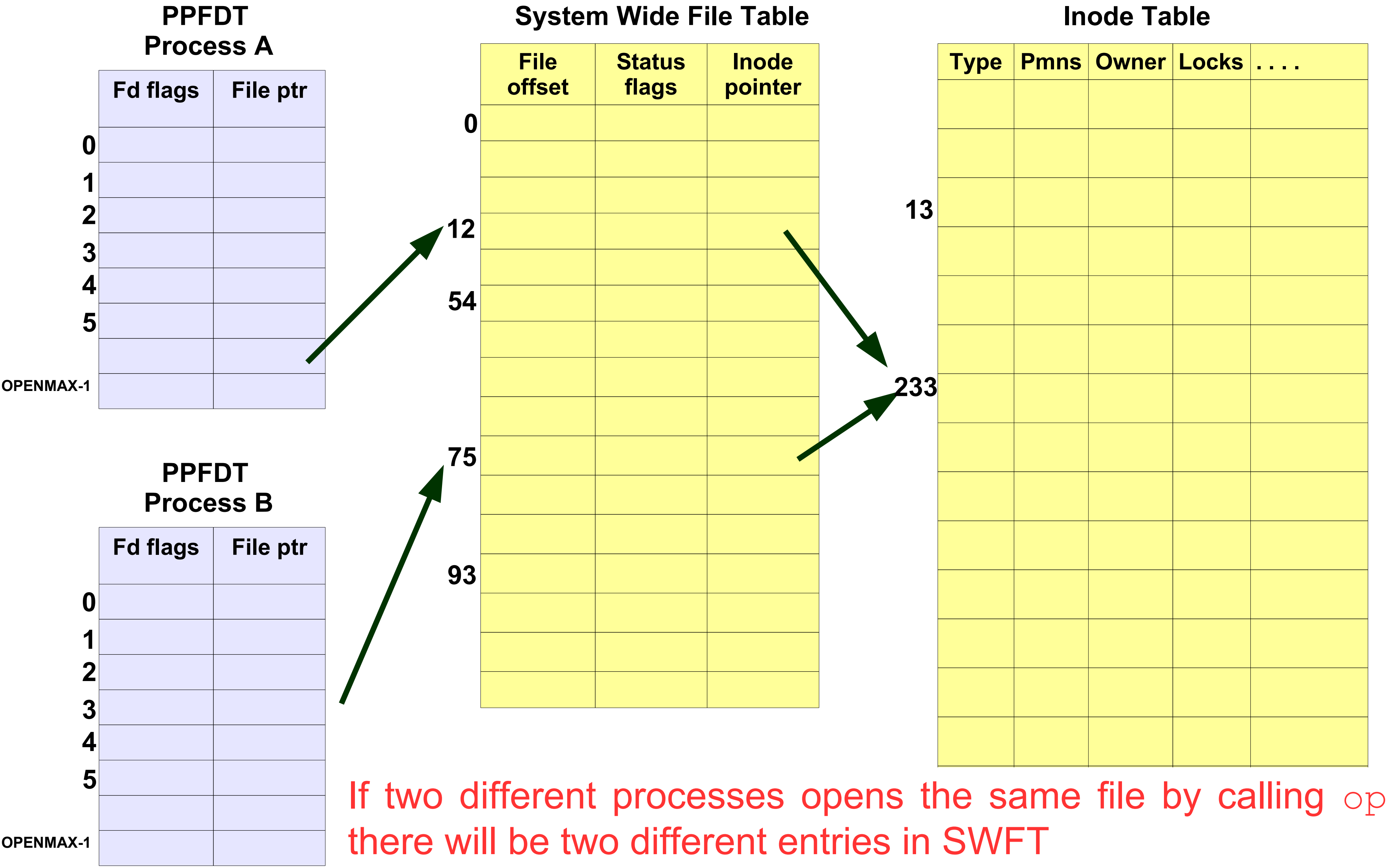
Inode Table

Type | Pmns | Owner | Locks | . . . .

