CO4

Session - 4

COURSE NAME: OPERATING SYSTEMS

COURSE CODE: 23CS2104R/A

FILES AND DIRECTORIES

AIM OF THE SESSION

To familiarize students with the basic concept of **files and directories.**

.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Demonstrate what is meant by Files.
2. Demonstrate what is meant by Directories.
3. Describes the types of File Allocation Method.
4. Describes the File System Implementation.

LEARNING OUTCOMES

At the end of this session, you should be able to:

1. Defines what is meant by Files.
2. Defines the File System Implementation.
3. Summarizes the Role of **files and directories**.

PERSISTENCE STORAGE

- Keep data **intact** even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device

- Two key abstractions in the virtualization of storage
  - File
  - Directory

FILE

A file is a named collection of related information recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage

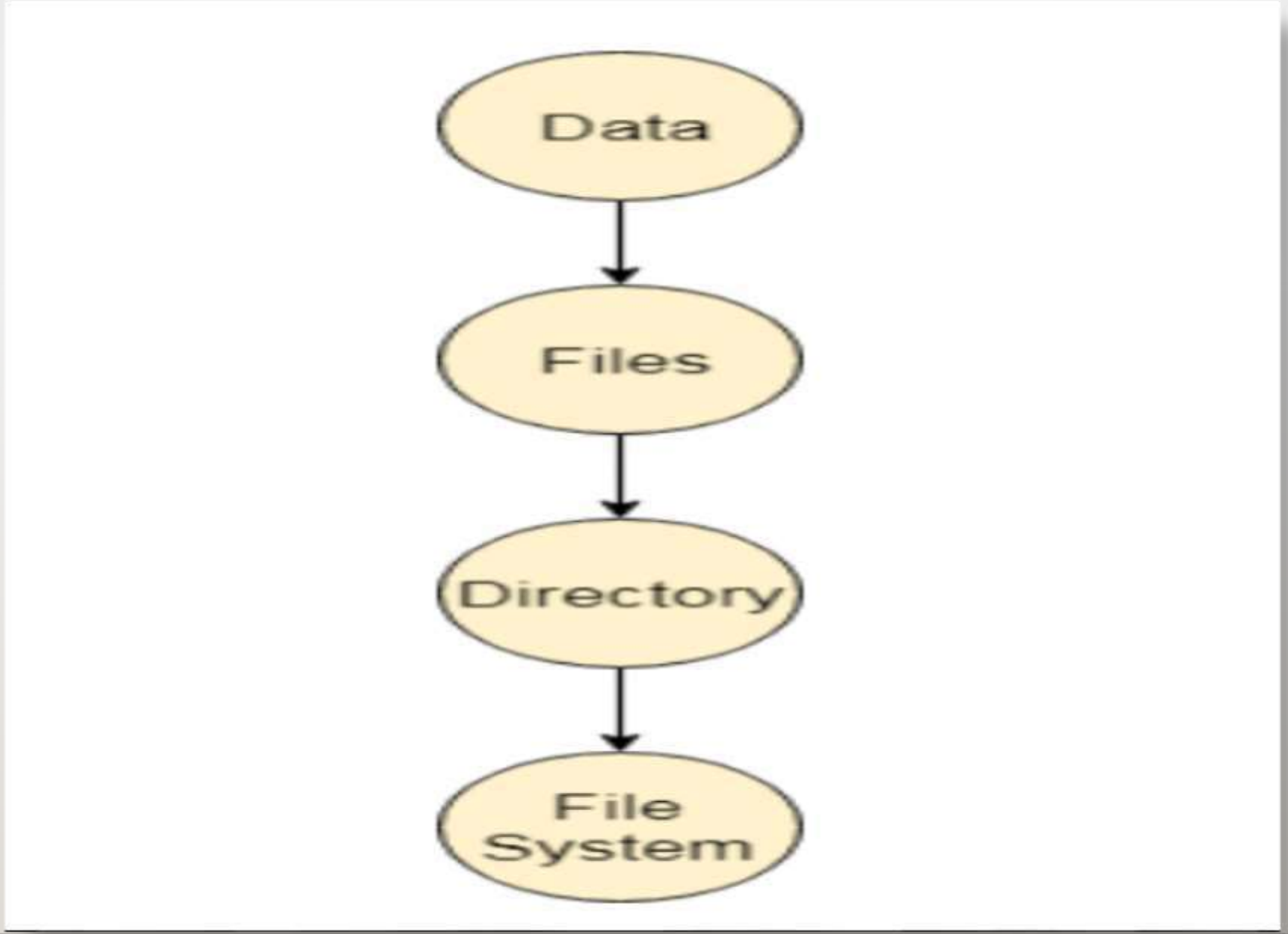
In general, A linear array of bytes OR sequences of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. Each file has a low-level name as an inode number The user is unaware of this name.

The filesystem has a responsibility to store data persistently on disks.

DIRECTORY

- Directory is like a file and also has a low-level name.
  - It contains a list of (user-readable name, low-level name) pairs.
  - Each entry in a directory refers to either *files* or other *directories*.

- Example)
  - A directory has an entry ("f1","10")
    - A file "f1" with the low-level name "10"



FILE ATTRIBUTES

1. Name

Every file carries a name by which the file is recognized in the file system. One directory cannot have two files with the same name.

2. Identifier

Along with the name, Each File has its extension which identifies the type of the file. For example, a text file has the extension .txt, and A video file can have the extension .mp4.

3. Type

In a File System, the Files are classified in different types such as video files, audio files, text files, executable files, etc.

4. Location

In the File System, there are several locations in which, the files can be stored. Each file carries its location as its attribute.

5. Size:

The Size of the File is one of its most important attributes. By size of the file, we mean the number of bytes acquired by the file in the memory.

6. Protection:

The Admin of the computer may want the different protections for the different files. Therefore each file carries its own set of permissions to the different groups of Users.

7. Time and Date:

Every file carries a time stamp which contains the time and date on which the file is last modified.

FILE OPERATIONS

1. **Create:** This operation is used to create a file in the file system.

2. **Open**: This operation is the common operation performed on the file. Once the file is created, it must be opened before performing the file processing operations.

3. **Write:** This operation is used to write the information into a file. A system call write() is issued that specifies the name of the file and the length of the data that has to be written to the file

4. **Read operation:** This operation reads the contents from a file. A Read pointer is maintained by the OS, pointing to the position up to which the data has been read.

5. Re-position or Seek :

The seek system calls re-positions the file pointers from the current position to a specific place in the file i.e. forward or backward depending upon the user's requirement.

