CMP325
Operating Systems
Lecture 28

File System Architecture

Muhammad Arif Butt, PhD

Note:

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

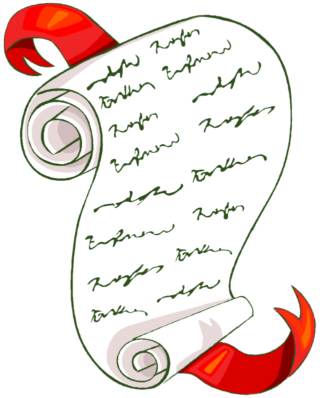
OS with Linux:

https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

System Programming:

https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today’s Agenda



- Seven File Types in UNIX
- Concept of Files and Directories
- Directory Structures
  - Single Level Directory Structure
  - Two Level Directory Structure
  - Tree Directory Structure
  - Acyclic Graph Directory Structure
  - General Graph Directory Structure
- Keeping Track of File’s Data Blocks
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation Technique
  - UNIX Allocation Technique
- Keeping Track of Free Data Blocks
  - Bit Vector
  - Linked List
  - Grouping
  - Counting
- File System Architecture
  - Creating a File
  - Accessing a File
  - File Descriptor to File Contents
  - Introduction
  - Disk Scheduling
- File Sharing and Links in UNIX

Concept of Files and Directories

FILE TYPES IN UNIX

- **Regular file (-)** Files that contain information entered in them by a user, an application program or a system utility program.
- **Directory (d)** Contains a list of file names plus pointers to associated i-nodes. Directories are actually ordinary files with special write protection privileges so only the file system can write into them, while read access is available to user programs.
- **Symbolic Link (l)** Links let you give a file more than one name.
- **Block Special File (b)** A block special file consists of as sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed, i.e. we can directly access block 154 without first having to read blocks 0 to 153. Block special files are typically used for disks. E.g. /dev/hda1, /dev/lp.
- **Character Special File (c)** Used to communicate with h/w that input or output one character at a time. Keyboard, printers, mice, plotters, networks are examples of character special files.
- **Named Pipe (p)** A file that passes data between processes. It stores no data itself, but passes data between process writing data into pipe and process reading from pipe. ls –l /dev | less
- **Socket (s)** A stylized mechanism for inter-process communications

FILE NAMING

- When a process creates a file it gives the file a name. When the processes terminates the file continues to exist and can be accessed by other processes using its name.
- The naming rules vary from system to system. Most OS allow strings of one to eight characters as legal file name allowing digits and some special characters. Some File Systems distinguish between upper and lower case letters while others do not.
- WINDOWS
  - File names up to 255 characters
  - Not case sensitive. File1, file1 and FILE1 all refer to same file.
  - Aware of file extensions, when a user double clicks on a file name , the program assigned to this file extension is launched with the file as parameter.
- UNIX
  - File names up to 255 characters, all acceptable except '/'.
  - Case sensitive. File1, file1 and FILE1 refer to three different files.
  - File extensions are just conventions and are not enforced by the OS.

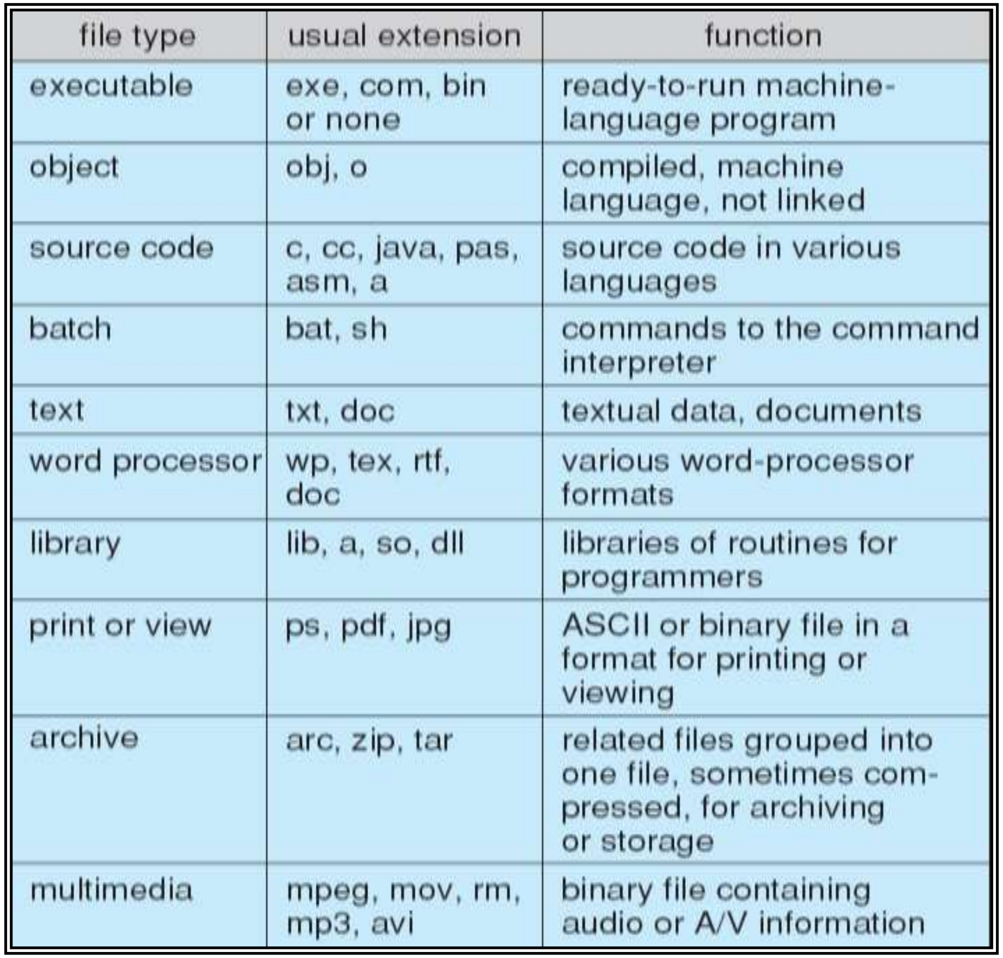
FILE ATTRIBUTES

- Every file has a name and its data.
- In addition, all OS associate certain other information with each file, we call these extra items the file's attributes. List of attributes varies from system to system.

- Basic Information
  - File Name.
  - File Type.
  - File Organization.
- Address Information
  - Starting address.
  - Size used.
- Access Information
  - Owner.
  - Access List.
  - Permitted Actions.
- Usage Information.
  - Date of Creation.
  - Date of last read access.
  - ID of last reader.
  - Date of last update access.
  - ID of last modifier.
  - File current usage.

COMMON FILE TYPES

- A common technique for implementing file types is to include the type as part of the file name – name and an extension as used in Windows.
- UNIX recognizes no file types, rather it uses a magic number stored at the beginning of a file. Magic number identifies the file as an executable file to prevent the accidental execution of a file not in this format.



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

- UNIX does not record the name of the creating program, either.
- It does allow file name extension , but these are not enforced by OS, rather are mostly to aid users in determining the type of contents of the file.

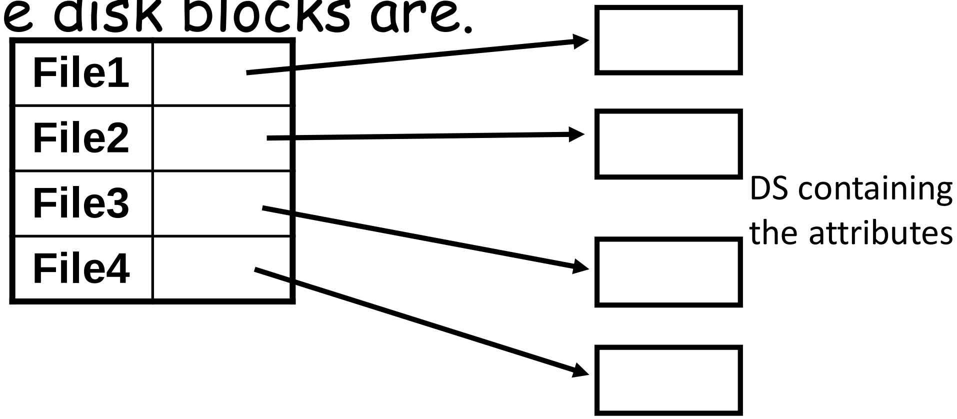
INTRODUCTION TO DIRECTORIES

- Systems store millions of files on terabytes of disk. To manage such a huge amount of data we need to organize them. This organization can be done by partitioning the disk and by the concept of directory:
- **Device Directory** - **The main function of directory system is to map the ASCII name of the file onto the information needed to locate the data.** A directory is a collection of directory entries. Each directory entry contains information like name, location, size and type for all files within that directory. It provides mapping between file names and applications. It's a special file owned by OS and accessible by file management system. In UNIX a directory is a file that holds the inode numbers and names of files in it.

IMPLEMENTING DIRECTORIES

- The information about the files within a directory can be stored directly in a directory entry as discussed on previous slides. In this simple design, a directory consists of a list of fixed size entries, one per file, containing a fixed length file name, a structure for the file attributes and one or more disk addresses telling where the disk blocks are. This approach is used in MS-DOS / Windows.
- For systems that uses i-nodes (UNIX), another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. The i-node number points to a Data Structure (a specific entry in the i-node table) that contains the file attributes and one or more disk addresses telling where the disk blocks are.

File1	**Attributes**
File2	**Attributes**
File3	**Attributes**
File4	**attributes**



WHERE FILE ATTRIBUTES ARE STORED

- In DOS / Windows the file attributes are stored in the Directory structure as part of the Directory entry for the file.

File Name	Type	— — —	FAT Pointer

- In UNIX a separate data structure is used for this purpose known as **Inode Tables**. However some information is kept inside the directory structure like file names and inode numbers for the file.

File Name	Inode #

Overview of Directory Structures

DIRECTORY STRUCTURE

While designing directory structure consider following:

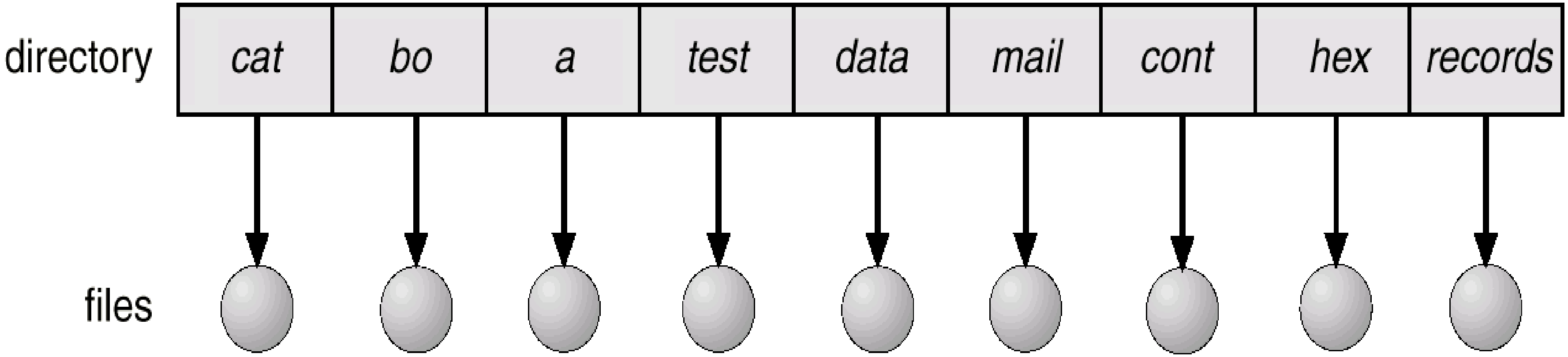
- Efficient searching.
- Naming – convenient to users.
  - Two or more users can have same name for different files.
  - One file can have several different names (links) in different directories.
- Grouping – logical grouping of files by properties, (e.g., all C++ programs, all java programs, all music files, all games, …)

Possible structures are:

- Single Level Directory.
- Two Level Directory.
- Tree Structured Directory.
- Acyclic Graph Directory.
- General Graph Directory.

SINGLE LEVEL DIRECTORY

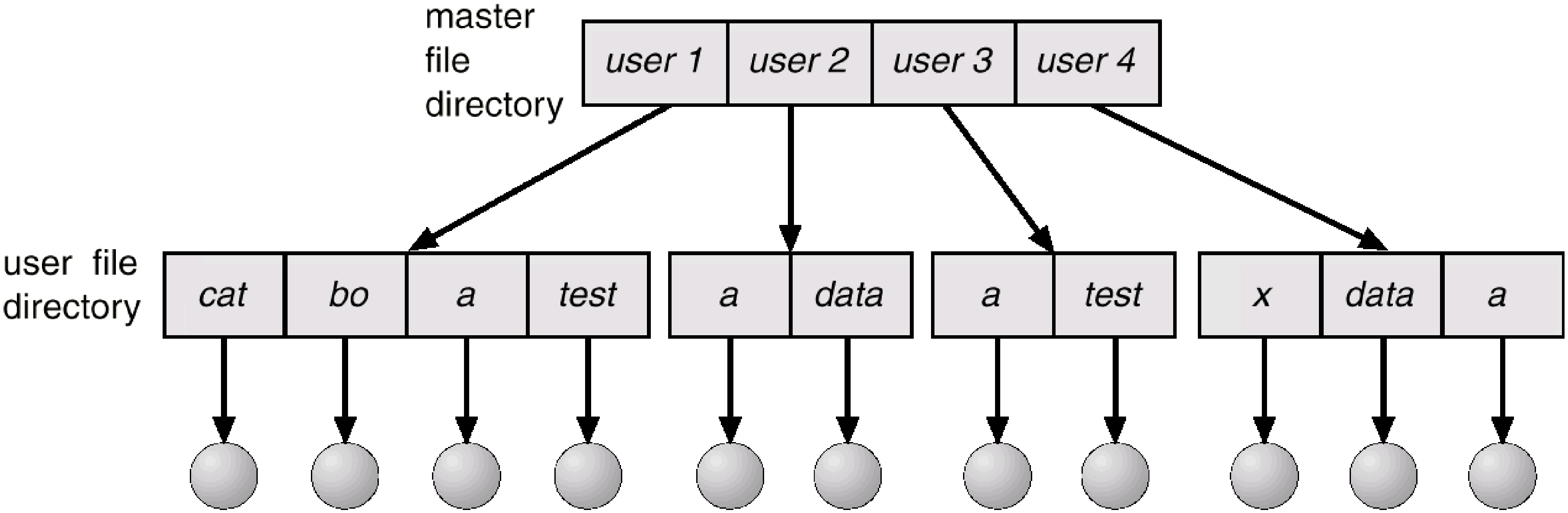
- A single directory for all users.



- Naming problem
- Grouping problem

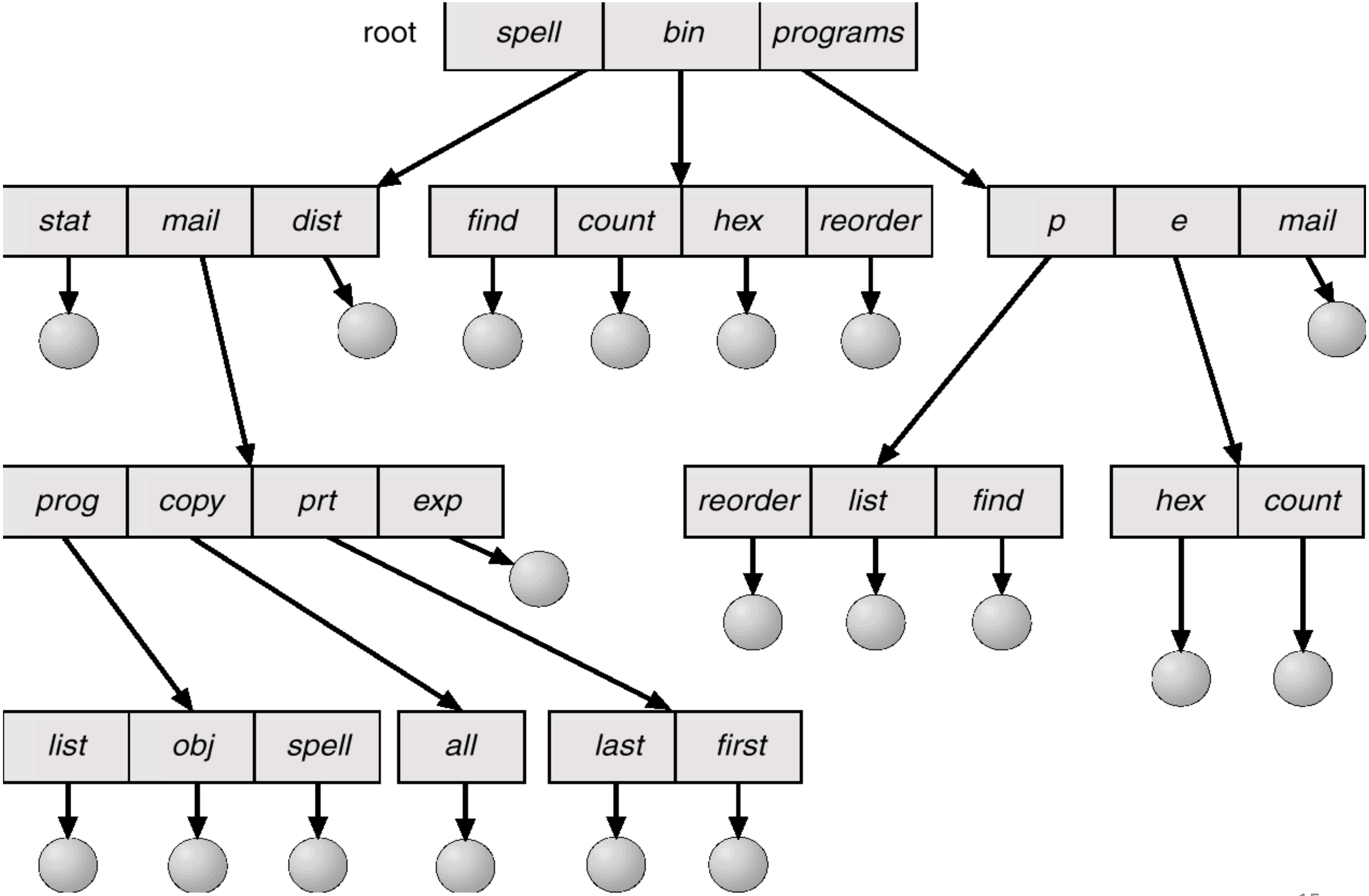
TWO LEVEL DIRECTORY

- Separate directory for each user.