13
233

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



PPFDT Process A

Fd flags | File ptr

0 1 2 3 4 5 OPENMAX-1

PPFDT Process B

Fd flags | File ptr

0 1 2 3 4 5 OPENMAX-1

System Wide File Table

File offset | Status flags | Inode pointer

0 12 54 75 93

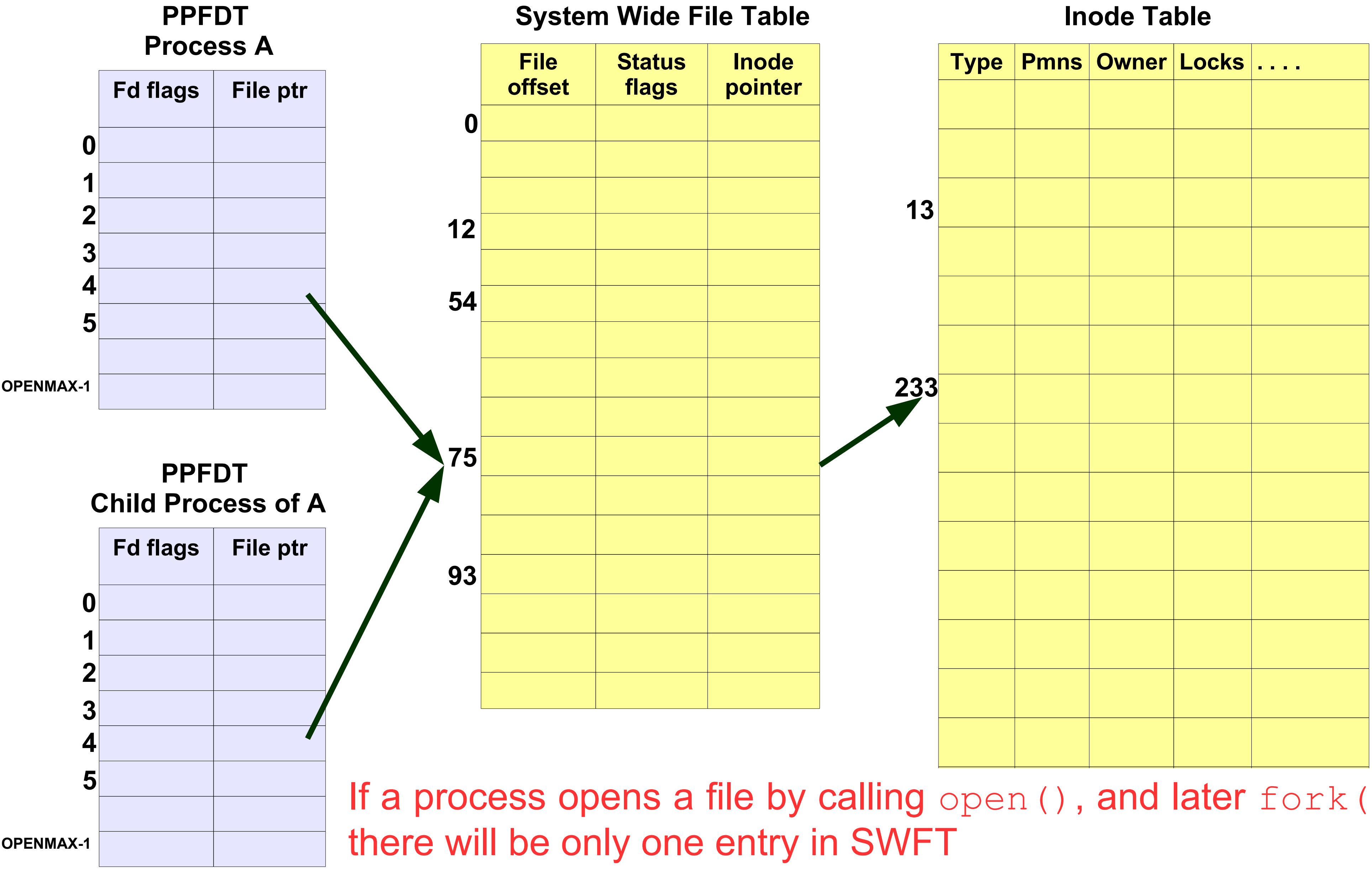
Inode Table

Type | Pmns | Owner | Locks | . . . .

13 233

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



PPFDT
Process A

Fd flags | File ptr

0
1
2
3
4
5
OPENMAX-1

PPFDT
Child Process of A

Fd flags | File ptr

0
1
2
3
4
5
OPENMAX-1

System Wide File Table

File offset | Status flags | Inode pointer

0
12
54
75
93

Inode Table

Type | Pmns | Owner | Locks | . . . .