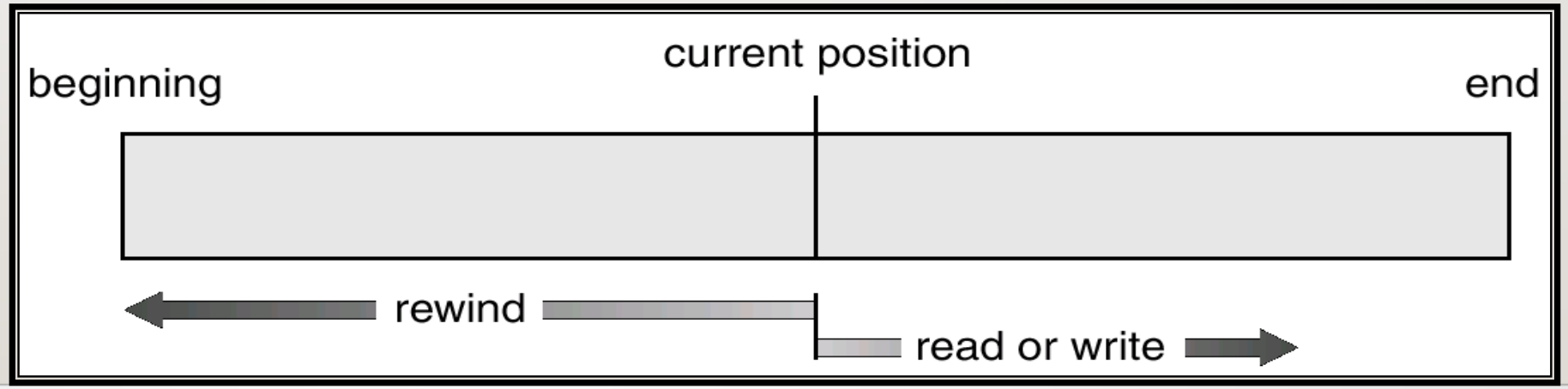
fseek(FILE *filePtr, long offset, int origin);

The origin can have the following three values:

SEEK_SET: It denotes the beginning or starting of the file.

SEEK_CUR: It denotes the current position of the file pointer.

SEEK_END: It denotes the end of the file.



6. Delete: Deleting the file will not only delete all the data stored inside the file it is also used so that disk space occupied by it is freed.

7. Truncate: Truncating is simply deleting the file except for deleting attributes. The file is not completely deleted although the information stored inside the file gets replaced.

8. Close: When the processing of the file is complete, it should be closed so that all the changes made permanent and all the resources occupied should be released. On closing it deallocates all the internal descriptors that were created when the file was opened.

9. Append operation: This operation adds data to the end of the file.

10. Rename operation: This operation is used to rename the existing file

HOW TO MANAGE FILES

Several pieces of data are needed to manage open file

- **Open-file table**: Tracks open files.
- **File pointer**: pointer to last read/write location, per process that has the file open.
- **File-open count**: counter of the number of times a file is open – to allow removal of data from the open-file table when the last process closes it.
- **Disk location of the file**: Cache of data access information.
- **Access rights:** **P**er-process access mode information.

FILE TYPES

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 com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

ALLOCATION METHOD

An allocation method refers to how disk blocks are allocated for files:

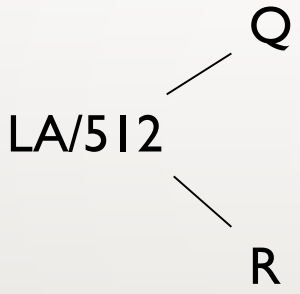
- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

CONTIGUOUS ALLOCATION METHOD

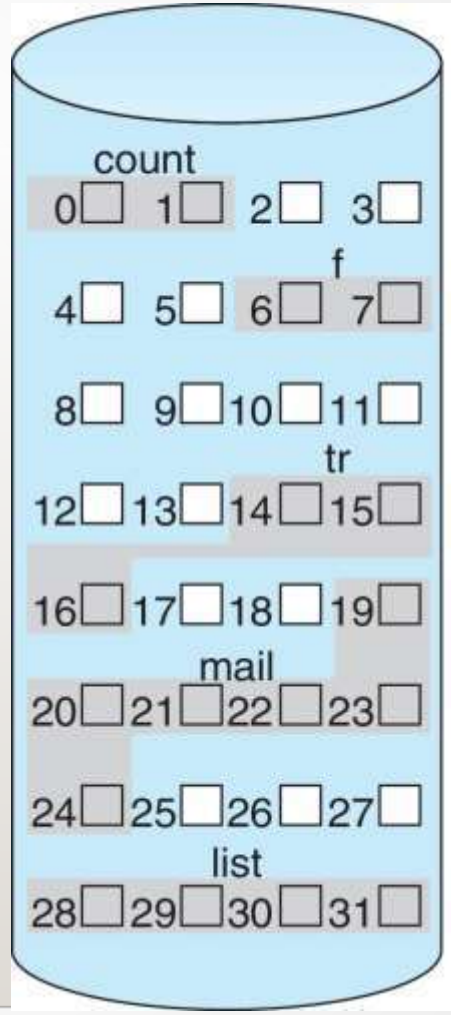
- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies a set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - Finding space on the disk for a file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

CONTIGUOUS ALLOCATION (CONT.)

- Mapping from logical to physical (block size =512 bytes)



- Block to be accessed = starting address + Q
- Displacement into block = R



directory

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

EXTENT-BASED SYSTEMS

- Many newer file systems (i.e.,Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extent
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

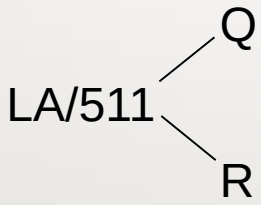
LINKED ALLOCATION

- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains a pointer to the next block
- No compaction, external fragmentation
- Free space management system called when a new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