- Path name
- Can have the same file name for different user, because each user has a separate directory.
- Efficient searching
- No grouping capability - A user cannot have separate directories for Linux, Data comm., music files, etc

TREE STRUCTURED DIRECTORIES

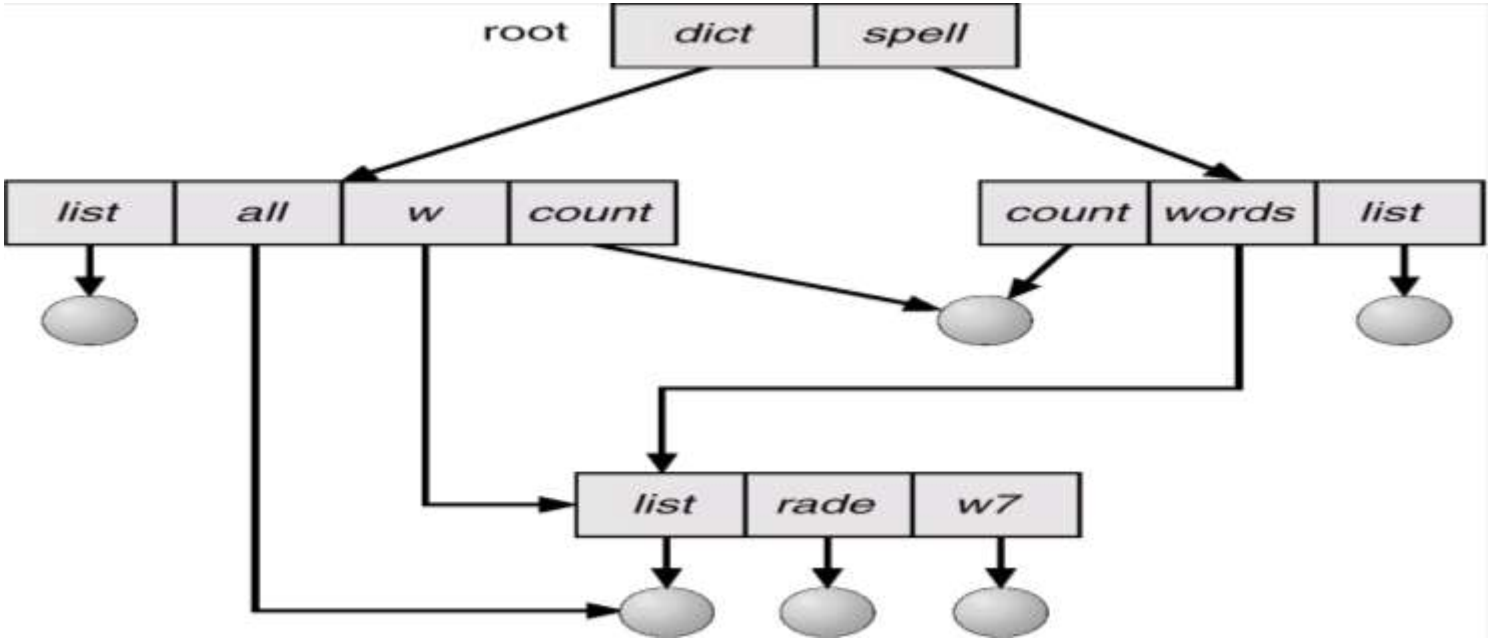


TREE STRUCTURED DIRECTORIES (cont…)

- **Efficient searching**
- **Grouping Capability -** Allows users to create their own sub directories and to organize their files accordingly.
- **Current directory (working directory) -** Each user has a current working directory. When reference is made to a file, the current directory is searched. If a file is not there in the current directory, then the user either change the directory or specify a path name, e.g. cd /spell/mail/prog/
- **Path Names**
  - Absolute Path is a complete road map starting from the root directory down to the specified file, giving directory names on the path.
  - Relative Path defines a path from the current directory.
- **Creating / Deleting files / directories**
  - Creating a new file is done in current directory. touch <file name>
  - Delete a file rm <file-name>
  - Creating a new subdirectory is done in current directory. mkdir <dir-name>
  - Deleting “mail” ⇒ deleting the entire sub tree rooted by “mail”. rm –r <dir name>

ACYCLIC GRAPH DIRECTORIES

- A tree structure prohibits the sharing of files / directories. An acyclic graph allows directories to have shared subdirectories and files. In other words, different pathnames can be used to access a file.
- Any changes made by one person in a file are immediately visible to the other.
- Similarly in shared subdirectories, a new file created by one person will automatically appear in all the shared subdirectories.
- Shared files can be implemented using:
  - Links.
  - Duplicating Directory entries.



ACYCLIC GRAPH DIRECTORIES (cont…)

Problems

- A file may have multiple absolute paths. If we are trying to traverse entire file system we will be visiting the shared files / directories more than once e.g. count file is there in dict directory as well as in spell directory.
- If count inside the dict directory is deleted and also the file, then the count inside the spell directory will point to some thing that doesn't exits. (dangling pointer)

Solutions

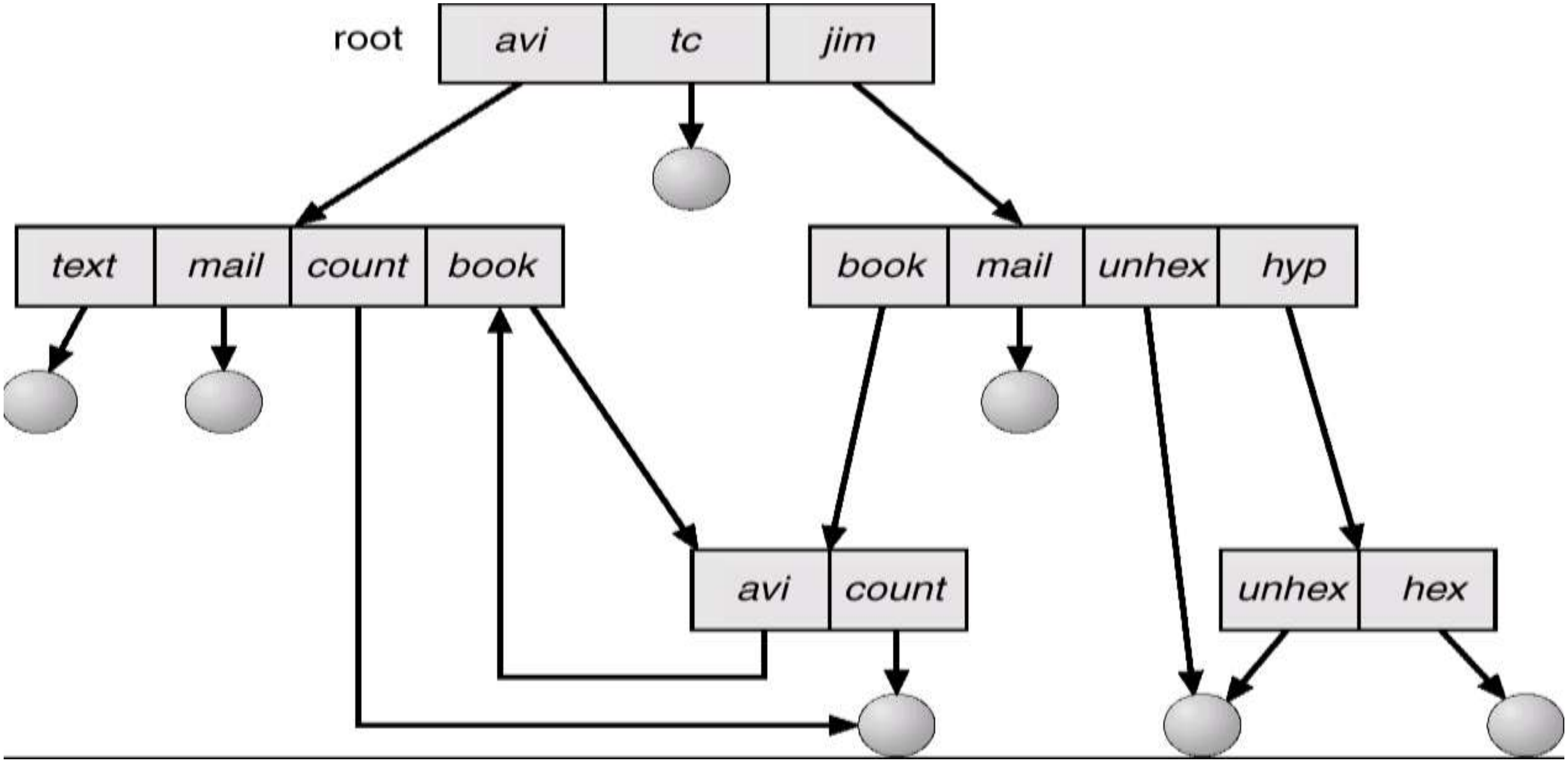
- **Back pointers -** While deleting a file that is pointed to from many locations use back pointers to delete all the pointers to this file and then delete the file so after deleting we do not have any dangling pointer.
- **Link count -** Some systems like Linux maintain link count. When a user deletes a file its link count is decremented by one. If it becomes zero that means the file is not referenced by any other pointer so the file is deleted else file is not deleted only the directory entry is deleted and link count is decremented.

GENERAL GRAPH DIRECTORY

- General Graph directory structure is not necessarily acyclic, i.e. it may contains cycles.

  book -> avi ->book

- When you will traverse a directory tree or back up a directory tree you may get into a cycle.



GENERAL GRAPH DIRECTORY (cont…)

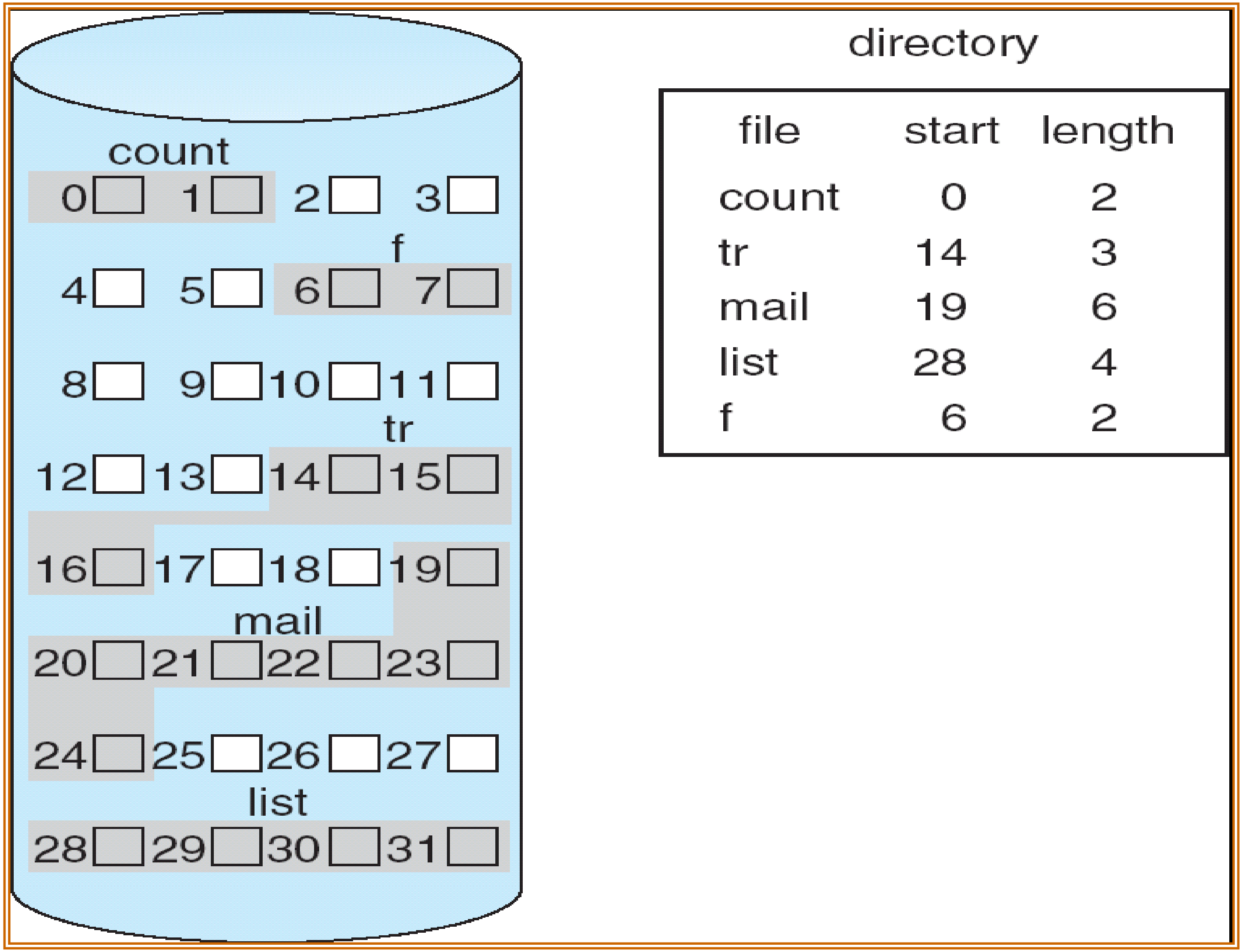
- How do we guarantee no cycles?
  - Allow only links to files and not to subdirectories.
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK.
  - Garbage collection. If at all cycles are created or links exists that does not refer to files, use garbage collector. (A utility used in Java also)

Keeping Track of File's Data Blocks

CONTIGUOUS ALLOCAITON

- One of the most important issues in implementing file storage is keeping track of which disk blocks go with which file. Various methods (contiguous allocation, Linked list allocation, Indexed allocation) are used in different operating systems.
- In contiguous allocation each file occupies a set of contiguous blocks on the disk.
- Directory entry contains starting block # and file size (number of blocks). E.g. if 1st block# is 15 and the file size is 6 blocks so 15 – 20 (both inclusive), then the directory entry will contain (15,6) for that particular file.
- Merits:
  - Good sequential and Random access. E.g. if you want to go 20 Bytes ahead just add 20 and get there.
- Demerits.
  - Wasteful of space / External Fragmentation (dynamic storage-allocation problem). Here instead of compaction we use de-fragmentation.
  - User has to declare file size before creating it.
  - Expensive file growth, i.e. if a file is of 100 blocks it doesn't have a free block ahead or at its tail and you need to add a block to it. You have to move the entire file to another location.
- Contiguous allocation is still used in some old systems like PIC.