13
233

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 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 `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()`

Repositioning CFO of an opened file

lseek() System call

```
off_t lseek(int fd,off_t offset,int whence);
```

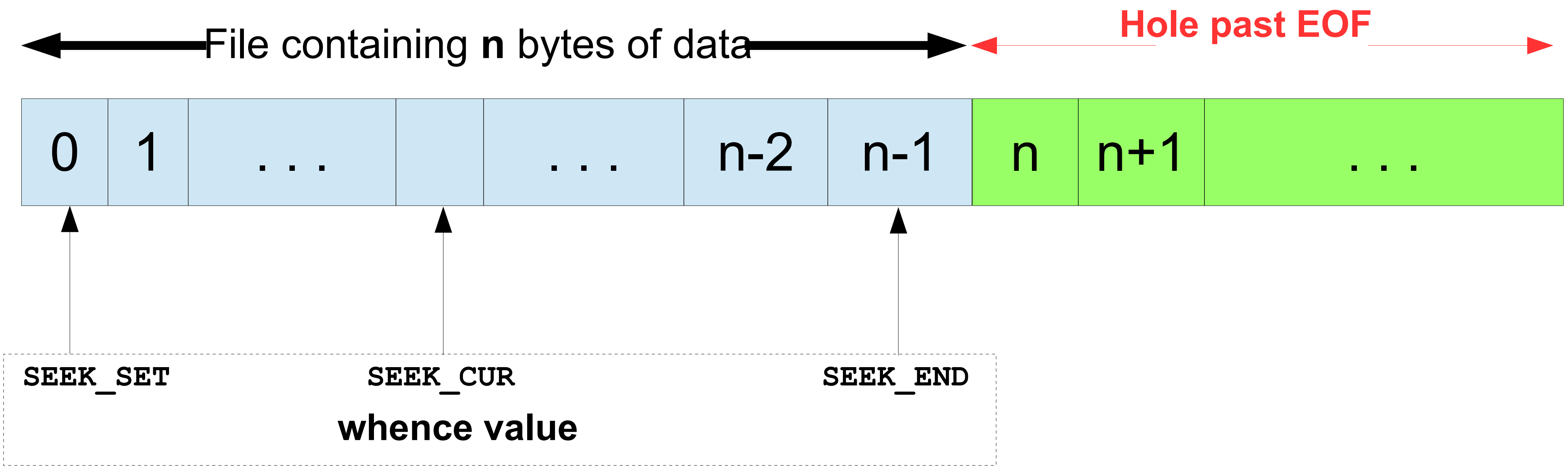
- For each open file, the kernel records a file offset, also called current file offset (cfo), which is there in the SWFT. This is the location in the file at which the next `read()` or `write()` will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0

- The file offset is set to point to the start of the file when the file is opened (unless the O_APPEND option is specified) and is automatically adjusted by each subsequent call to `read()` or `write()` so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive `read()` and `write()` calls progress sequentially through a file

- The `lseek()` system call adjusts the file offset of the open file referred to by the file descriptor `fd`, according to the values specified in `offset` and `whence`. On success, returns the resulting offset location and -1 on failure

Interpreting whence argument of `lseek()`

```
off_t lseek(int fd,off_t offset,int whence);
```



File containing **n** bytes of data

Hole past EOF

0	1	. . .		. . .	n-2	n-1	n	n+1	. . .

SEEK_SET SEEK_CUR SEEK_END

whence value

Examples

```
off_t posn;
posn = lseek(fd, 54, SEEK_SET);
posn = lseek(fd, +/-54, SEEK_CUR);
posn = lseek(fd, +/-54, SEEK_END);
```

`lseek()` System call (cont...)