LINKED ALLOCATION EXAMPLE

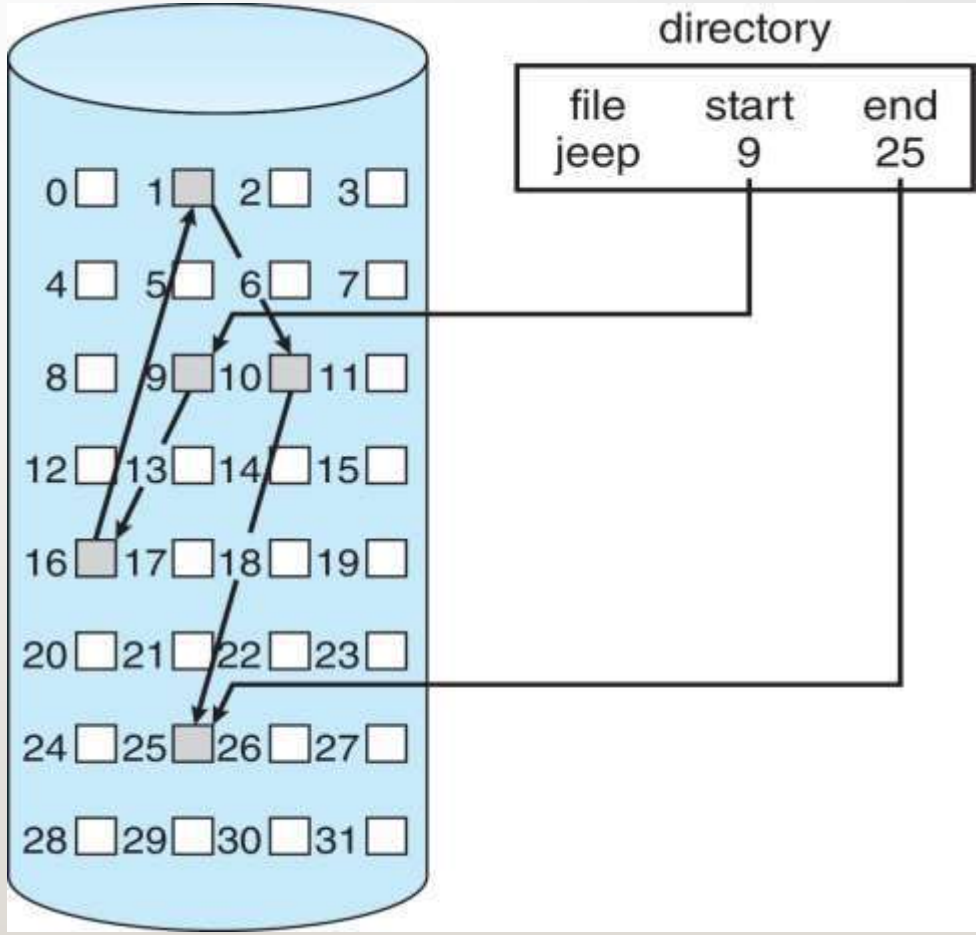
- Each file is a linked list of disk blocks.
- blocks may be scattered anywhere on the disk.

Mapping



Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file.

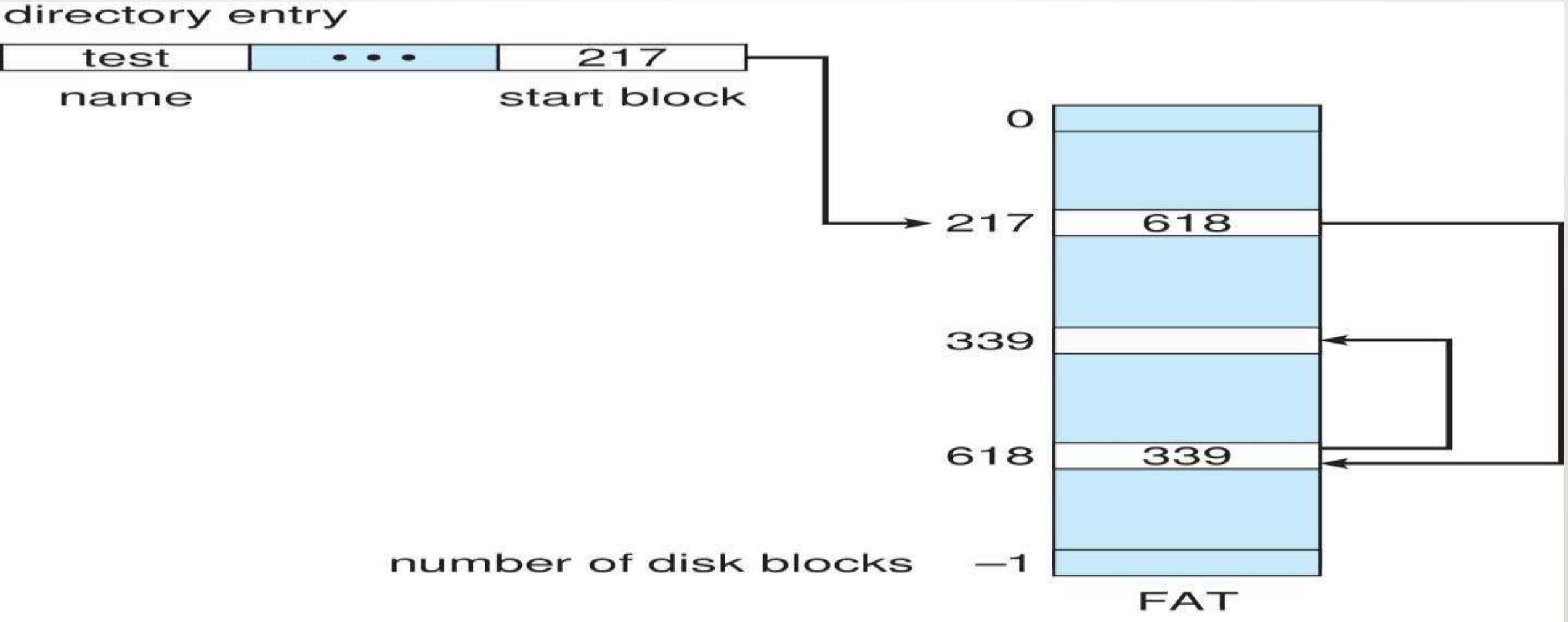
Displacement into block = R + 1



FAT ALLOCATION METHOD

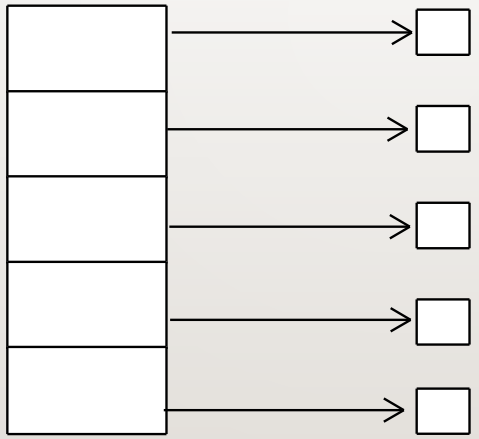
- The beginning of volume has a table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple.
- Used by DOS Operating System