CONTIGUOUS ALLOCATION (cont…)



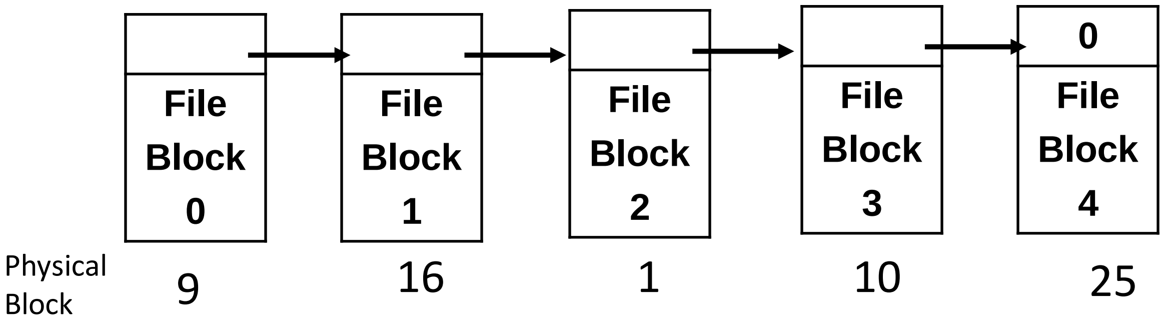
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space

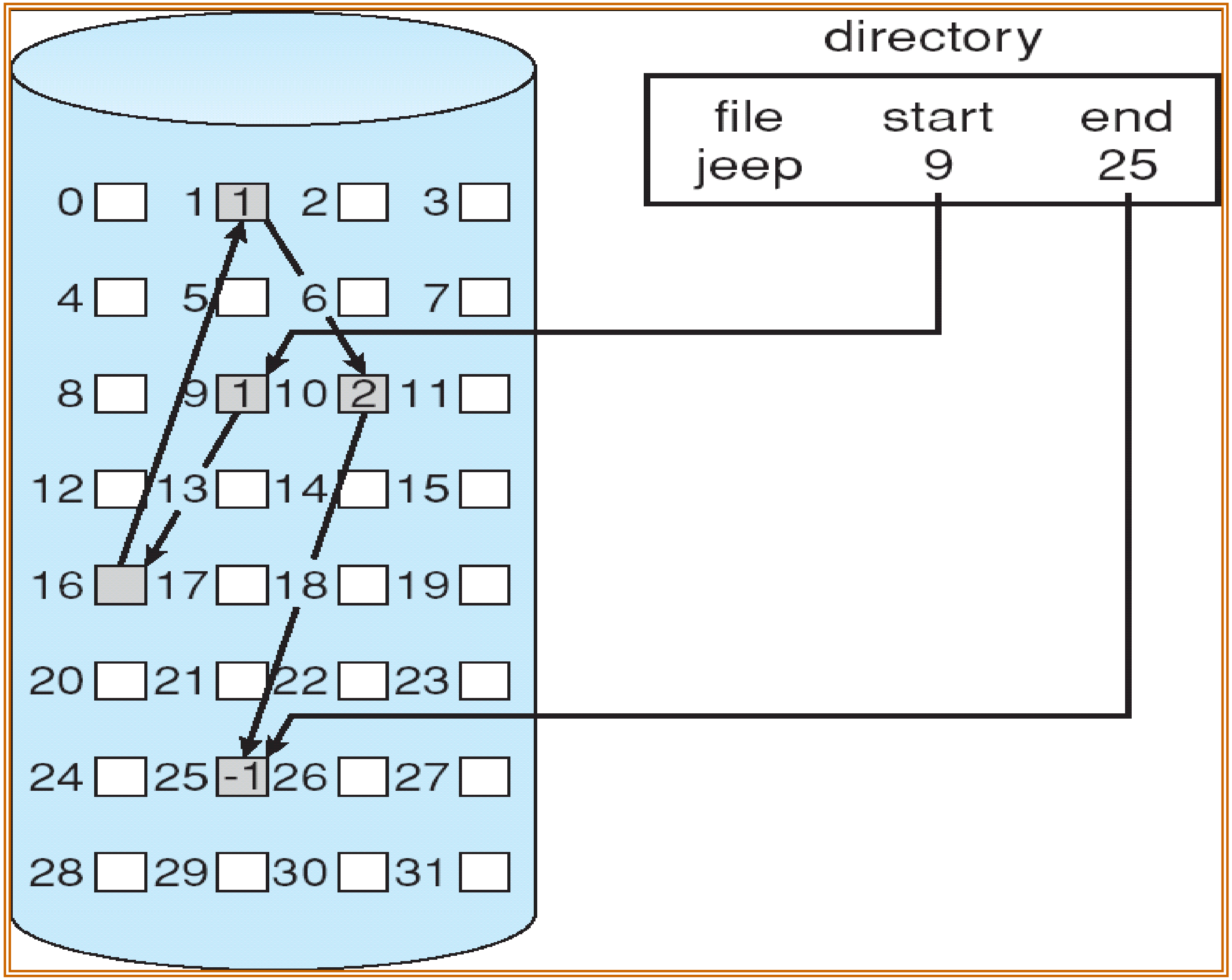
LINKED ALLOCATION

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk. The first word of each block is used as a pointer to the next one. The rest of the block is for data.
- Unlike contiguous allocation every disk block can be used in this method and no space is lost due to disk fragmentation (no external fragmentation). However internal fragmentation exist in the last block.
- Reading a file sequentially is straightforward, random access is extremely slow. Its like a link list, e.g. if the file size is of 100 blocks and you want to add a block after the 50th block. You have to traverse fifty nodes and then readjust the pointers of the link list .



- In the figure a file is shown that uses disk blocks 9, 16, 1, 10 and 25 in that order. Block 9 is the starting block and block 25 is the last block of the file. The above information can be kept in a table called File Allocation Table.

LINKED ALLOCATION (cont…)



Why the end pointer is kept?

LINKED ALLOCATION (cont…)

Why the end Pointer is kept?

- If you want to append at the end of a file, in absence of end pointer you will have to traverse entire link list from the start pointer onwards. Traversing link which are on disk blocks is of course very expensive.
- For Example. Consider a file of 100 blocks, and you want to append at the end of this file. Hit the last block using end pointer directly. Allocate a new block and place its address in the previous end block and make the new block the end block and make its entry in the directory.
- Merits:
  - Can grow files dynamically
- Demerits.
  - Bad sequential access (a seek required between each block)
  - Very bad random access
  - Lose one block, lose rest of the file

LINKED ALLOCATION (cont…)

File Allocation Table

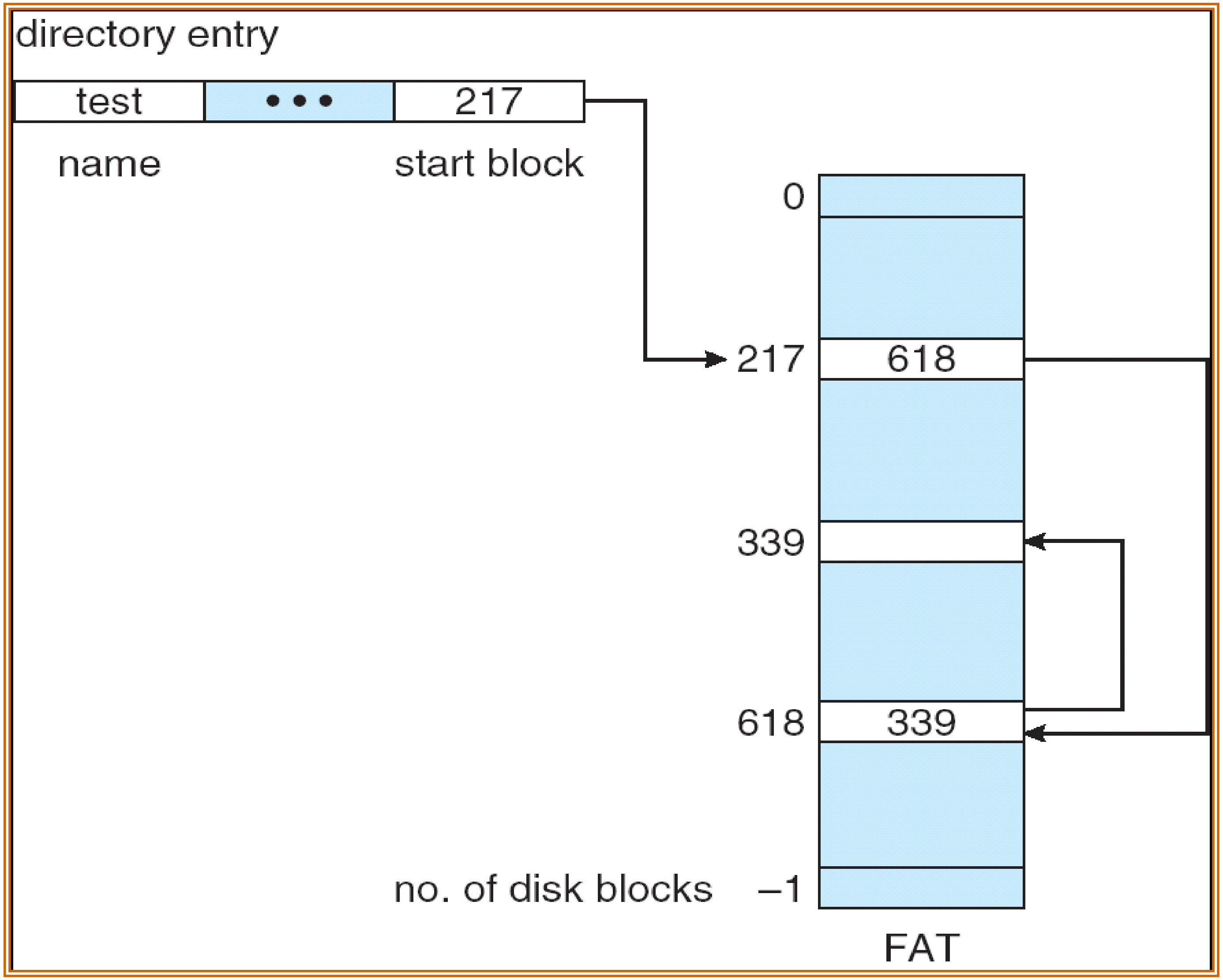
- *File-allocation table (FAT)* uses linked allocation.
- MS-DOS and OS/2 uses this scheme.
- Links not in pages, but in FAT
  - FAT contains an entry for each block on that disk
  - FAT entries corresponding to blocks of a file are linked together
- Access Properties
  - Sequential access expensive unless FAT cached in memory
  - Random access expensive always, but really expensive if FAT not cached in memory
- The primary advantage is traversing among the blocks is efficient. Although the chain must still be followed to find a block within a file but since the chain is entirely in memory so it can be followed without making any disk references.
- The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.

Physical Block	
0	
1	**10**
9	**16**
10	**25**
16	**1**
25	**-1**

- With 20 GB disk and a 1 KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. The table can be paged in a paged system to conserve memory but still it will occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

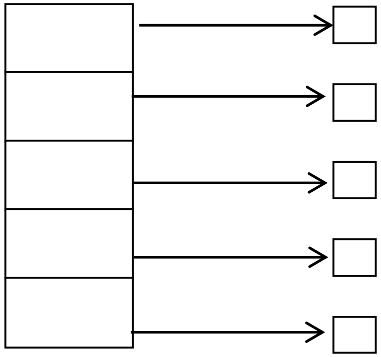
LINKED ALLOCATION (cont…)

File Allocation Table

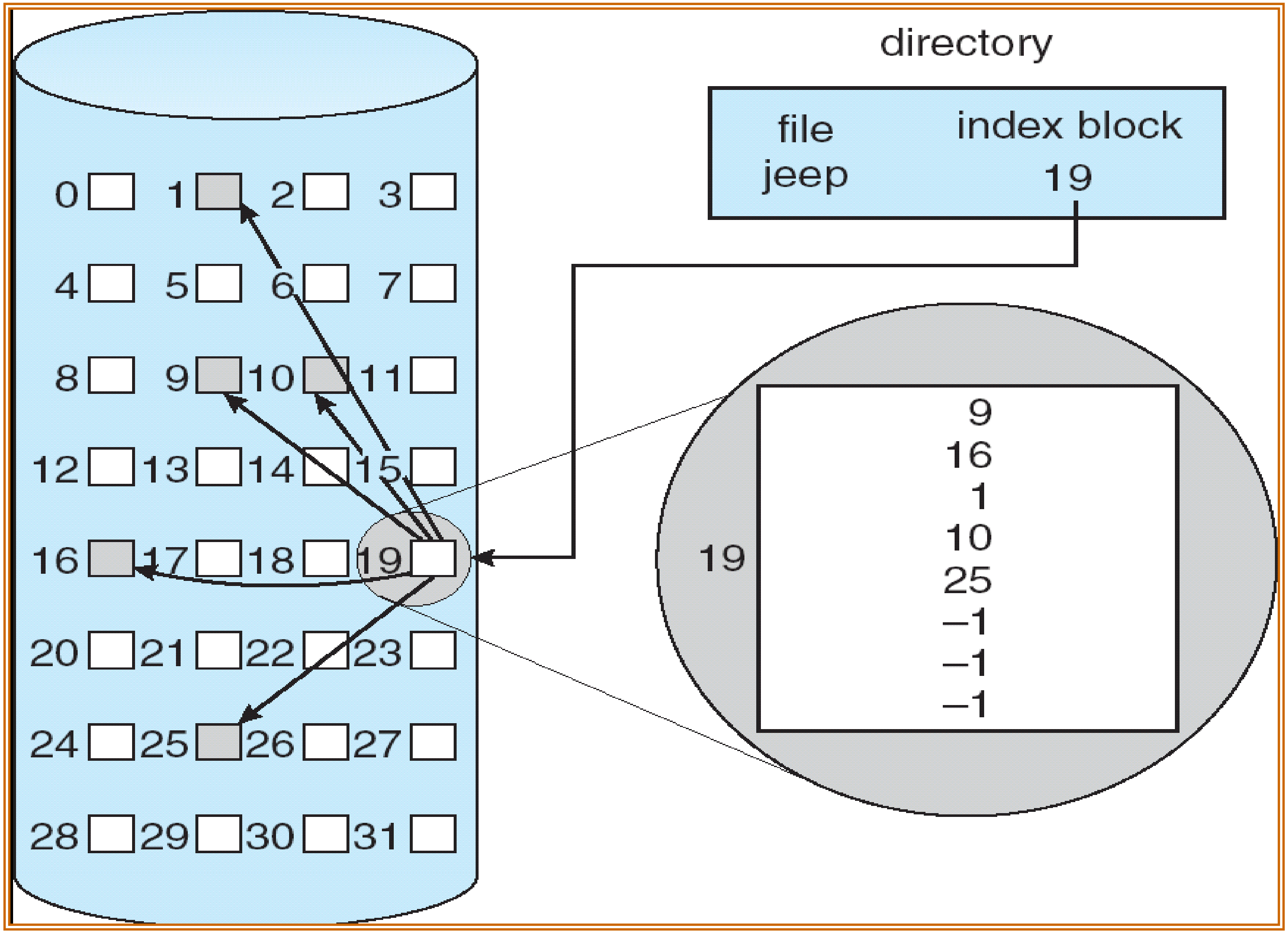


INDEXED ALLOCATION

- Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index node).
- Pointers of all file blocks are kept at one place called index table. Logical view is shown below.
- In the figure on the next slide you see block # 19 is the index block or index table. The directory entry gives you the block # of the index block. Inside the index block you get the pointers of block #s of all the blocks where file's data resides.
- Supports sequential access. You can traverse sequentially over the blocks as they are there in the index block.
- Supports random access, i.e. if you want to go to 6th block directly you can go easily by getting the 6th entry of index block.
- Suppose we can store 512 pointers in a block. If file size grows more than 512 blocks then the index block can't come in one block. For that we need more than one index block.
- To link more than one index block for a file we have two ways:
  - Linked Scheme.
  - Multilevel Index Tables.
  - Combined Scheme.


index table

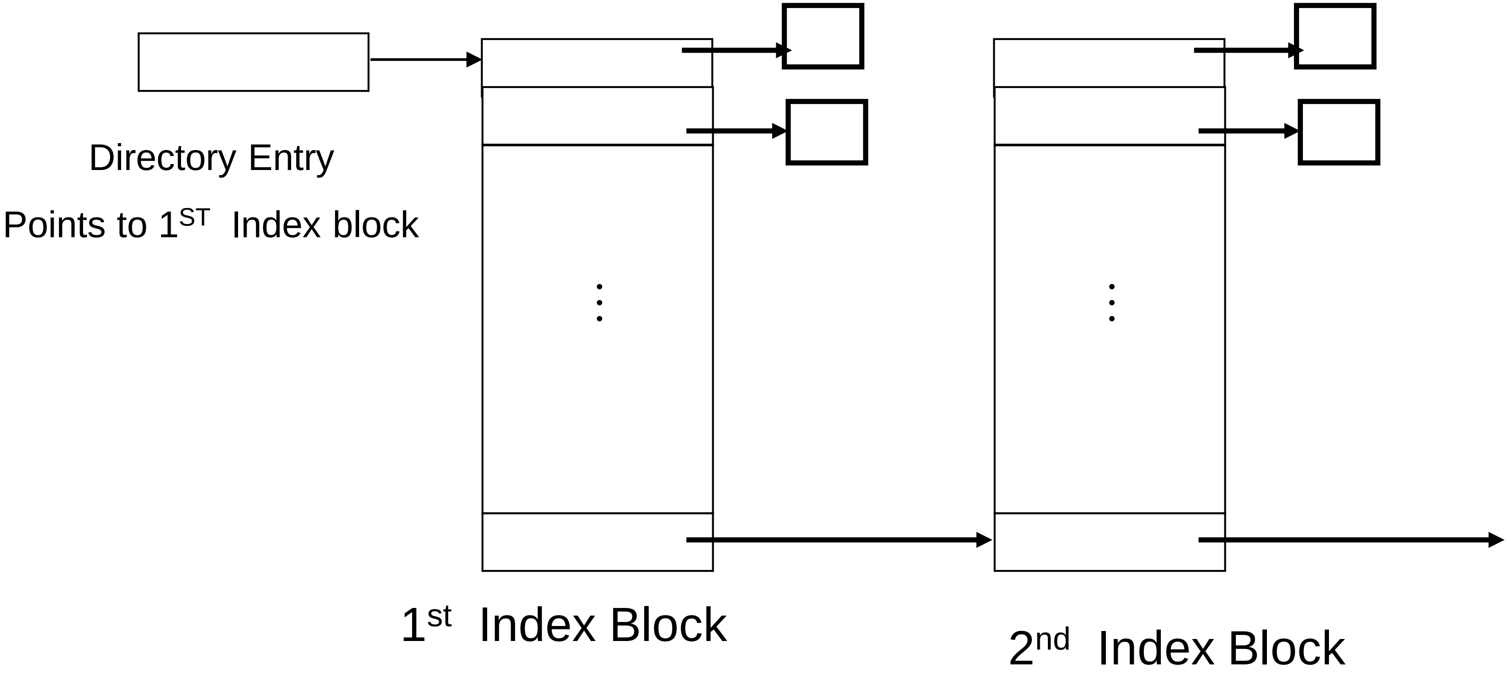
INDEXED ALLOCATION (cont…)



INDEXED ALLOCATION (cont...)