- The directive “**whence**” can take following five values:

WHENCE		Description
`SEEK_SET`	`0`	The cfo is set `offset` bytes from the beginning of the file
`SEEK_CUR`	`1`	The cfo is set `offset` bytes from current value of cfo
`SEEK_END`	`2`	The cfo is set `offset` bytes from the end of the file
`SEEK_HOLE`	`3`	The cfo is set to start of the next hole greater than or equal to `offset`
`SEEK_DATA`	`4`	The cfo is set to start of the next non-hole (i.e., data region) greater than or equal to `offset`

Examples:

`lseek1.c, lseek2.c, lseek3.c`

Misc File Related System Calls

rename() Function

```
int rename(const char*oldpath,const char* newpath );
```

- A programmer can rename a file or a directory with the **rename()** library function
- A sample code snippet that renames a file named `file1.txt` to `file2.txt` in the present working directory is shown below:

```
if(rename(“file1”,”file2”) == -1)
            perror(“rename(1)”);
```

remove() and unlink()

```
int remove(const char *pathname);
int unlink(const char* pathname);
```

- Remove is a library call that deletes a name from file system. It calls **`unlink()`** for files and **`rmdir()`** for directories
- However, if any process has this file open currently, the file won't be actually erased until the last process holding it open closes it. Until then it will be removed from the directory (i.e., `ls` won't show it), but not from disk
- When a file is deleted, the OS Kernel performs following tasks:

  i. Frees the inode number associated with that file

  ii. Frees all the data blocks associated with that file and add them to the list of free blocks

  iii. Delete the entry from the directory containing that file
- The metadata of the file is still there in the inode block and the data of the file in its data blocks (U just need to know how to access those blocks)

Symlink and link Function

```
int symlink(const char* oldpath, const char* newpath);
int link(const char* oldpath, const char* newpath);
```

- The `link()` and `symlink()` functions are used to create a hard link and a soft link to a file
- Following sample code snippets show the usage of these library functions:

```
if(symlink(“/tmp/file1”,”/home/arif/slinktofile1”) == -1)
      { perror(“symlink”); exit(1);}
```

```
if(link(“/tmp/file1”,”/home/arif/hlinktofile1”) == -1)
      { perror(“link”); exit(1); }
```

Review OS with Linux Video Lec 21 for detailed concepts of Links

chown ,fchown and lchown Function

```
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *linkname, uid_t owner,gid_t group);
```

- These system calls change the owner and group of the file specified by path or file descriptor
- If owner or group is specified as -1, then that ID is not changed
- Only a process with super user privileges can use these functions to change any file user ID and group ID
- However, if a process effective user ID matches a file user ID and its effective group ID, the process can change the file group ID only (Will discuss this later)
- `lchown()` is like `chown()`, but does not dereference symbolic links

chmod and fchmod System Call

```
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- These two functions allow us to change the file access permissions for an existing file
- The **chmod** function operates on the specified file, whereas the **fchmod** function operates on a file that has already been opened using its file descriptor
- The mode is the same as discussed in the flags argument of `open()`
- Following code snippet will give the owner read and write permissions to the file and deny access to all other users

```
if(chmod(“file.txt”,S_IRUSR|S_IWUSR) == -1){

   perror(“chmod”); exit(1);}
```

umask Function

```
mode_t umask(mode_t mask);
```

- The **umask()** function sets the file mode creation “**mask**” for the process and returns the previous value
- Remember the mask value of a process is the same as that of its creating shell, i.e. its parent. (mask value is inherited after fork)
- The file mode creation mask is used whenever the process creates a new file or a new directory

Review OS with Linux Video Lec 22 and 23

```
umask(0077);
int fd = open("myfile.txt",O_CREAT|O_RDWR,0633);
```

access() System Call