FILE-ALLOCATION TABLE

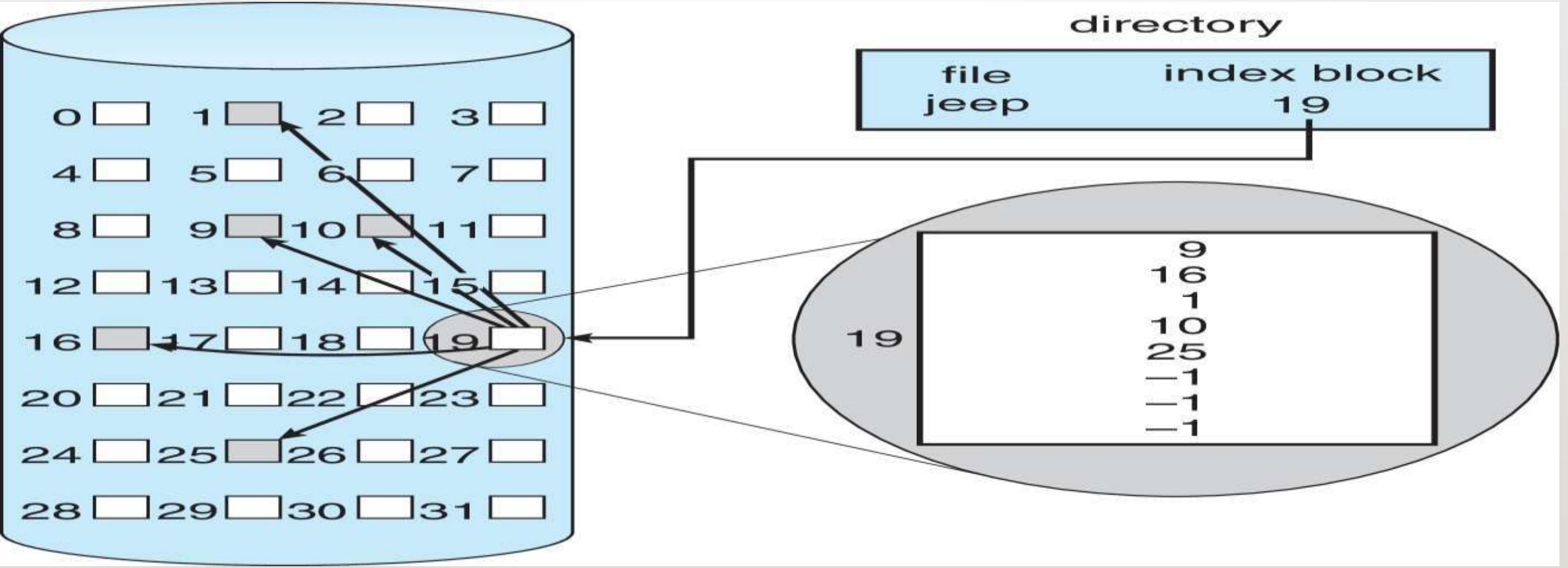


INDEXED ALLOCATION METHOD

- Each file has its own **index block**(s) of pointers to its data blocks
- Logical view

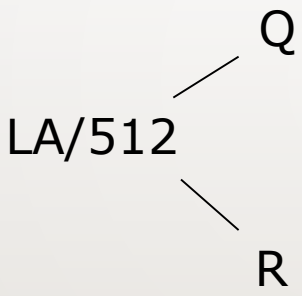


EXAMPLE OF INDEXED ALLOCATION



INDEXED ALLOCATION – SMALL FILES

- Need index table
- Random access
- Dynamic access without external fragmentation, but have the overhead of index block

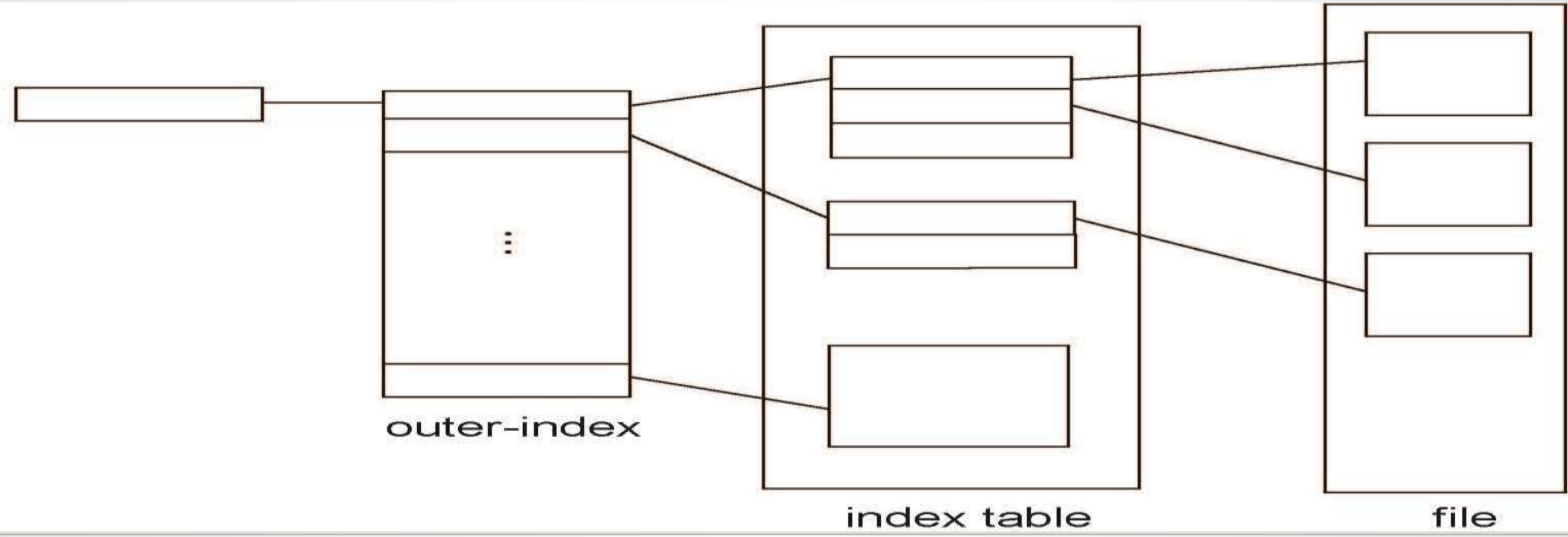


- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

INDEXED ALLOCATION – LARGE FILES

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
  - Linked scheme – Link blocks of index table (no limit on size)
  - Multi-level indexing