LINKED SCHEME

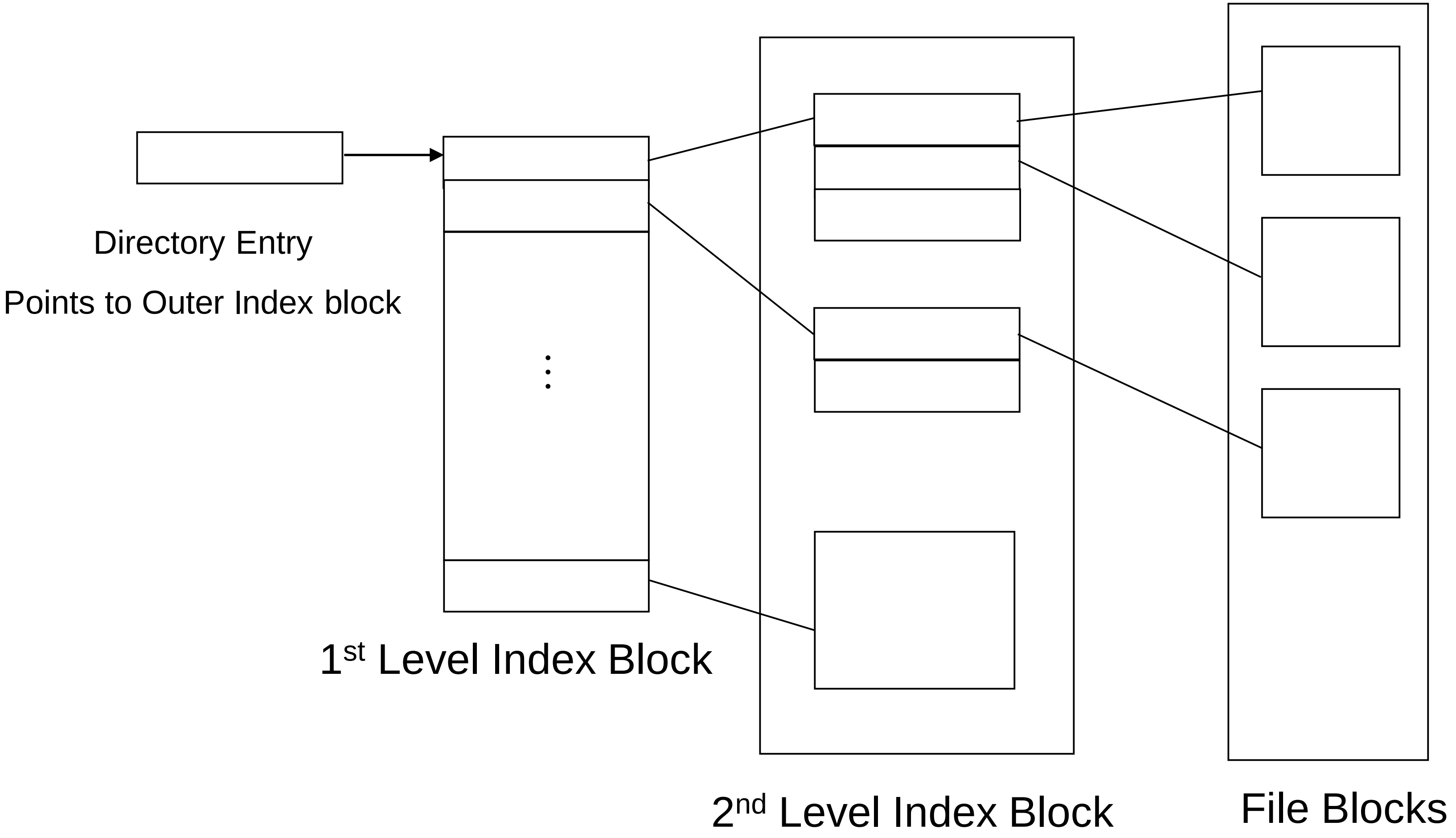
- It’s a linked list of index blocks.
- Suppose each block is of size 512 Bytes.
- First 511 pointers of every index block will point to data blocks of file and last pointer will point to next index table.



INDEXED ALLOCATION (cont...)

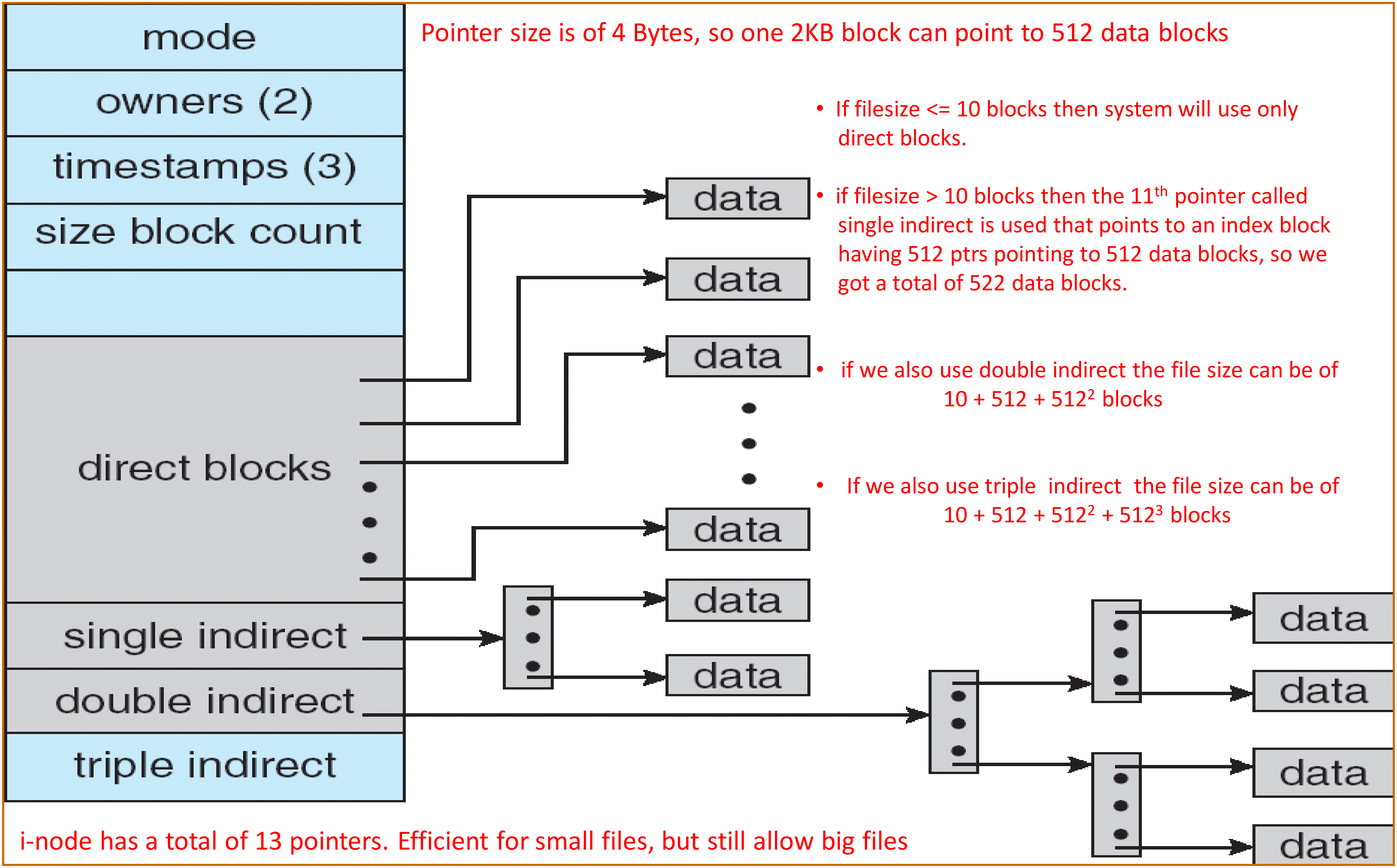
TWO LEVEL INDEX TABLE

- 512 pointers of 1st level index table points to 512 inner level index tables.
- So there are 512 2nd level index tables each pointing to 512 disk blocks.
- So we can manage a file size of 512 X 512 blocks using two level index table.



COMBINED SCHEME - UNIX (2K bytes per block)

Pointer size is of 4 Bytes, so one 2KB block can point to 512 data blocks



- If filesize <= 10 blocks then system will use only direct blocks.
- if filesize > 10 blocks then the 11th pointer called single indirect is used that points to an index block having 512 ptrs pointing to 512 data blocks, so we got a total of 522 data blocks.
- if we also use double indirect the file size can be of $10 + 512 + 512^2$ blocks
- If we also use triple indirect the file size can be of $10 + 512 + 512^2 + 512^3$ blocks

i-node has a total of 13 pointers. Efficient for small files, but still allow big files

EXAMPLE

- Consider a UNIX system with 4 KB disk Block and size of disk pointer is 4 Bytes. If the system uses only 10 direct pointers, what will be the size of the file? What can be the maximum file size? What is the amount of space needed to store pointers?

Keeping Track of Free Data Blocks

BLOCK SIZE

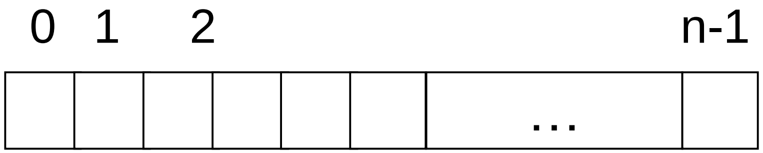
- Files are normally stored on disk. Two general strategies are possible for storing an n byte file:
  - n consecutive bytes of disk space are allocated.
  - File is split up into fixed size blocks that need not to be adjacent.
- The size of a block can be equal to a sector, track, cylinder or even a page.
- Having a large allocation unit, like a cylinder means that every file even a 1 Byte file will occupy a space of entire cylinder.
- Having a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.
- Small blocks are bad for performance but good for disk space utilization.
- As per a study on UNIX system most of the files are of 2 KB size. So keeping a 2 KB block size, it is expected to have a less number of space wasted.
- In reality, a very few files are a multiple of the disk block size, so some space is always wasted in the last block of a file.
- Once a block size has been chosen, the next issue is how to keep track of free blocks, that is discussed on next slides.

FREE SPACE MANAGEMENT

- To keep track of free disk space, the system maintains a free space list.
- When a new file is created (in a free space), that free space is removed from free space list.
- When a file is deleted, its disk space is added to the free space list.
- There are different methods of Free space management that are discussed on next slides:
  - Bit Vector / Bit Map.
  - Linked List.
  - Grouping.
  - Counting.

FREE SPACE MANAGEMENT - BIT VECTOR

- For every free block we allocate a bit. A disk with n blocks requires a bitmap with n bits. So a n bit vector is required.



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- To check the status of block #154, index the bit vector with 154.
- Overhead to maintain bitmap:
  - Block size is 4 KB. Disk Size is 40 GB. Then the overhead for bitmap will be $40 \times 2^{30} / 2^{12} = 40 \times 2^{18}$ bits. So the bit vector size will be 1280 KB.
  - Bit map require less space, i.e. one bit per block

FREE SPACE MANAGEMENT – BIT VECTOR (cont...)

- Bit map must be kept on disk
  - Copy in memory and disk may differ.
  - Cannot allow for block[*i*] to have a situation where bit[*i*] = 0 in memory and bit[*i*] = 1 on disk.
- Solution:
  - Set bit[*i*] = 0 in disk.
  - Allocate block[*i*]
  - Set bit[*i*] = 0 in memory
- Block Number Calculation

  Sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of block number is shown below:

(number of bits per word) * (number of zero valued words) + offset of the first 1 bit

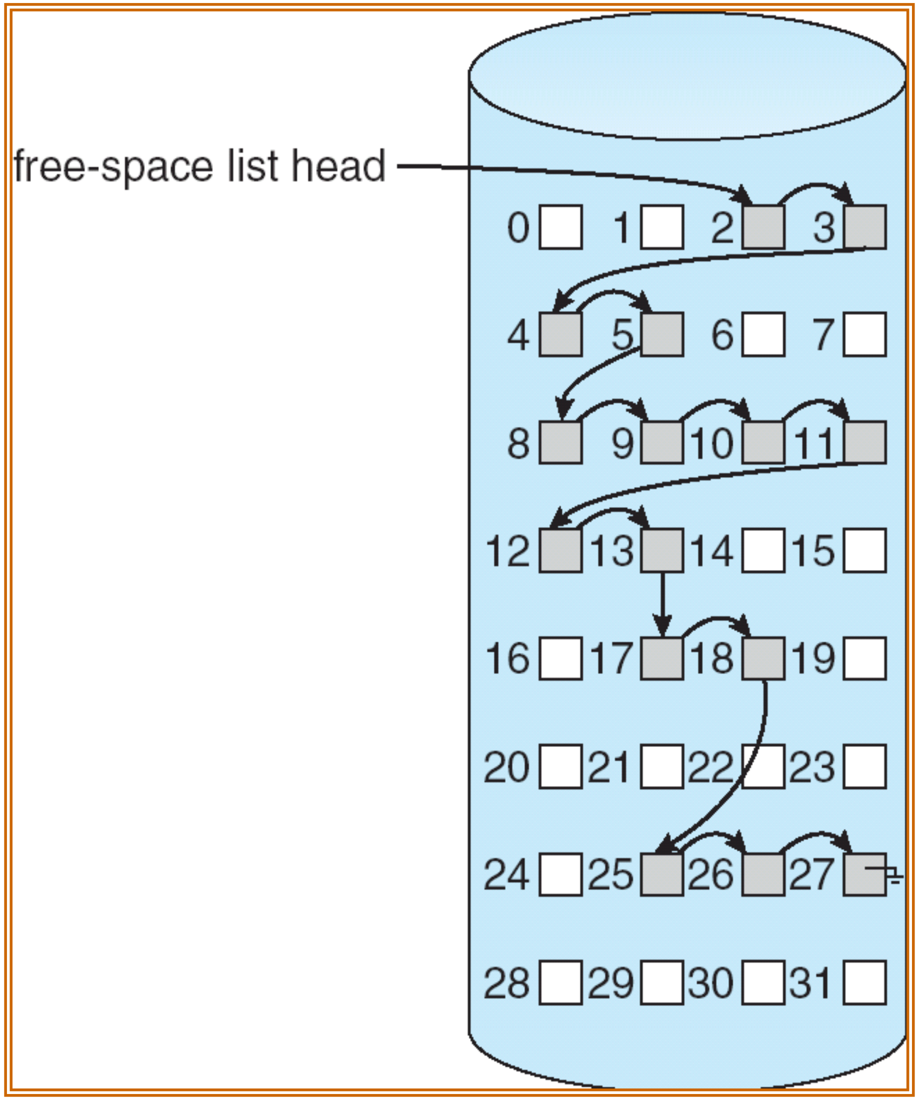
- Example
  - Consider a disk where block 2, 3, 4, 5, 8, 9, 10, 11, 12, 17 are free and rest are allocated. The free space bit map would be

0011110011111000010000000000……

FREE SPACE MANAGEMENT – Linked List

Linked List / Free List

- Another approach to free space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- Maintain a link list of free blocks, i.e. first block contains a pointer to the next free disk block and so on.
- Cannot get contiguous space easily, so its not good for contiguous allocation.
- No waste of space, overhead is the number of pointers used to maintain free space. (For every block there will be a pointer to another disk block)



FREE SPACE MANAGEMENT

Grouping

- A variation of linked list. Instead of having a link list on disk, we collect all the pointers and place them in one block. Just like index block. The difference is the pointers in index block points to data blocks while here the pointers will point to free blocks.
- First (n-1) pointers points to free blocks and last pointer points to another such block.
- Problem. How many pointers can be stored in one block. If total number of pointers for free blocks can't be accommodated in one block, we can use multilevel indexing technique.

Counting

- Another variation of linked list. An entry of index block contains two fields: address of the first free block and number of free contiguous blocks that follow the first block- good for contiguous allocation e.g.

20	5

means block# 20, 21, 22, 23, 24 are free.