```
int access(const char *pathname, int mode);
```

- The `access()` system call determines whether the calling process has access permission to a file or not and it can also check for file existence as well
- The mode argument is a bit mask consisting of one or more of the permission constants shown in the table below:
- If a process has all the specified permissions the return value is 0, otherwise the return value is -1 & sets errno to `EACCES`
- The `open()` system call performs its access tests based on the EUID and EGID, while the `access()` system call bases its tests on the *real* UID & GID

Mode	**Description**
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission
F_OK	Test for existence of file

Examples:

`access.c, truncate.c, umask1.c, umask2.c`

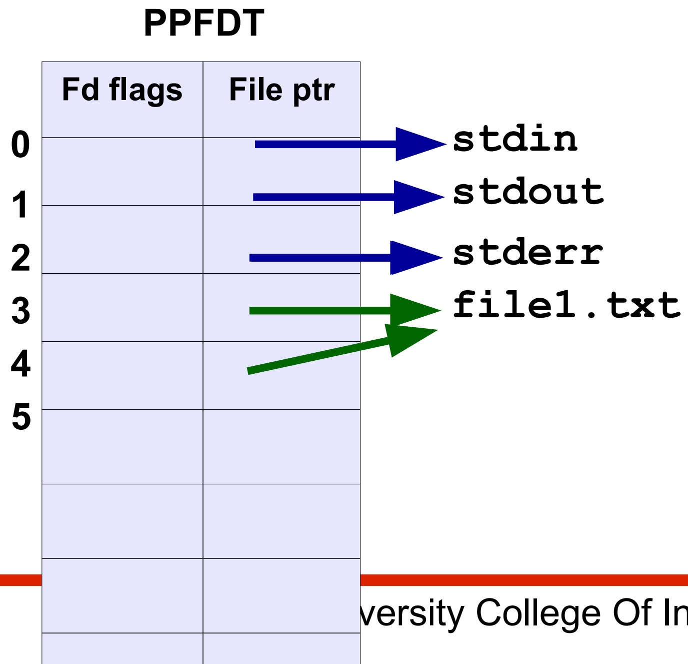
I/O Redirection using dup()

Review OS with Linux Video Lec 8

dup() System call

```
int dup(int oldfd);
```

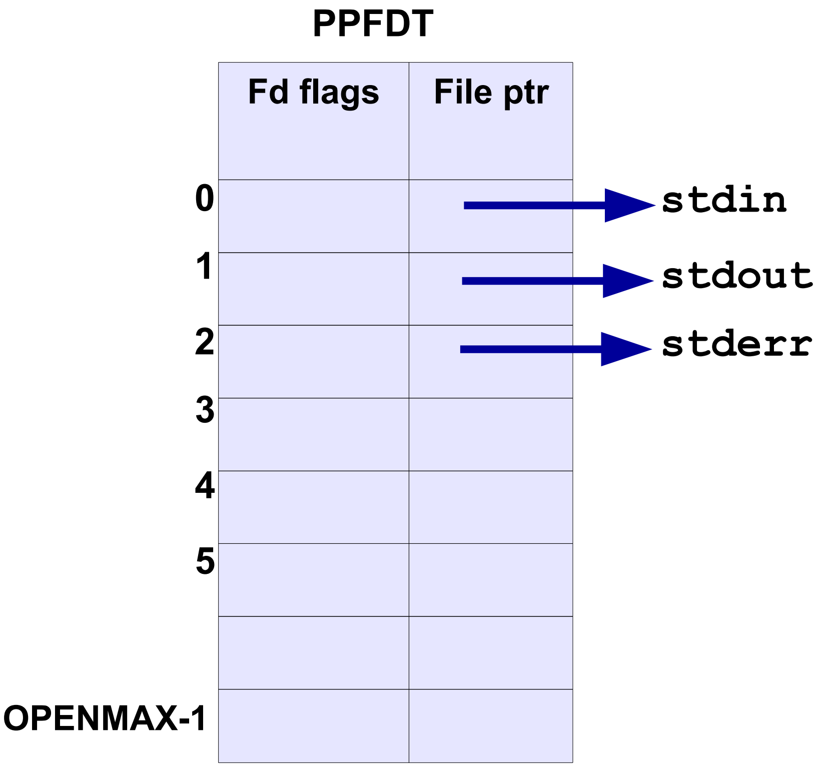
- The `dup()` call takes `oldfd`, an open file descriptor, and returns a new descriptor that refers to the same open file description
- The old and the new descriptor both point to the same entry in the SWFT. After a successful return from these function , old and new fd's can be used interchangeably
- The new descriptor is guaranteed to be the lowest unused file descriptor.



Example: `dup.c`

Facts about I/O Redirection on the Shell