INDEXED ALLOCATION – TWO-LEVEL SCHEME



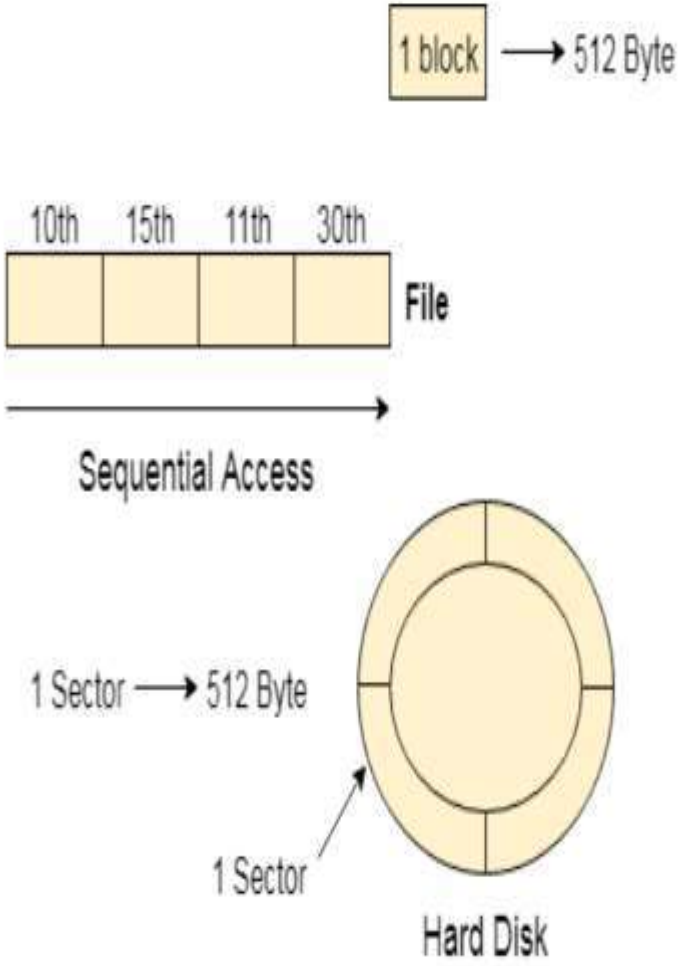
FILE ACCESS METHODS

- SEQUENTIAL ACCESS
- RANDOM/DIRECT ACCESS
- INDEXED ACCESS

SEQUENTIAL ACCESS

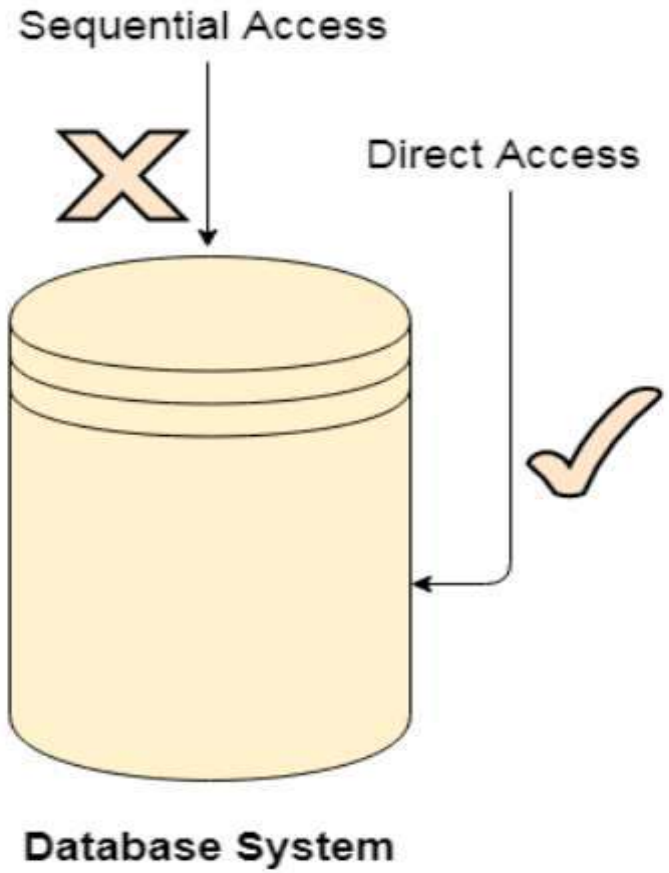
Most of the operating systems access the file sequentially.

- In other words, we can say that most of the files need to be accessed sequentially by the operating system.
- In sequential access, the OS read the file word by word.
- A pointer is maintained which initially points to the base address of the file. If the user wants to read first word of the file then the pointer provides that word to the user and increases its value by 1 word. This process continues till the end of the file.