COMPARISON BETWEEN SPACE ALLOCATION TECHNIQUES

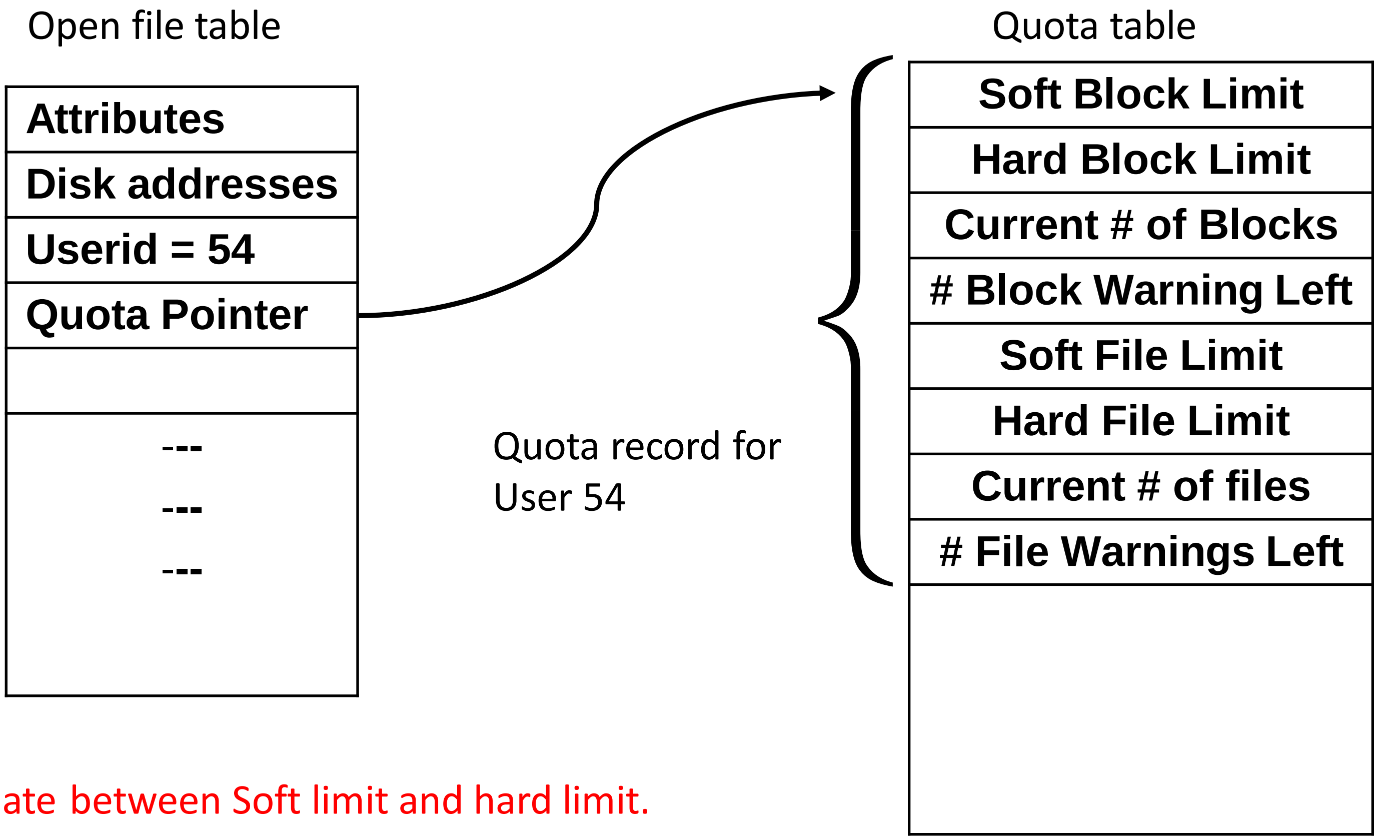
Example

Consider a system in which the directory entry block, bitmap block and index block are all in the main memory. Also consider a file that is of size 100 blocks. Find out the total number of I/O operations (inserting, deleting, reading a file block) required for the following actions. (Calculate for contiguous, index and linked allocation techniques).

- Inserting a block after the 50th block.
- Read 50th block.
- Insert after the 10th block.
- Delete 50th block.

DISK QUOTAS

- To prevent users from using too much disk space multi-user OS often provide a mechanism for enforcing disk quotas.
- System administrator allocate each user a maximum number of files and blocks and the OS ensure that the users do not exceed their quotas.



Open file table

Attributes
Disk addresses
Userid = 54
Quota Pointer
--- --- ---

Quota table

Soft Block Limit
Hard Block Limit
Current # of Blocks
Block Warning Left
Soft File Limit
Hard File Limit
Current # of files
File Warnings Left

Quota record for User 54

Differentiate between Soft limit and hard limit.

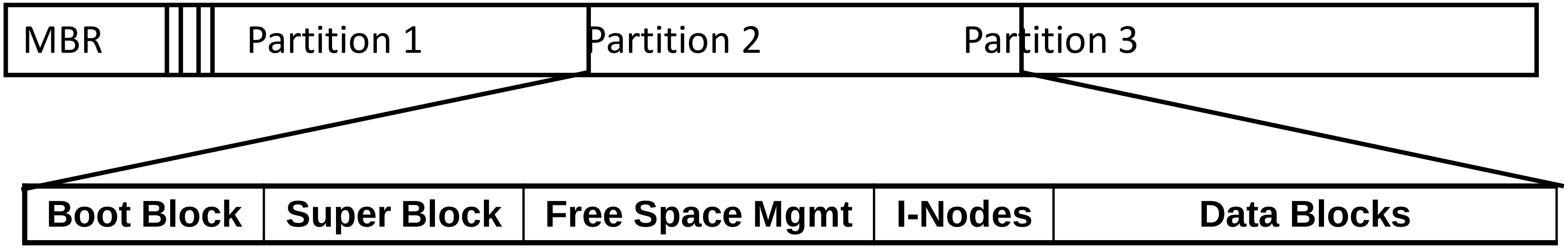
File System Architecture

FILE SYSTEM ARCHITECTURE

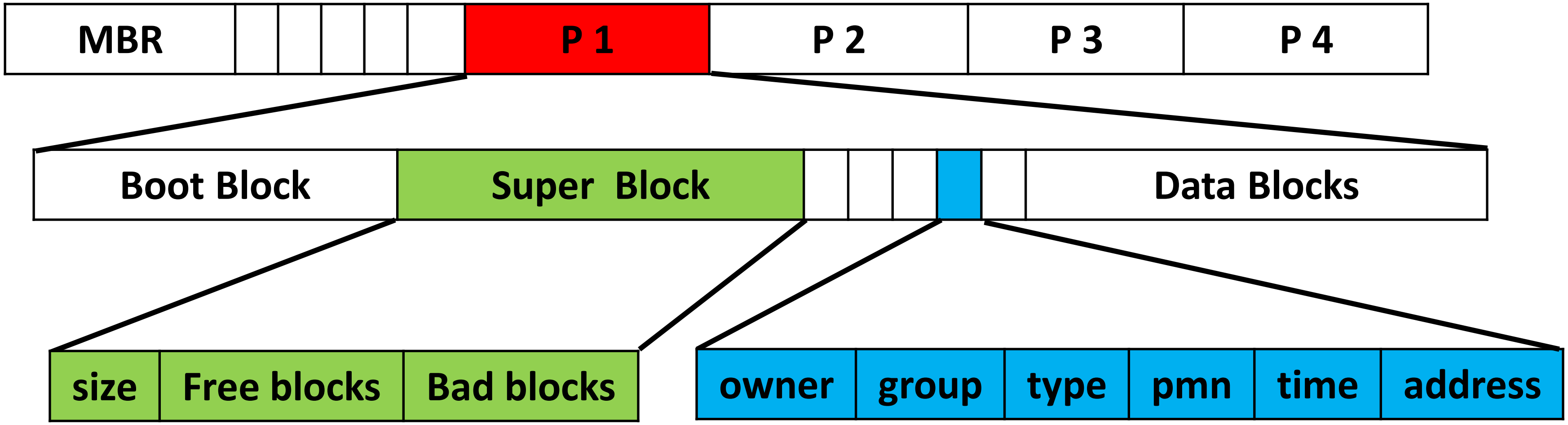
- Sector 0 of the disk is called the **MBR** (Master Boot Record) and is used to boot the computer. The end of MBR contain **partition table**, which gives the starting and ending address of each partition. One of the partitions in the table is marked as active & is called **active partition**.
- When a computer is powered up it needs to have an initial program to run. This initial program is called **bootstrap** program. In most computers the bootstrap program is located on ROM.
- The main job of bootstrap program is to initialize all aspects of the system from CPU registers to device controllers and contents of main memory and then go to **sector 0** of the disk and executes the MBR. The first thing the MBR program does is locate the active partition, and read its first block called the **boot block**, brings it in main memory and then executes it. The program in the boot block loads the OS contained in that partition.

FILE SYSTEM ARCHITECTURE (cont…)

- File systems are stored on disks. Most disks are divided into one or more partitions, with independent file system on each partition.
- Sector 0 of the disk is called the MBR (Master Boot Record) and is used to boot the computer. The end of MBR contain partition table, which gives the starting and ending address of each partition.
- One of the partitions in the table is marked as active & is called **active partition**. When the computer boots, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, and read its first block called the boot block and executes it. The program in the boot block loads the OS contained in that partition.



Schematic Structure of UNIX File System



FILE SYSTEM ARCHITECTURE (cont…)

ON DISK STRUCTURES

- **Boot Block -** Contains information needed by the system to boot an OS from that partition. In UNIX it is called *boot block*. In NTFS, it is called *partition boot sector*.
- **Super Block -** It contains partition details, like the numbers of blocks in the partition, size of blocks, free block count & free block pointers, and free FCB count and free FCB pointers. In UNIX this is called *super block*. In NTFS it is called *Master File Table*. Destruction of superblock will render the file system unreadable.

File Permissions
File Dates **(create, access, write)**
File Owner, Group, ACL
File Size
File Data Blocks
Hard Link Count
--- **---** **---** **---**

File Control Block

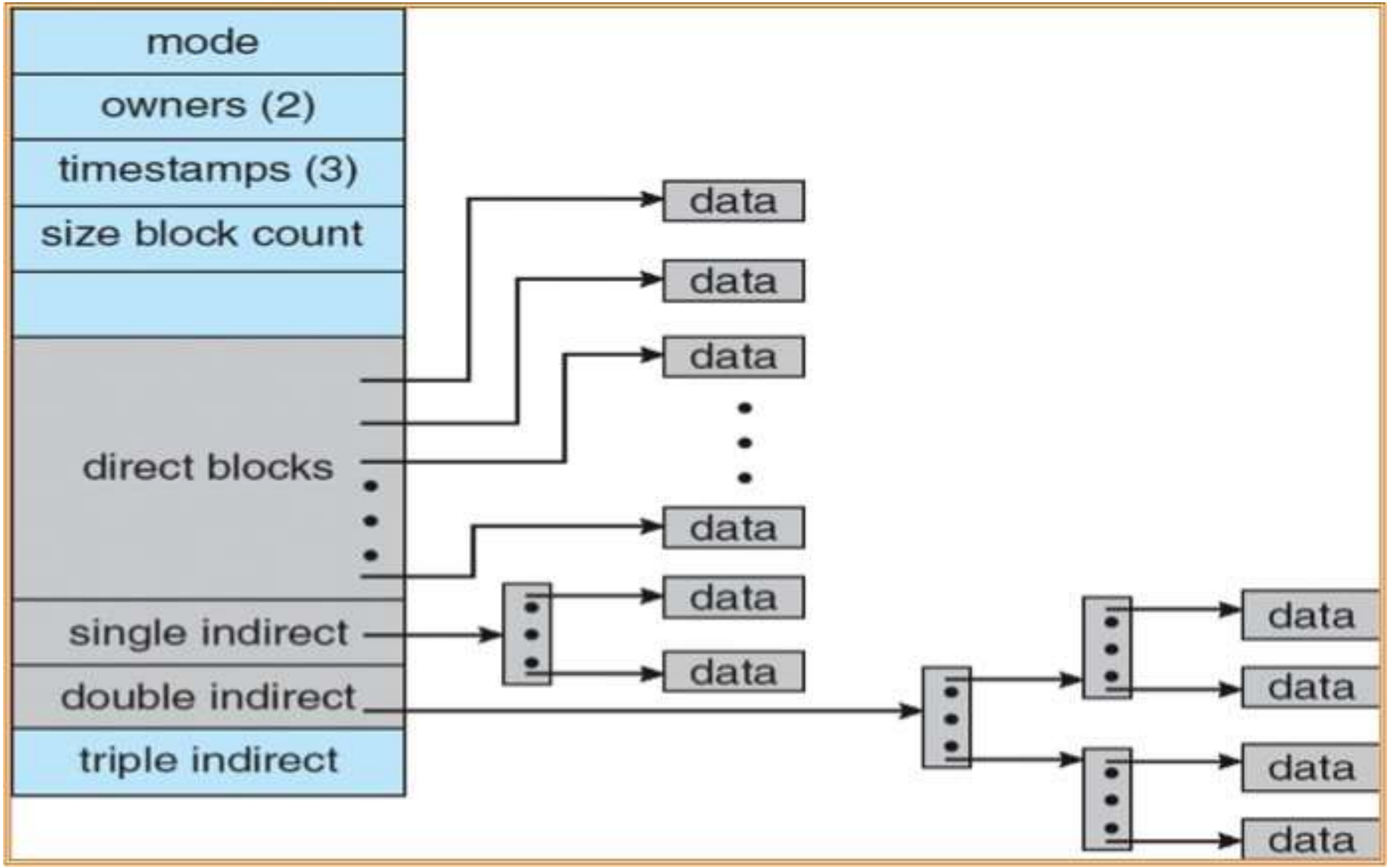
- **Directory Structure**.
- File Control Block - The way a Process Control Block is used for keeping the attributes of a process, similarly a File Control Block is used for keeping the attributes of a file. In UNIX it is called inode.

FILE SYSTEM ARCHITECTURE (cont…)

IN MEMORY DATA STRUCTURES

- **Partition Table -** It contains information about each mounted partition.
- **Directory Structure** - It holds the directory information of recently accessed directories.
- **System Wide Open File Table -** It contains a copy of the FCB of each open file, as well as other information.
- **Per Process File Descriptor Table -** It contains a pointer to the appropriate entry in the system wide open file table, as well as other information.

Structure of UNIX Inode Block



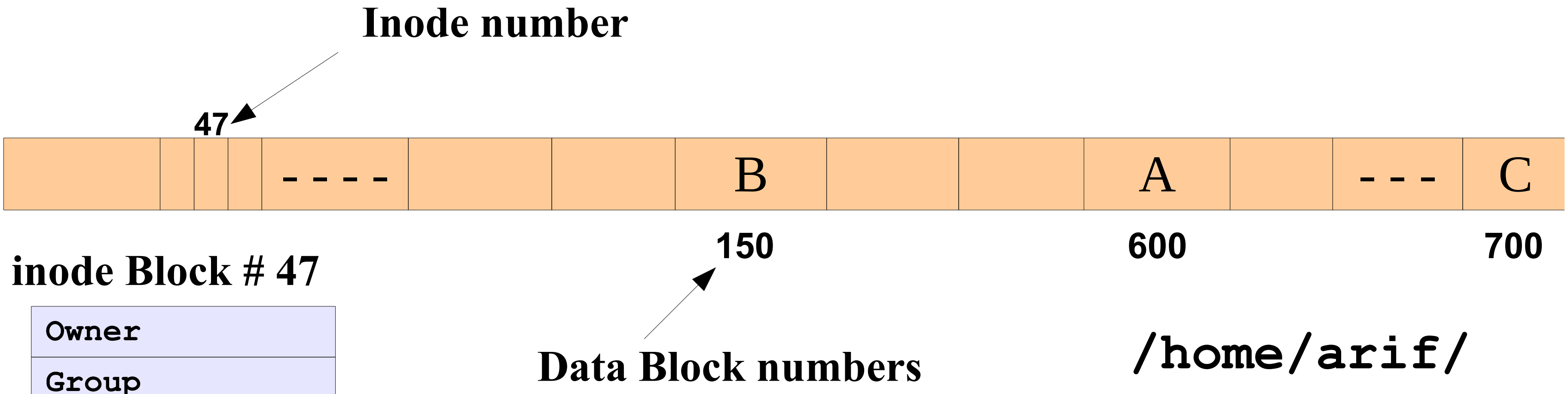
File System in Practice (Creating a File)

What happens when a user creates a file. The kernel stores the contents of the file in the data area, the properties of the file in an inode, and the name in a directory. So creating a new file involves the following four main operations:

- Store file's properties in a free inode
- Store file's data in free data blocks
- Record addresses of data blocks (in inode)
- Add file name to directory (inode, filename)