```
$ cat
```

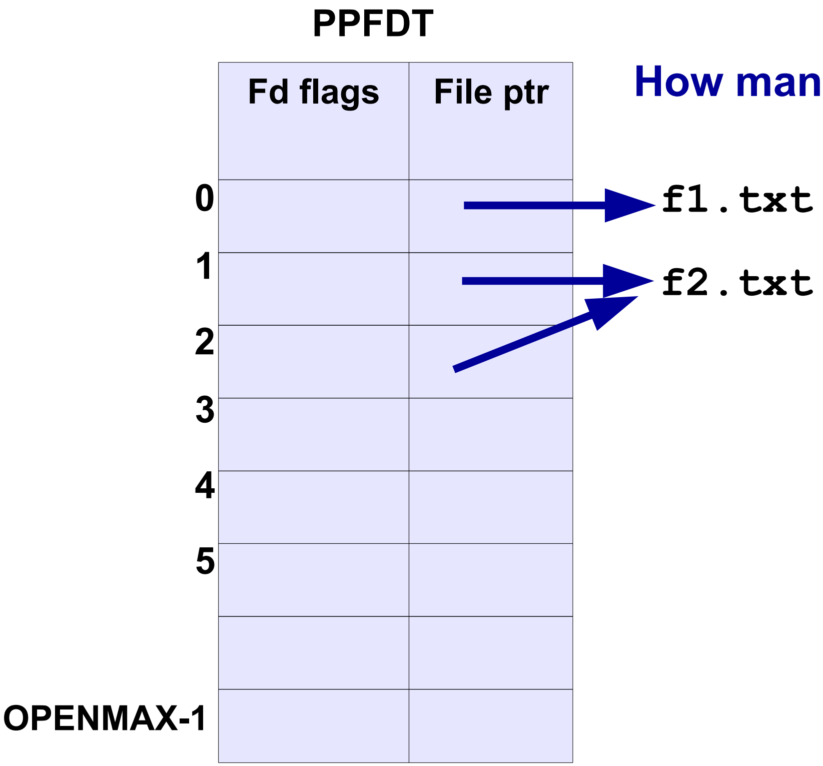


PPFDT

	Fd flags	File ptr	
0		→	stdin
1		→	stdout
2		→	stderr
3			
4			
5			
OPENMAX-1			

Facts about I/O Redirection on the Shell

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



PPFDT

	Fd flags	File ptr
0		
1		
2		
3		
4		
5		
OPENMAX-1		

f1.txt

f2.txt

100$ Q:

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

Example: `listargs.c`

```
$ ./a.out 0< /etc/passwd 1> /dev/tty 2> errfile
```

dup() System call

```
int dup(int oldfd);
```

- We know that `dup()` call guarantees that the new descriptor returned is the lowest unused file descriptor
- If we run the following LOCs, the `open` call will return 3, the `dup` call will return the lowest unused descriptor which will be zero. So finally descriptor zero points to the opened file instead of `stdin`

```
fd = open(...);
close(0);
newfd = dup(fd);
```

- To make the above code simpler, and to ensure we always get the file descriptor we want, we can use **`dup2()`**

dup2() System call

```
int dup2(int oldfd, int newfd);
```

- The `dup2()` system call makes a duplicate of the file descriptor given in `oldfd` using the descriptor number supplied in `newfd`
- If the file descriptor specified in `newfd` is already open, `dup2()` closes it first
- We can simplify the preceding calls to `close(0)` and `dup(fd)` on previous slide to the following:

  `dup2(fd, 0);`
- A successful `dup2()` call returns the number of the duplicate descriptor (i.e., the value passed in `newfd`)
- If `oldfd` is a valid file descriptor, and `oldfd` and `newfd` have the same value, then `dup2()` does nothing—`newfd` is not closed, and `dup2()` returns the `newfd`

dup3() System call

```
int dup3(int oldfd, int newfd, int flags);
```

- The dup3() system call performs the same task as dup2(), but adds an additional argument, flags, that is a bit mask that modifies the behavior of the system call
- At the time of this writing, dup3() supports one flag, O_CLOEXEC, which causes the kernel to enable the close-on-exec flag (FD_CLOEXEC) for the new file descriptor
- When a file descriptor is opened (as with open or dup), this bit is initially cleared on the new file descriptor, meaning that descriptor will survive into the new program after exec
- The dup3() system call is new in Linux 2.6.27, and is Linux-specific

Examples: Input Redirection

Method 1: `close-open` `(stdinredir1.c)`

```
close(0);
fd = open("/etc/passwd", O_RDONLY);
```

Method 2: `open-close-dup-close` `(stdinredir2.c)`