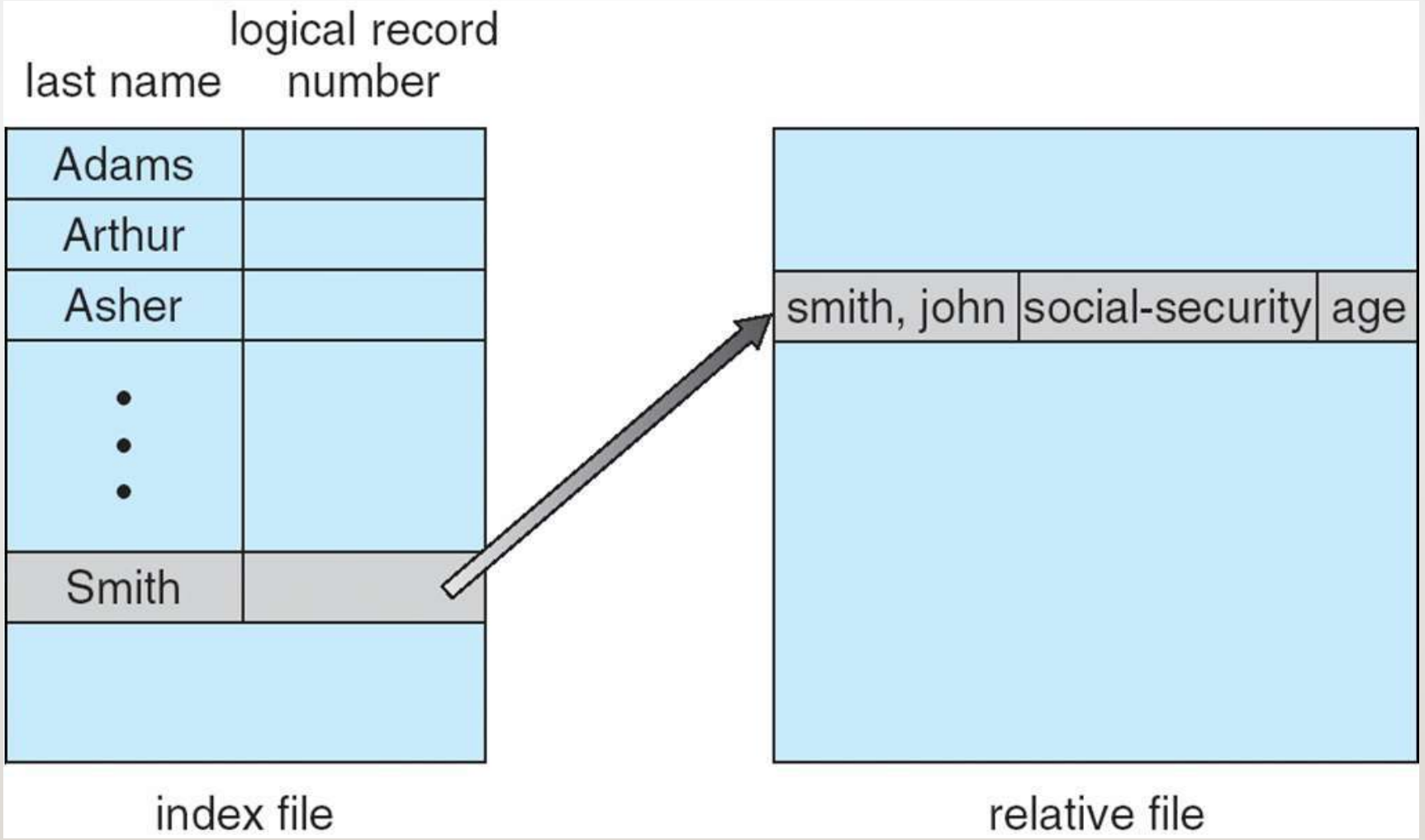


DIRECT ACCESS

- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

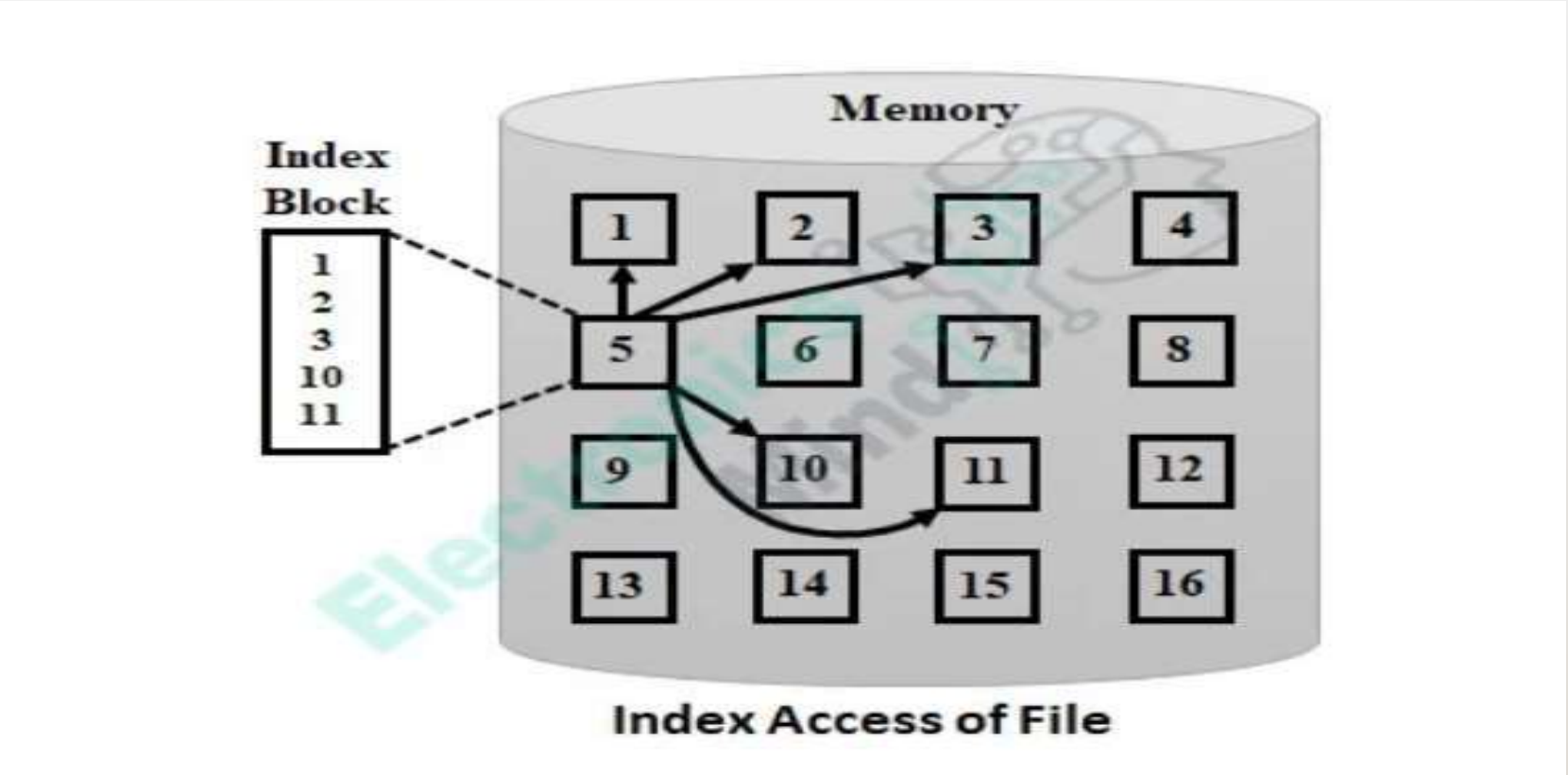


INDEX AND RELATIVE FILE ACCESS



INDEXED ACCESS METHOD

This method is typically an advancement in the direct access method which is the consideration of index. A particular record is accessed by browsing through the index and the file is accessed directly with the use of pointers or addresses present in the index as shown below.



Index Access of File

- **Sequential Access:** **read next**
  write next
  reset
  no read after last write
  (rewrite)
- **Direct Access –** file is fixed length logical records
  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

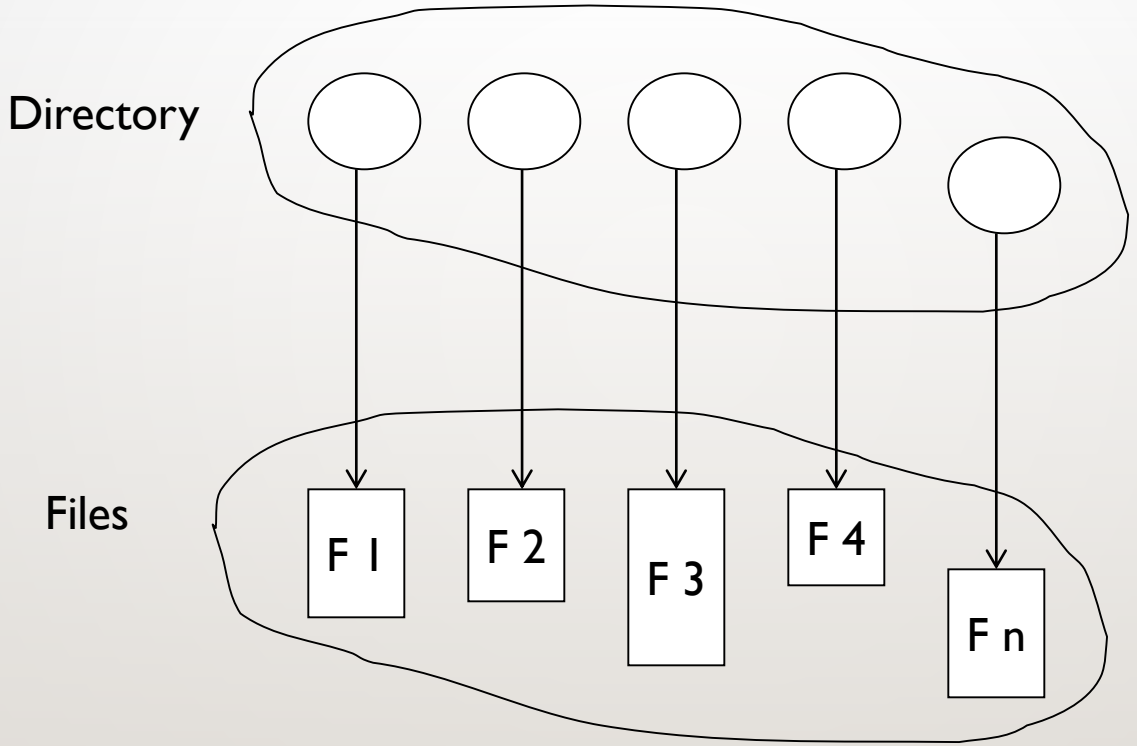
  n = relative block number
- Relative block numbers allow OS to decide where file should be placed