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

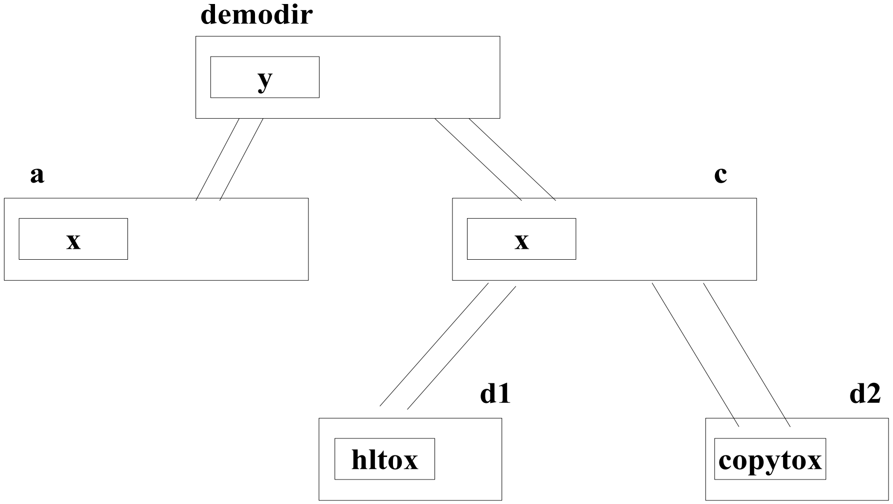
/home/arif/

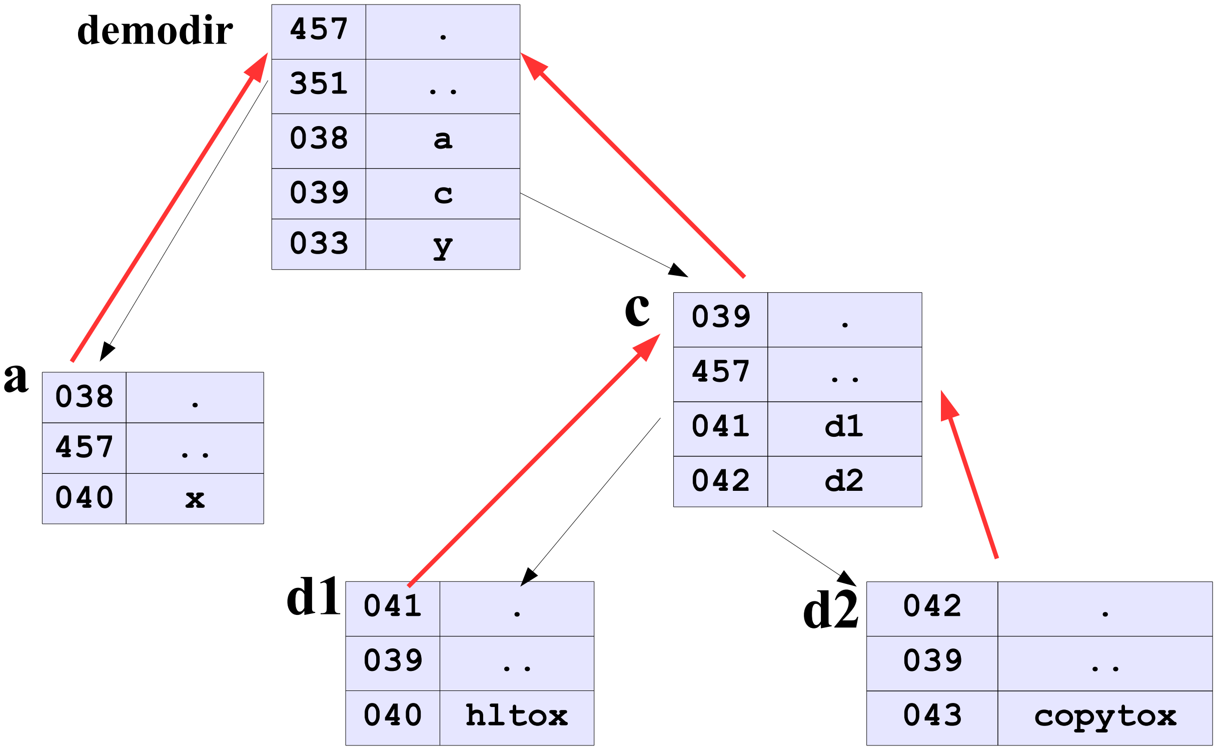
54	.
6	..
47	f1.txt
125	f2.txt
34	dir1

File System in Practice (Understanding Directories)

```
$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
```





demodir

457	.
351	..
038	a
039	c
033	y

a

038	.
457	..
040	x

c

039	.
457	..
041	d1
042	d2

d1

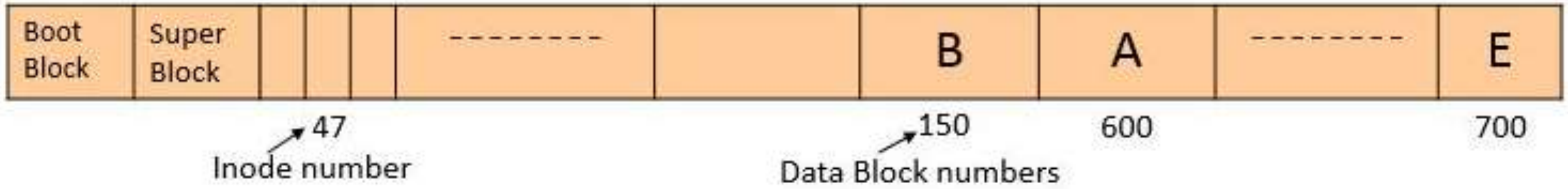
041	.
039	..
040	hltox

d2

042	.
039	..
043	copytox

File System in Practice (Accessing a File)

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



Inode # 2

/

2	.
2	..
561	home/
	bin

Inode # 561

home/

561	.
2	..
533	arif/
	rauf

Inode # 533

arif/

533	.
561	..
615	file1

Inode # 615

file1

Learning Linux is fun with Arif Butt

<EOF>

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

Connection to an Opened File

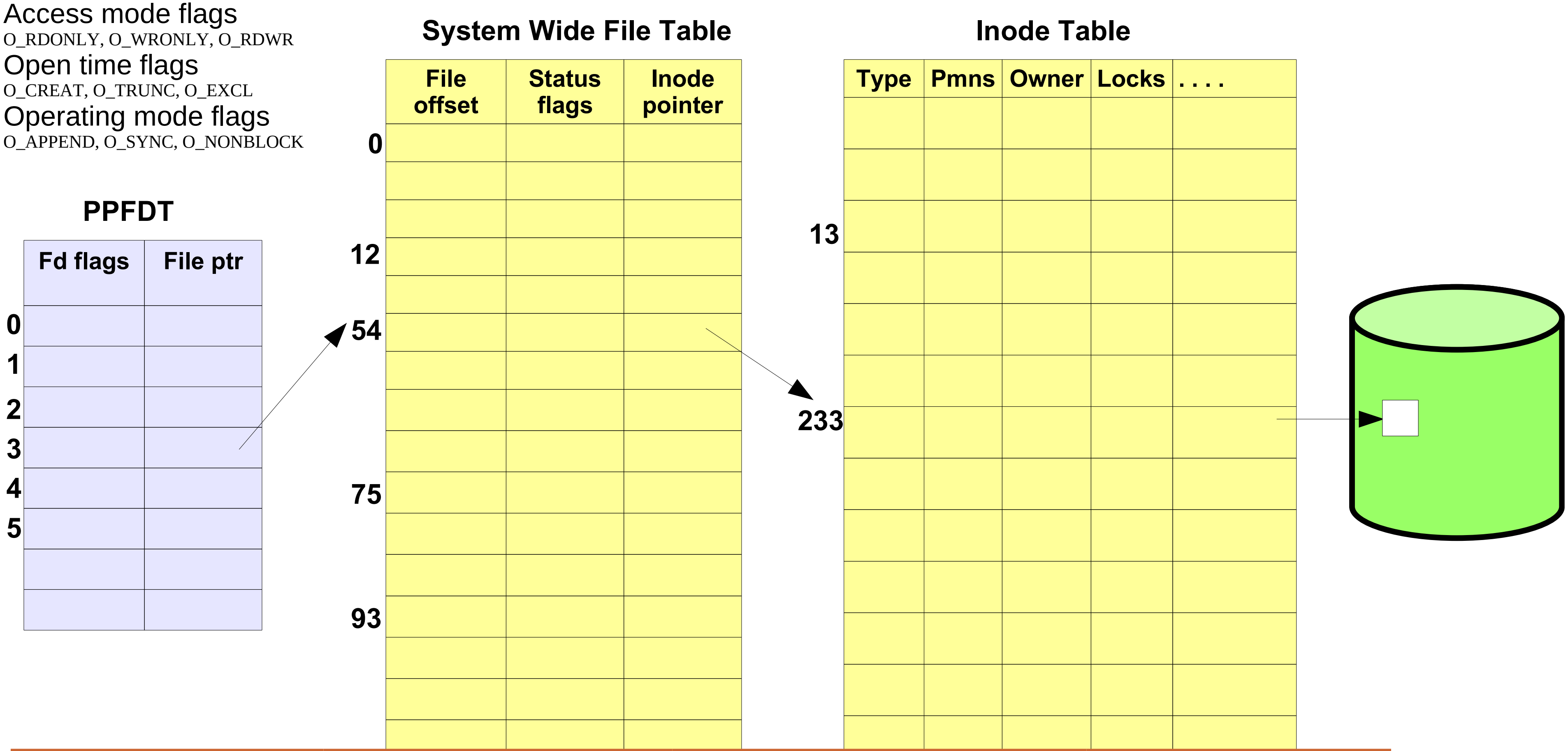
FILE DESCRIPTOR TO FILE CONTENTS

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

FILE DESCRIPTOR TO FILE CONTENTS

PPFDT:

For each process, the kernel maintains a table of open file descriptors. The total possible number of entries in this table is kernel variable OPEN_MAX (i.e., the maximum number of files that a process can open at any instant of time). Each entry in this table records information about a single file descriptor, including:

- A set of flags controlling the operation of the file descriptor (there is just one such flag, the `close-on-exec` flag; and
- A reference/pointer to the open file description

FILE DESCRIPTOR TO FILE CONTENTS

System Wide File Table:

The kernel maintains a system-wide table of all open file descriptions. The maximum number of entries in this table are the max number of files a system can open at any instant of time. Every open file description inside the system wide file table stores information relating to an open file, including:

- **Current file offset** (as updated by `read()` and `write()`, or explicitly modified using `lseek()`)
- **Status flags** specified when opening the file (i.e., the flags argument to `open()`), are of three types:
  - **Access mode flags:** (Required) (O_RDONLY, O_WRONLY, O_RDWR)
  - **Open time flags:** (optional) (O_CREAT, O_TRUNC, O_EXCL)
  - **Operating mode flags:** (optional) (O_APPEND, O_SYNC, O_NONBLOCK)
- A reference/pointer to the i-node object for this file.

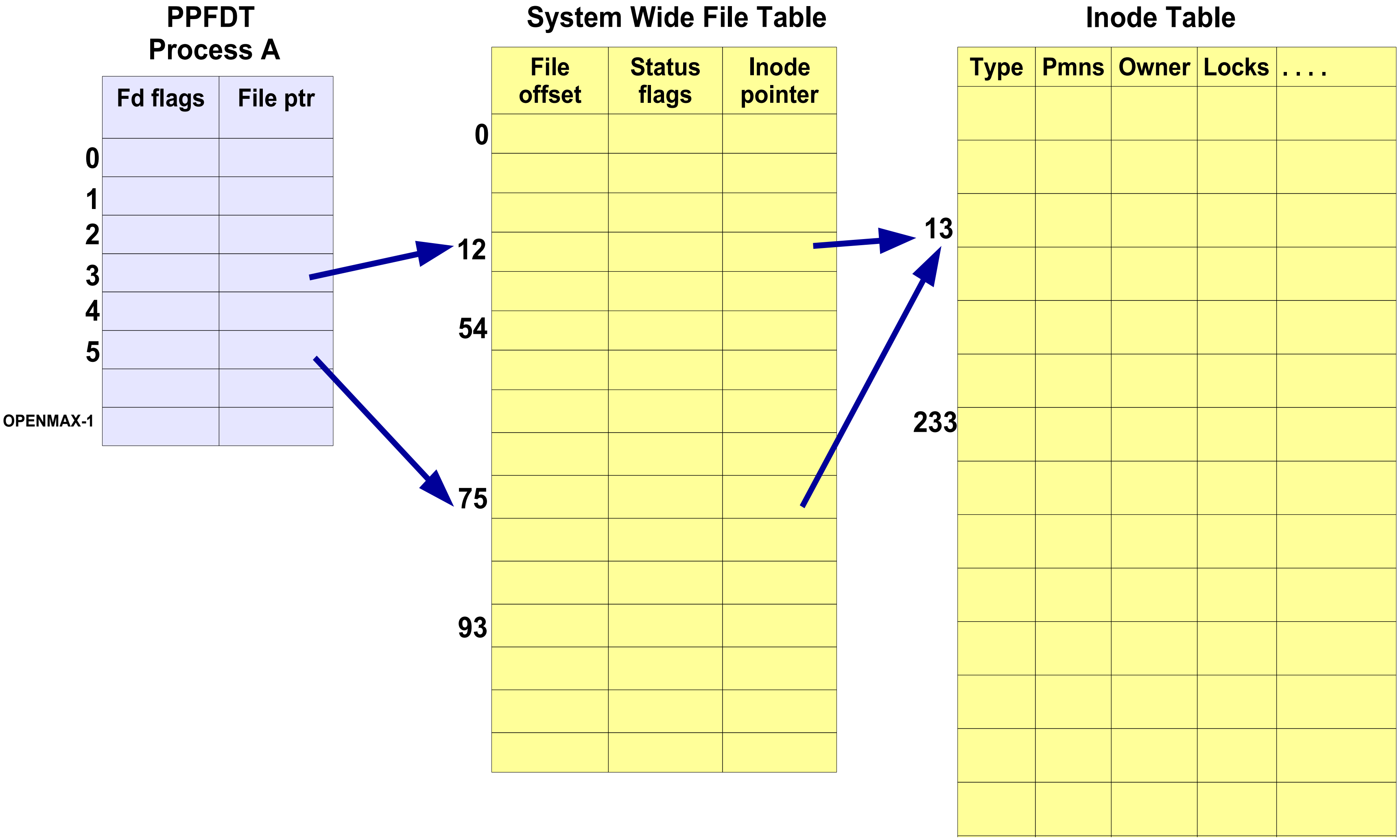
FILE DESCRIPTOR TO FILE CONTENTS

i-Node Table:

Each file system has a table of i-nodes for all files residing in the file system. The i-node for each file includes platter of information, some of which is given below:

- File type (**`-, d, l, p, c, b, s`**)
- File locks
- File Size (in Bytes as well as in blocks)
- Owner, Group
- Access Permissions
- Three time stamps (Modification `(ls -l)`, access `(ls -lu)`, status change `(ls -lc)`)
- Number of hard links
- Address for ten direct data block, and of three indirect pointers