```
fd = open("/etc/passwd", O_RDONLY);
close(0);
newfd = dup(fd);
close(fd);
```

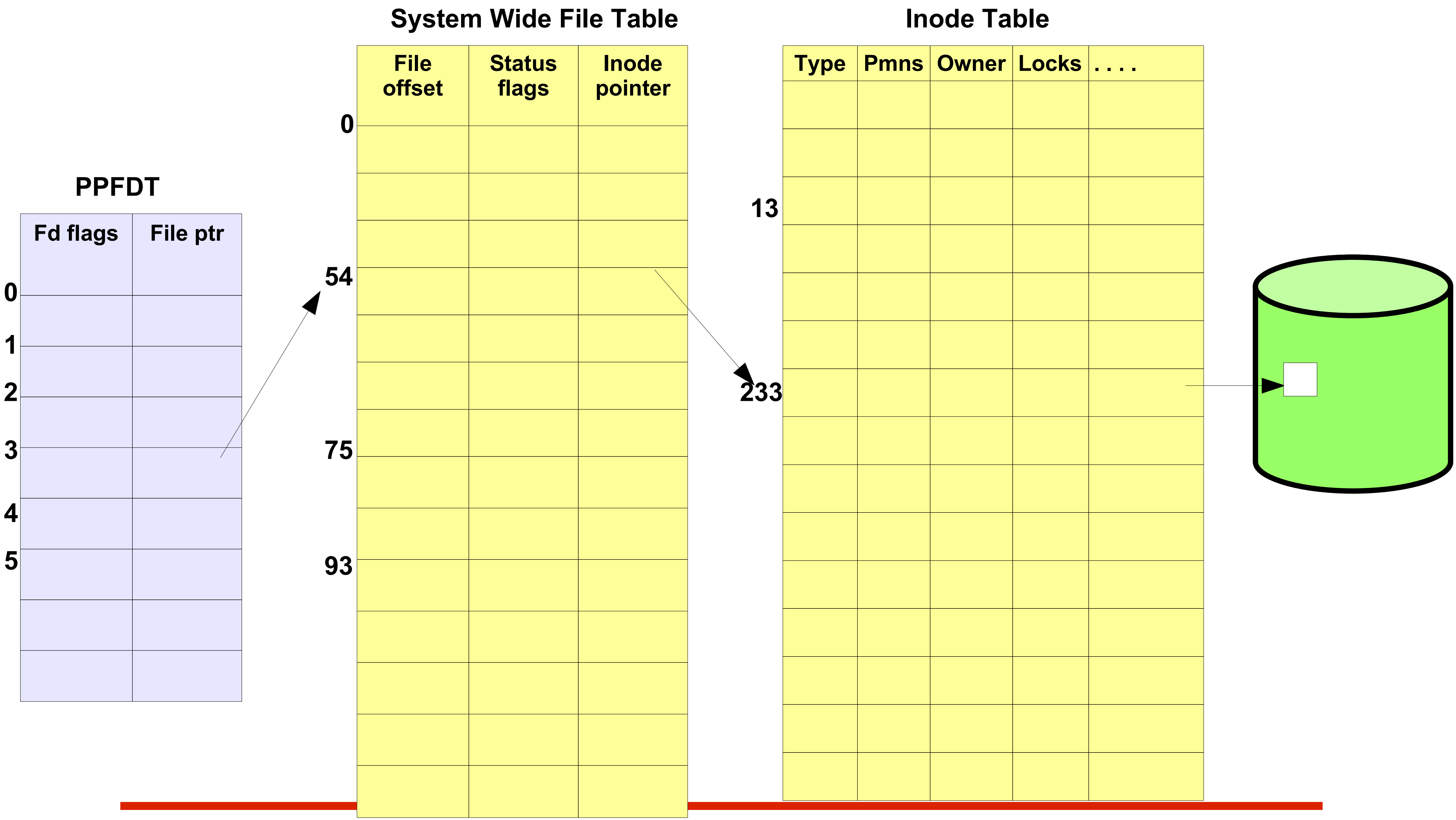
Method 3: `open-dup2-close` `(stdinredir3.c)`

```
fd = open("/etc/passwd", O_RDONLY);
newfd = dup2(fd, 0);
close(fd);
```

`fcntl()` System Call

What `fcntl()` can do?