SIMULATION OF SEQUENTIAL ACCESS ON DIRECT-ACCESS FILE

sequential access	implementation for direct access
reset	$cp = 0;$
read next	*read cp*; $cp = cp + 1;$
write next	*write cp*; $cp = cp + 1;$

DIRECTORY STRUCTURE

- A collection of nodes containing information about all files

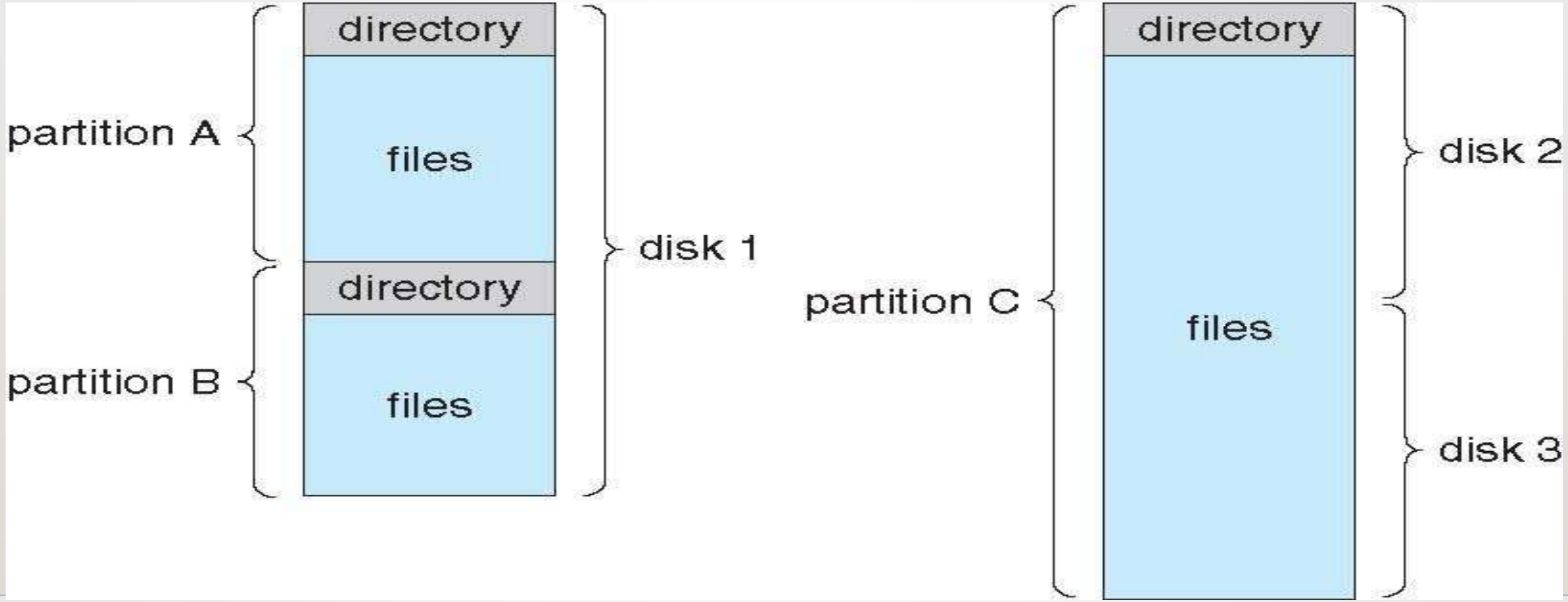


Both the directory structure and the files reside on disk

DISK STRUCTURE

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system.
- Partitions also known as minidisks, slices.
- Entity-containing file system is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

FILE-SYSTEM ORGANIZATION



TYPES OF FILE SYSTEMS

- We mostly talk of general-purpose file systems
- But systems frequently have many file systems, some general- and some special-purpose
- Consider **Solaris** has
  - **tmpfs** – memory-based volatile FS for fast, temporary I/O
  - **objfs** – interface into kernel memory to get kernel symbols for debugging
  - **ctfs** – contract file system for managing daemons
  - **lofs** – loopback file system allows one FS to be accessed in place of another
  - **procfs** – kernel interface to process structures
  - **ufs, zfs** – general purpose file systems

OPERATIONS PERFORMED ON DIRECTORY

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

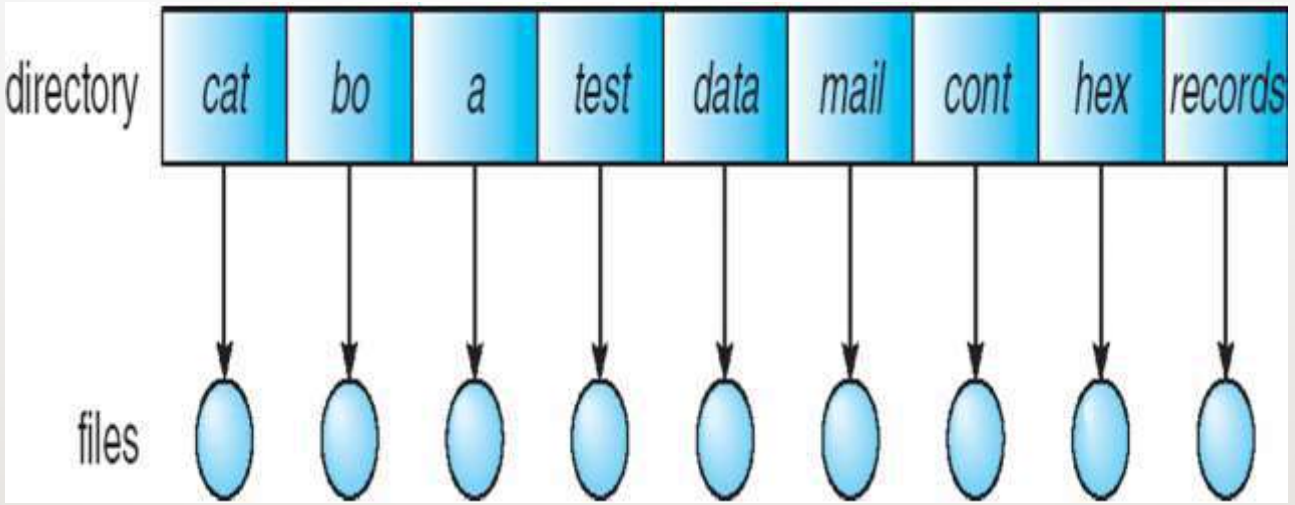
DIRECTORY ORGANIZATION

The directory is organized logically to obtain

- **Efficiency** – locating a file quickly
- **Naming** – convenient to users
  - Two users can have the same name for different files
  - The same file can have several different names
- **Grouping** – a logical grouping of files by properties, (e.g., all Java programs, all games, …)