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

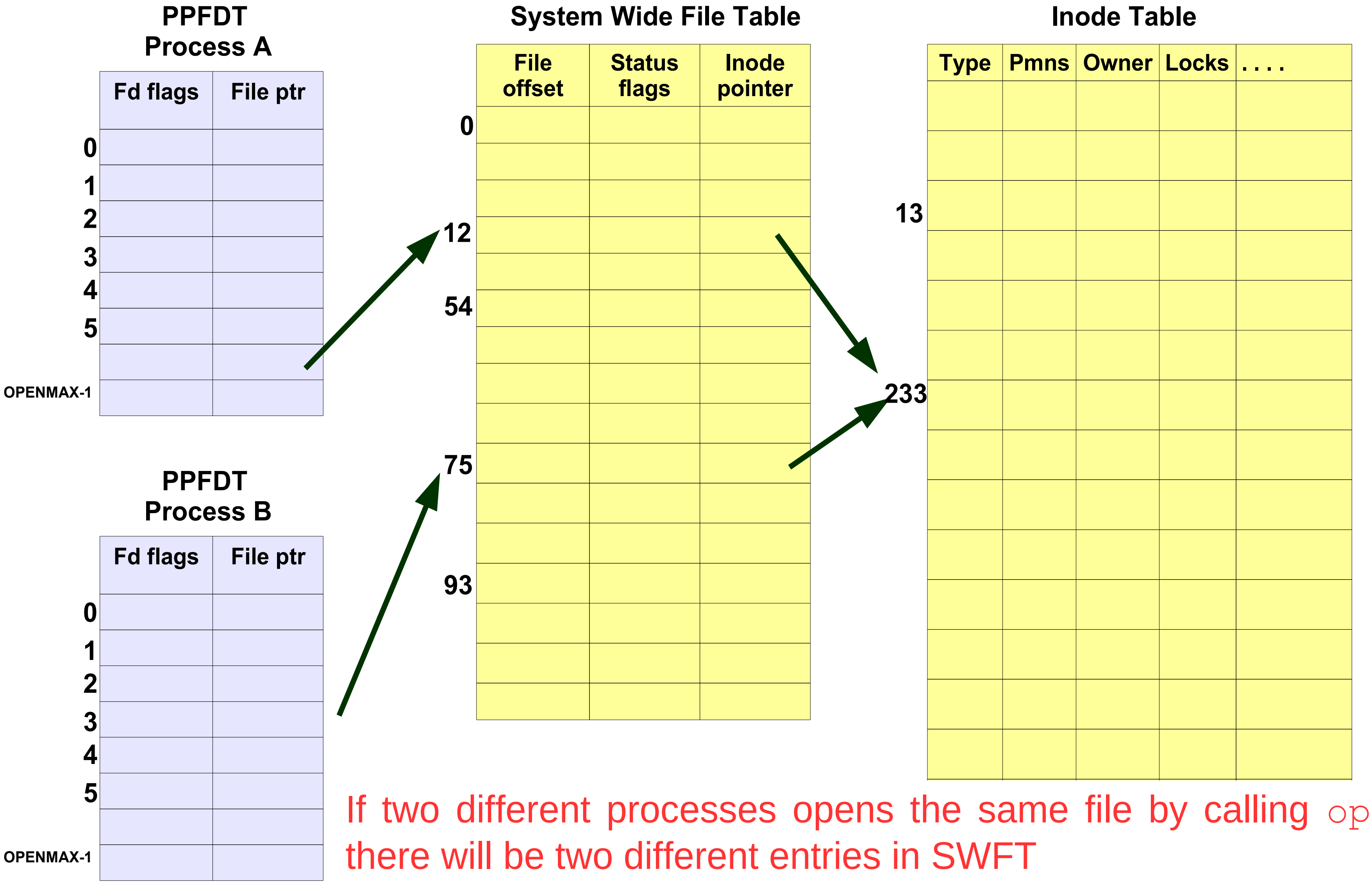
Inode Table

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

PPFDT
Process B

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

System Wide File Table

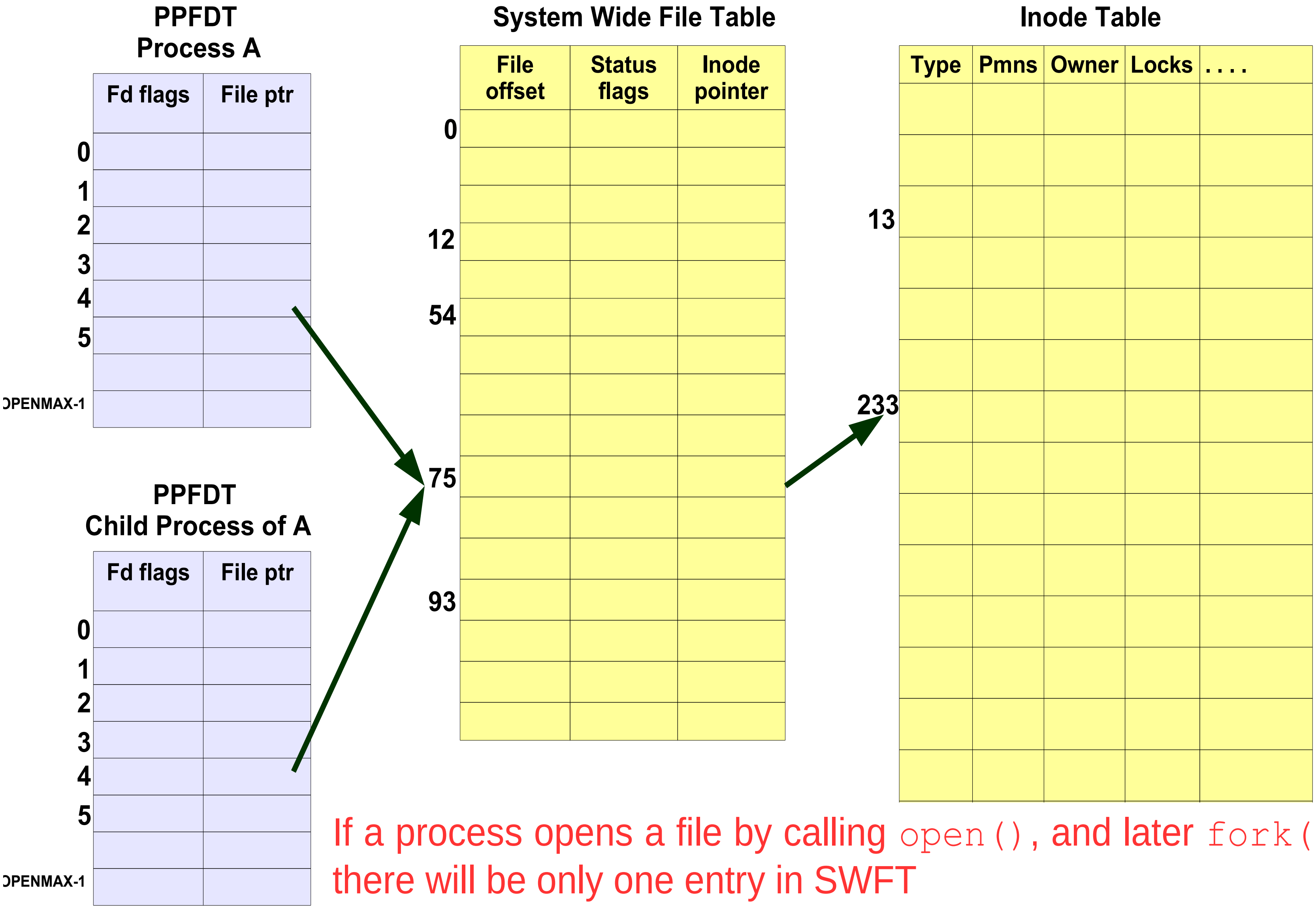
File offset | Status flags | Inode pointer
0 12 54 75 93

Inode Table

Type | Pmns | Owner | Locks | . . . .
13 233

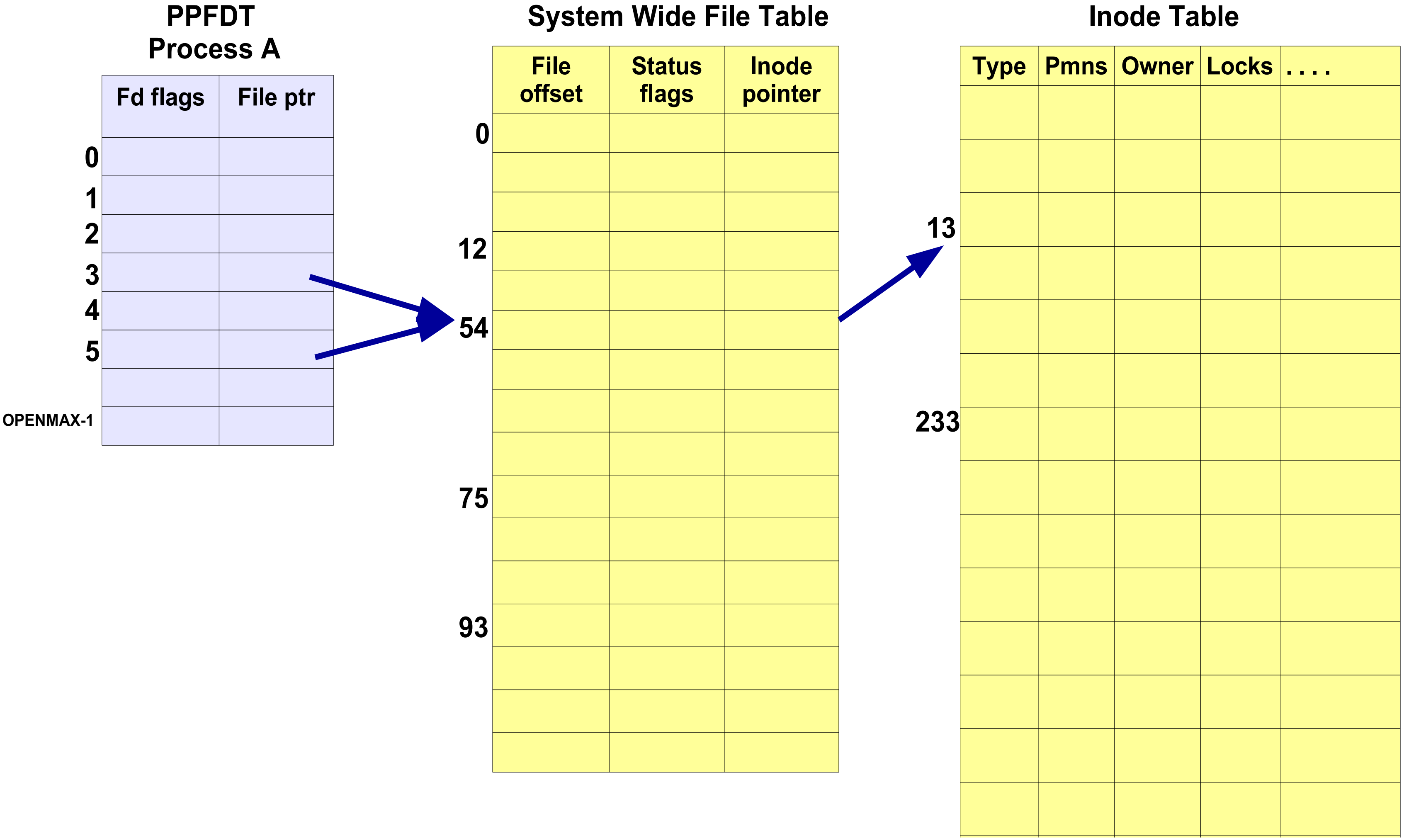
If two different processes opens the same file by calling `open()`, there will be two different entries in SWFT

Relationship Between fd and Open Files



If a process opens a file by calling `open()`, and later `fork()`, then there will be 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

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

File Sharing and links in UNIX

Please view the video lecture on hard and soft links at the following link:

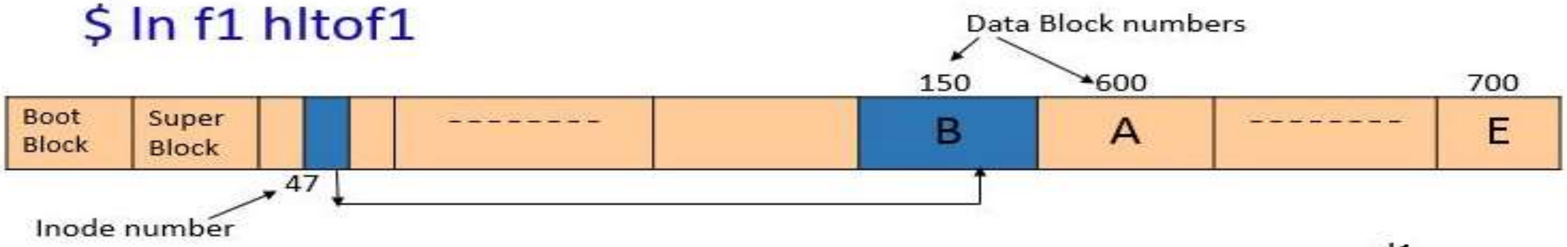
https://www.youtube.com/watch?v=g8xZgtuYiWI&t=4s

FILE SHARING

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through
  - Duplicating files.
  - Common login for members of a team.
  - Setting appropriate access permissions.
  - Common groups for members of a team.
  - Links.

LINKS IN UNIX - Hard Links

```
$ ln f1 hltof1
```



d1

47	F1
47	Hltof1

- A directory entry for the existing file is created. No new data block or inode block
- The hard link is represented as a regular file (-), and has the same inode number as that of the original file, and both has the same set of permissions, if you see its long listing
- With the creation of every hard link, the link count is incremented. By deleting a hard link or even the original file only the link count is decremented. You can still access the data, only directory entry is deleted. The data block and inode block are freed only when the link count reaches zero
- You touch one link, the timestamps of all hard links change
- A Regular user cannot have hard link to directories.
- You cannot have hard link across the file systems / partitions

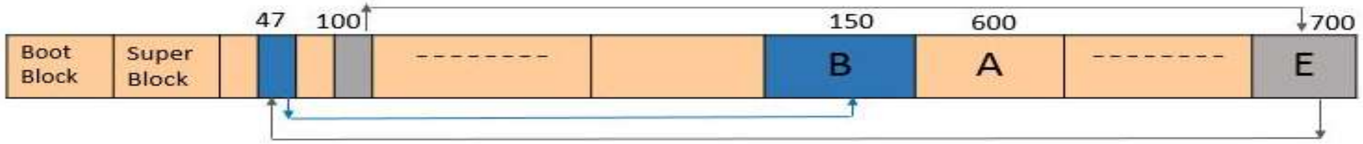
LINKS IN UNIX – Soft Links

```
ln -s  existingfile   linkname
```

- A directory entry, inode block and a data block are created
- Soft link is represented as (l), and has the different inode number as that of the original file. The permissions on the symlink are irrelevant, since the permissions of the target apply
- The link count remains one
- Deletion of a link does not affect the original file; only the link is removed. If the original file is deleted, the space for the file is de-allocated, leaving the links dangling which can easily be searched and deleted by the administrator
- Whether you touch the file or soft link the time stamp of original file change
- You can have symbolic link to directories
- You can have symbolic link across the file systems / partitions because they share an inode number and an inode table is unique to a file system

LINKS IN UNIX – Soft Links

```
$ ln -s f1 sltof1
```



d1

47	f1
100	sltof1

- A directory entry, inode block and a data block are created
- Soft link is represented as (l), and has the different inode number as that of the original file. The permissions on the symlink are irrelevant, since the permissions of the target apply
- The link count remains one
- Deletion of a link does not affect the original file; only the link is removed. If the original file is deleted, the space for the file is de-allocated, leaving the links dangling which can easily be searched and deleted by the administrator
- Whether you touch the file or soft link the time stamp of

LINKS IN UNIX

Question: Why can't we have hard links to directories?

Allowing hard links to directories would break the directed acyclic graph structure of file system. This will create directory loops which will make fsck(8), find(1) and other file tree walkers error prone

Question: Why can't we have hard links across partitions?

A hard link let us have multiple file names that point to the same inode. This can work only if the two hard links are on the same partition or file system

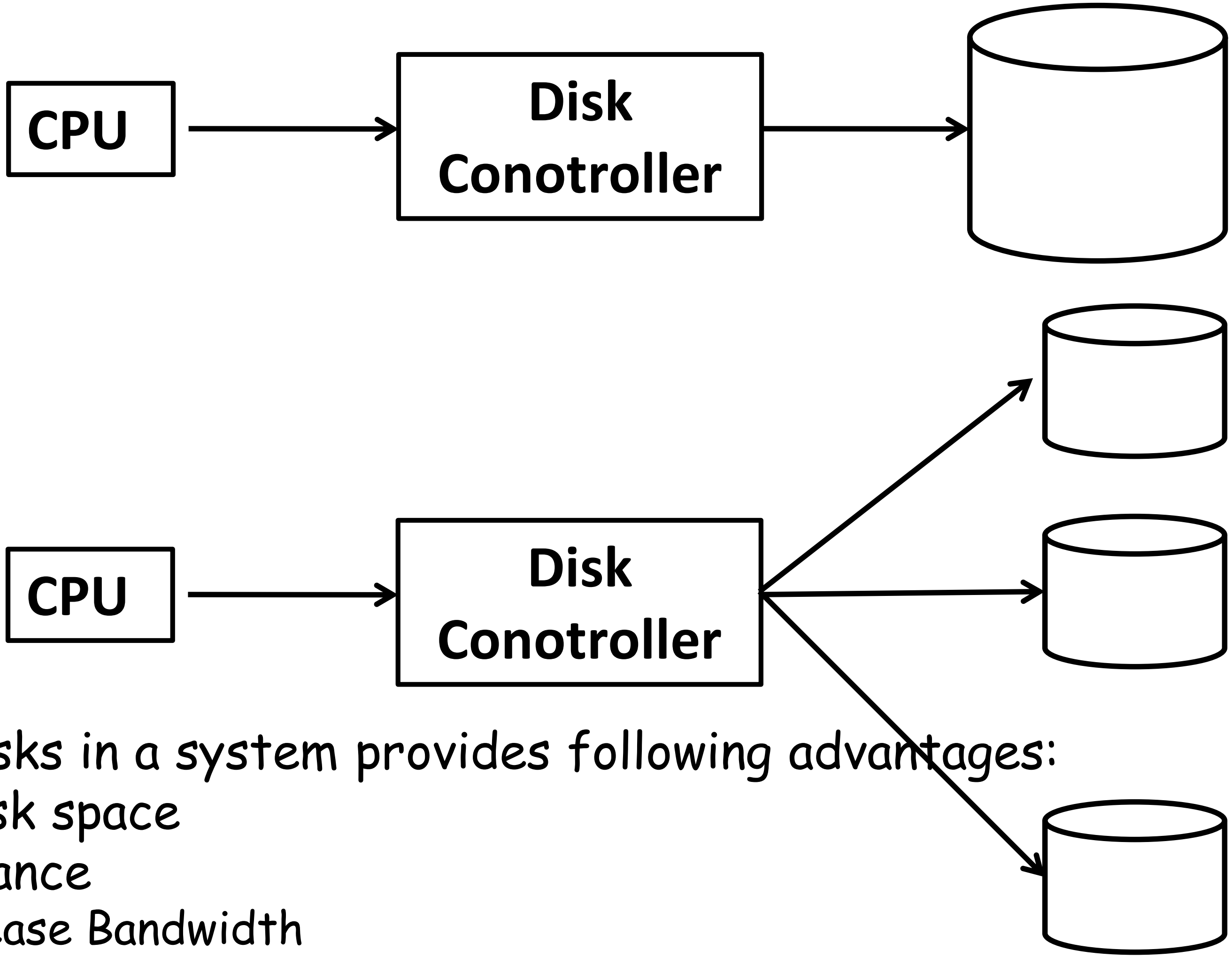
A symbolic link instead points to the file name, which is then linked to the inode holding the file's data

RAID

FILE SYSTEM RELIABILITY

- Loss of data in a file system can have catastrophic effect. Need to ensure reasonable level of safety against data loss in the event of system failures. Three threats are:
  - Accidental or malicious deletion of data by user
  - Media (disk) failure
  - System crash during file system modifications, leaving data on disk in an inconsistent state
- Solutions are:
  - Back up
  - RAID disks
  - Versioning File Systems
  - **Versioning File Systems -** File System created a new version every time a file is modified. Old versions are kept until explicitly deleted. After a system crash in the middle of a FS operation, FS metadata may be in an inconsistent state, e.g., a file was deleted, but its disk blocks have not yet been added to the free list.

REDUNDANT ARRAY OF INDEPENDENT DISKS



Multiple disks in a system provides following advantages:

- Large disk space
- Performance
  - Increase Bandwidth
  - Lower latency, because smaller disk spins faster
- Reliability
  - Ability to recover from failure
  - By storing redudent information on multiple disks

REDUNDANT ARRAY OF INDEPENDENT DISKS

- The term RAID was originally coined in a paper by a group of researchers at the University of California at Berkeley. The RAID strategy replaces large capacity disk drives with multiple smaller capacity disks and distribute data on them in such a way that could be used to improve disk **performance**, **reliability** or both. Peterson (one of the researcher) defined RAID as “Redundant Array of Inexpensive Disks”, but later industry defined I as “Independent” instead of “Inexpensive”.
- The basic idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller. A RAID should look like a SLED (Single Large Expensive Disk) to the OS but have better performance and better reliability.

RELIABILITY IN RAID

Improvement of Reliability via Redundancy

- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
- Suppose *mean time of failure* of a single disk is 100,000 hours(4166 days). In an array of 100 disks the mean time of failure of a disk will be 100,000 / 100 = 1000 hrs (41.6 days). This is not acceptable, the solution to the problem of reliability is to introduce redundancy; we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information.
- The simplest but most expensive approach of introducing redundancy is to duplicate every disk. This technique is called *mirroring / shadowing*. A logical disk consists of two or more disks and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if second disk fails before the first failed disk is replaced.
- Mean time to failure of a mirrored disk depends on two factors:
  - Mean time to failure of individual disk
  - Mean time to repair. (Time it takes to replace a failed disk and to restore the data on it)

PERFORMANCE IN RAID

Improvement of Performance via Parallelism.