```
int fcntl(int fd,int cmd, long arg);
```



System Wide File Table

Inode Table

PPFDT

fcntl() (Duplicate file descriptor)

```
int fcntl(int fd,int cmd, long arg);
```

The `fcntl()` system call can be used instead of `dup()` to return a duplicate file descriptor of an already opened file. The second argument passed to `fcntl()` for this purpose is **F_DUPFD**. It will return the lowest-numbered available file descriptor greater than or equal to the third argument

```
int fd = open("/etc/passwd", O_RDONLY);
printf("The first file descriptor is %d\n",fd);
int rv = fcntl(fd, F_DUPFD, 54);
printf("Duplicate file descriptor is %d\n",rv);
```

Example: `fcntl_dup.c`

fcntl() (Get file status flags)

```
int fcntl(int fd,int cmd, long arg);
```

The `fcntl()` system call can be used to get the file status flags of an already opened file from SWFT. For example suppose you have opened a file and want to check the file access mode flags (O_RDONLY, O_WRONLY, O_RDWR). The second argument passed to `fcntl()` for this purpose is **F_GETFL** and the third argument is ignored. It will return all the file status flags in an integer variable which when bitwise anded with the O_ACCMODE constant will tell you about the permissions. The constants can be found in `/usr/include/asm-generic/fcntl.h`

```
int fd = open(“file”, O_RDONLY);
int flags = fcntl(fd, F_GETFL, 0);
flags = flags & O_ACCMODE;
if (flags == O_RDONLY) printf("read only\n");
```

Example: `fcntl_checkpmns.c`

fcntl() (Set file operating mode flags)

```
int fcntl(int fd,int cmd, long arg);
```

O_APPEND flag is used to ensure that each call to `write()` implicitly includes an `lseek` to the end of the file. Moreover, the kernel combines `lseek()` and `write()` into an atomic operation. Suppose you forgot to set this flag while making the `open()` call. Now if you have already opened a file and want to set O_APPEND flag, you can do that with `fcntl()` system call with a simple three-step procedure:

```
int flags = fcntl(fd, F_GETFL, 0); //get settings
flags = flags | O_APPEND;           //modify settings
fcntl(fd, F_SETFL, flags);          //set them back
```

fcntl() (Set file operating mode flags)

```
int fcntl(int fd,int cmd, long arg);
```

O_SYNC flag is used to turn off disk buffering. It tells the kernel that call to `write()` should return only when the bytes are written to the actual hardware rather than the default action of returning when the bytes are copied to a kernel buffer. However, setting O_SYNC eliminates all the efficiency kernel buffering provides. Suppose you want to set this flag, but forgot to set it while making the `open()` call. Now if you have already opened a file and want to turn off Kernel disk buffering, you can do that with `fcntl()` system call with a simple three-step procedure:

```
int flags = fcntl(fd, F_GETFL, 0); //get settings
flags = flags | O_SYNC;            //modify settings
fcntl(fd, F_SETFL, flags);         //set them back
```

File / Record Locking

Types of Locking Mechanisms:

- **Advisory locks:** Kernel maintains knowledge of all files that have been locked by a process. But it does not prevent a process from modifying that file. The other process can, however, check before modifying that the file is locked by some other process. Thus advisory locks require proper coordination between the processes
- **Mandatory Locks:** are strict implications, in which the kernel checks every read and write request to verify that the operation does not interfere with a lock held by a process. Locking in most UNIX machines is by default advisory. Mandatory locks are also supported but it needs special configuration

Types of Advisory Locks:

- **Read Locks/Shared Locks:** Locks in which you can read, but if you want to write you'll have to wait for everyone to finish reading. Multiple read locks can co-exist
- **Write Locks/Exclusive Locks:** Locks in which there is a single writer. Everyone else has to wait for doing anything else (reading or writing). Only one write lock can exist at a time

fcntl() (File/Record Locking)

```
int fcntl(int fd,int cmd, struct flock* lock);
```

- The fcntl() system call can be used for achieving read/write locks on a complete file or part of a file
- To lock a file the second argument to fcntl() should be F_SETLK for a non-blocking call, or F_SETLKW for a blocking call
- The third argument to fcntl() is a pointer to a variable of type struct flock (See its details in man page)
- Locks acquired using fcntl() are not inherited across fork(). But are preserved across execve()

Example: fcntl_lock.c

Things To Do

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