- With disk mirroring, the speed at which the read requests are handled is doubled, since the read requests can be sent to either disks.
- However, the transfer rate of each read is the same as in single disk system, but the number of reads per unit time has doubled.
- We can improve the transfer rate as well, by stripping data across multiple disks, called *Data Stripping*. This stripping can be at *bit level* or *block level*.
- For example in bit level stripping, if we have an array of eight disks, we write bit **i** of each byte to disk **i**. The array of eight disks can be treated as single disk with sectors that are eight times the normal size, and more important that have eight time the access rate.

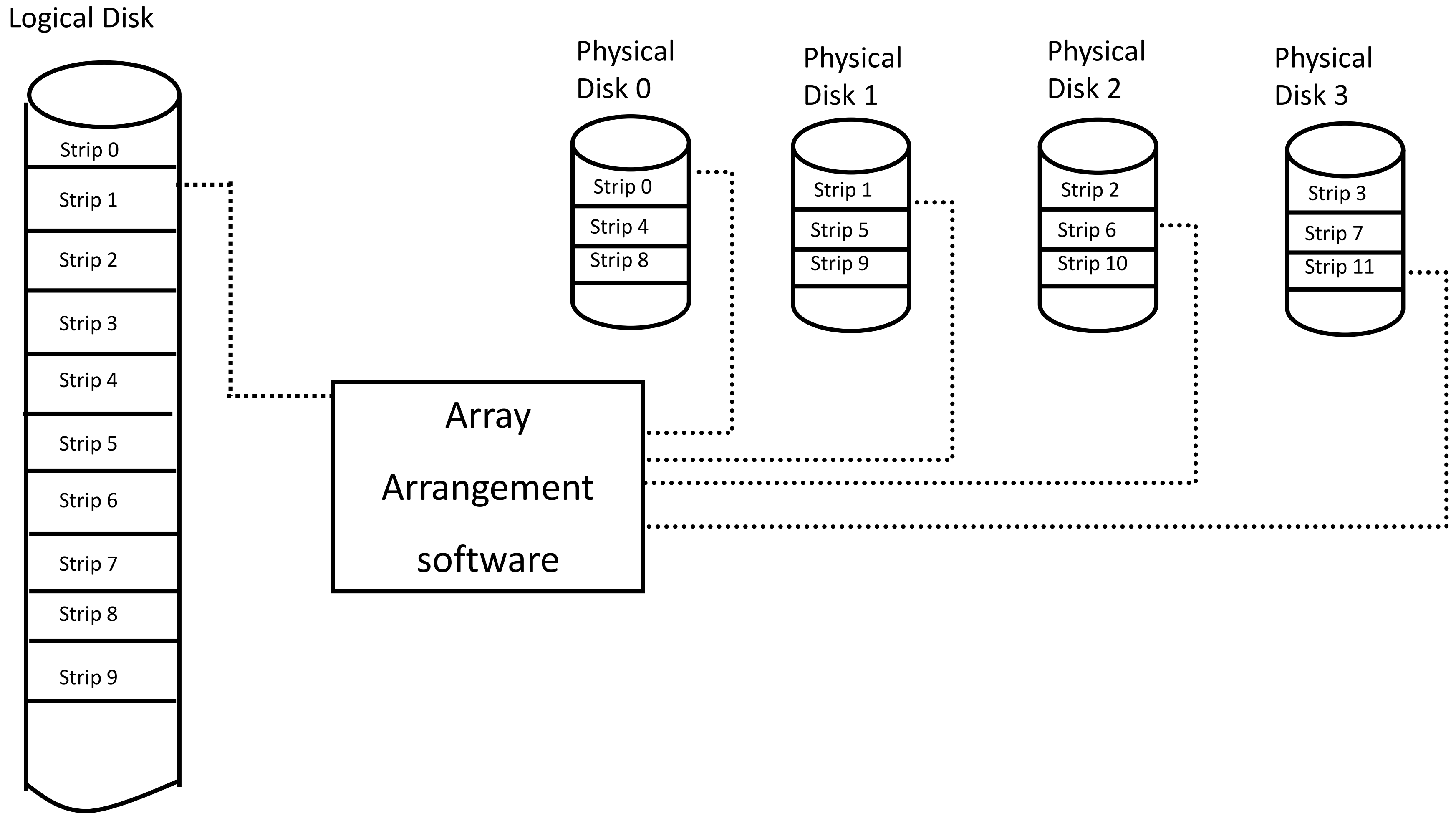
Note: Mirroring provides high reliability but it is expensive. **Stripping** provides high data transfer rates, but it does not improve reliability.

RAID - 0

- It is also called ***Non-redundant stripping***. RAID 0 is not a true member of the RAID family, because it does not include redundancy to improve performance.
- Stripping (Distributing data over multiple disks) is done at the level of blocks, but without any redundancy (of data or parity bits).
- The user and system data are distributed across all of the disks in the array. Thus if two different I/O requests are pending for two different blocks of data, then there is a good chance that these requested blocks are on different disks. Therefore the two requests can be issued in parallel, reducing the I/O queuing time.
- Disadvantages
  - Works worst with OS that habitually ask for data one sector / block at a time. The result will be correct but there is no parallelism and hence no performance gain.
  - The other disadvantage is low mean time to failure.

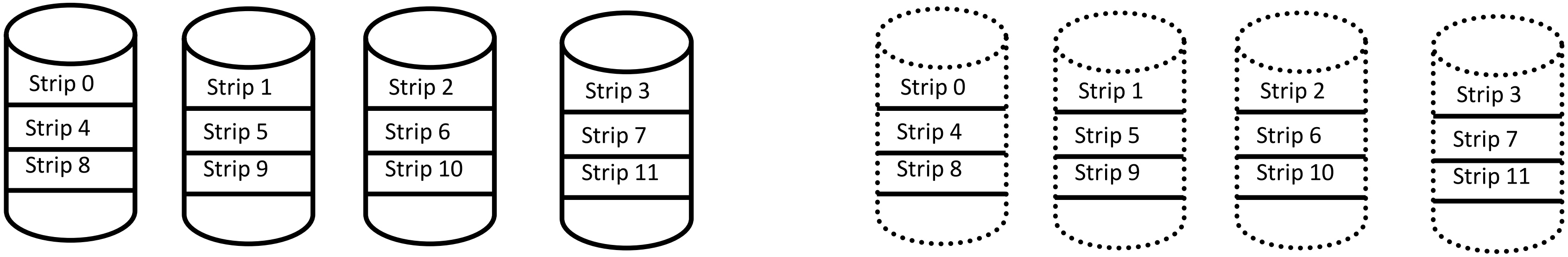
RAID - 0 (cont…)

Data Mapping for a RAID level 0 Array



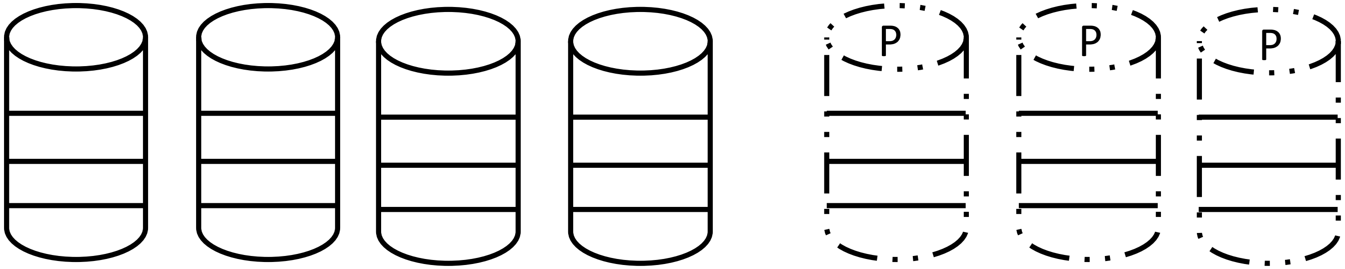
RAID - 1

- RAID 1 is also known as **Disk Mirroring / Shadowing.**
- In RAID 1 redundancy is achieved by simply duplicating the data. Data stripping is used as in RAID 0, but this time each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. In the figure an array of four disks is shown, there are four primary disks and four back up disks.
- A read request can be serviced by either of the two disks that contains the requested data, whichever involves less seek time plus rotational latency.
- A write request requires that both corresponding strips be updated, but this can be done in parallel. The write performance is dictated by the slower of the two writes. However, there is no "write penalty" with RAID 1.
- Fault tolerance is excellent; if a drive crashes, the copy is simply used instead.
- The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports.



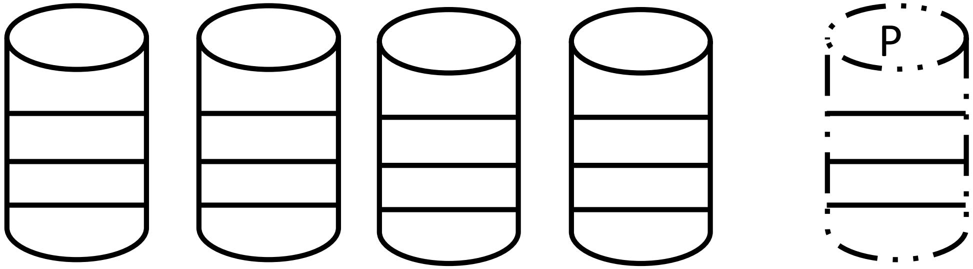
RAID - 2

- RAID 2 is also known as ***Memory Style Error Correcting Code Organization***.
- Error Correcting schemes store two or more extra bits, and can reconstruct the data if a single bit gets damaged. Typically **Hamming code** is used, which is able to correct single-bit errors and detect double bit errors.
- Consider a four disk RAID array, first bit of each byte could be stored in disk 1, the second bit in disk2, and so on until the fourth bit is stored in disk 4, and the error-correction bits are stored in further three disks.
- If one of the disks fails, the remaining bits of the byte and the associated error correction bits can be read from other disks and be used to reconstruct the damaged data.
- On a single read, all disks are simultaneously accessed. On a single write, all data disks and parity disks must be accessed for the write operation.
- RAID 2 is only used in an environment in which many disk errors occur. (normally not implemented).



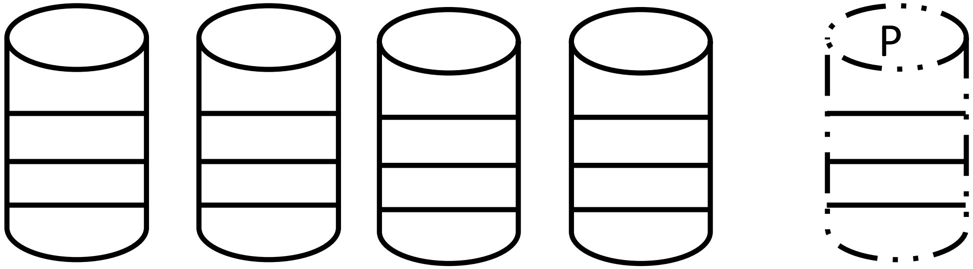
RAID - 3

- RAID 3 is also known as ***Bit Interleaved parity organization.***
- RAID 3 is organized in a similar fashion to RAID 2. The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array is.
- RAID 3 employs parallel access, with data distributed in small strips.
- A single bit is used for error correction as well as for detection.
- Parity Calculation
  - **X4 = X3 + X2 + X1 + X0**
- Suppose X1 has failed.
  - **X1 = X4 + X3 + X2 + X0**



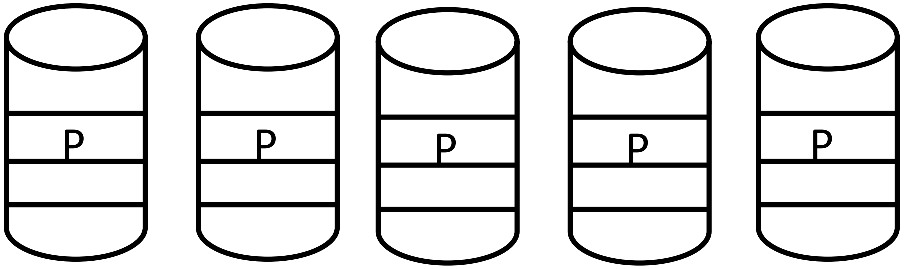
RAID - 4

- RAID 4 is also known as ***Block Interleaved parity organization.***
- RAID 4 uses block level stripping and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks.
- If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.
- *A* block read accesses only one disk, so multiple read accesses can proceed in parallel.
- In order to write, both the old value of the parity block and the old value of the data block being written have to be read for the new parity to be computed. This is known as *read-modify-write*. So write of a block, need two read and two write operations.



RAID - 5

- RAID 5 is also known as ***Block Interleaved distributed parity organization.***
- RAID 5 differs from RAID 4 by spreading data and parity among all disks, rather storing data in **n** disks and parity in one disks.
- A parity block cannot store parity for blocks in the same disk, e.g. the parity for data blocks of first three disks is stored in the fourth disk.



RAID - 6

- RAID 6 is also known as *P + Q redundancy scheme.*
- RAID 6 is much like RAID 5, but stores extra redundant information to guard against multiple disk failures.
- Instead of parity it uses the Read-Solomon code.

H/W RAID vs S/W RAID

- Hardware RAID
  - Separate physical disks combined into one or more logical disks by the disk controller or disk storage cabinet hardware.
- Software RAID
  - Noncontiguous disk space combined into one or more logical partitions by the fault-tolerant software disk driver, FTDISK
  - Software RAID implements RAID 1 (Disk mirroring) and RAID 5 (Disk duplexing)

DISK ATTACHMENT

Computers access disk storage in two ways.

- Host attached Storage.
- Network Attached Storage.

Host Attached Storage

Storage accessed via local I/O ports. These ports are available in several technologies. Typical are:

- IDE
- EIDE
- SCSI

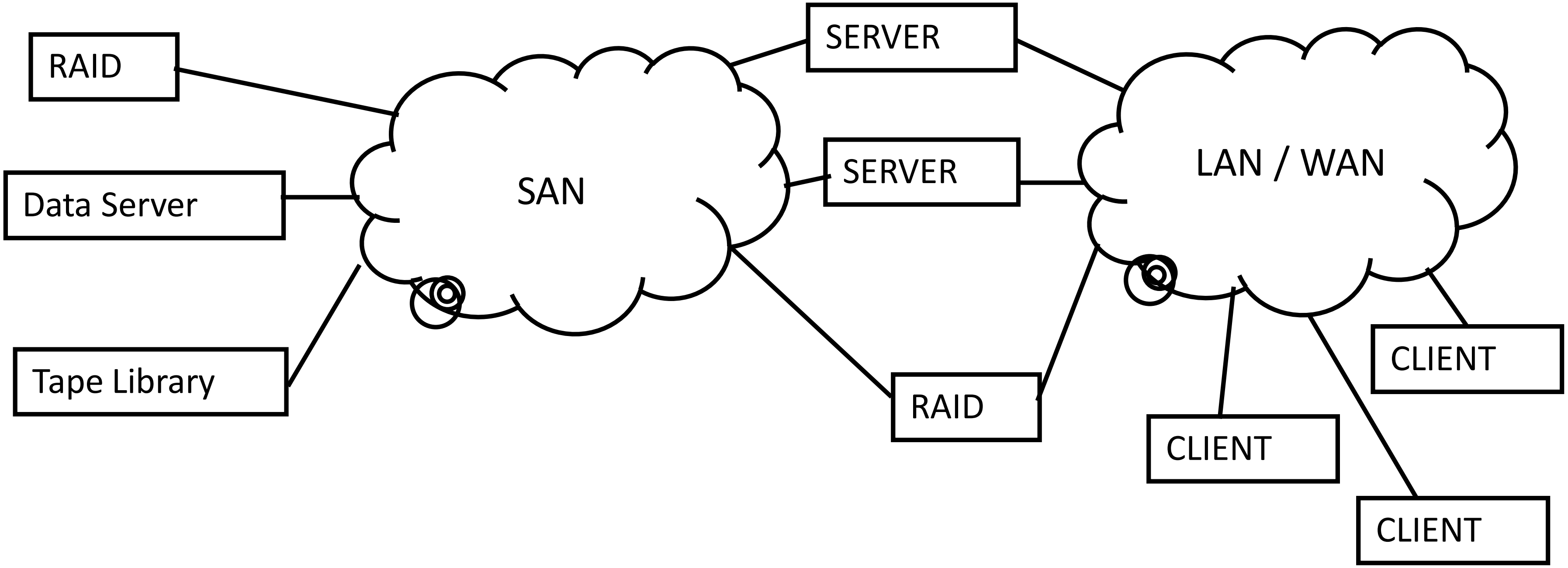
Network-Attached Storage (NAS)

- A NW-attached storage device is a special purpose storage system that is accessed remotely over a data network.
- Clients access NAS via a remote-procedure-call interface such as NFS for UNIX systems, or CIFS (Common Internet File System) for Windows machines.
- NAS provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host attached storage.

DISK ATTACHMENT (cont…)

Storage-Area Network (SAN)

- One drawback of NAS systems is that the storage I/O operations consume bandwidth on the data NW.
- A SAN is a private network (using storage protocols rather than networking protocols) among the servers and storage units, separate from the LAN / WAN that connects the servers to the clients.
- SAN systems are available but are not well standardized or interoperable.



Summary

We're done for now, but Todo's for you after this lecture…

- Go through the related video lectures # 20 and 21.
- Practice, practice and practice…