SINGLE-LEVEL DIRECTORY

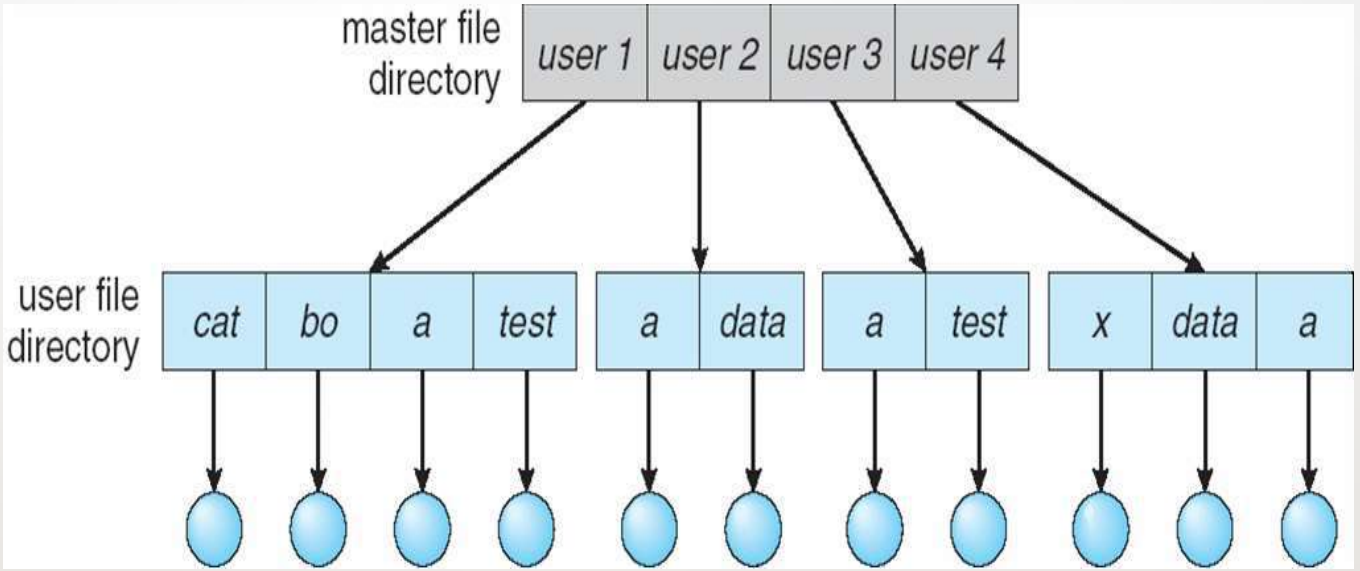
- A single directory for all users



- Naming problem
- Grouping problem

TWO-LEVEL DIRECTORY

Separate directory for each user



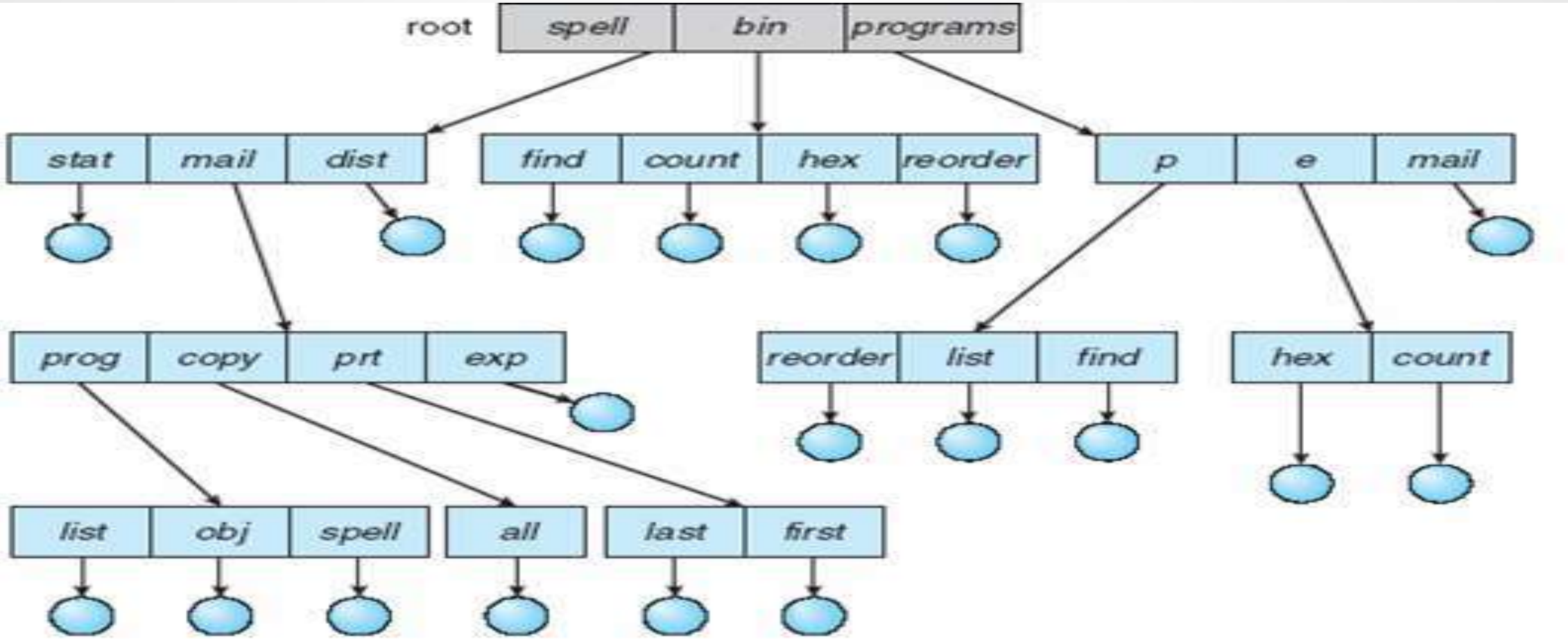
Path name

Can have the same file name for different user

Efficient searching

No grouping capability

TREE-STRUCTURED DIRECTORIES



TREE-STRUCTURED DIRECTORIES (CONT.)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd /spell/mail/prog**
  - **type list**

TREE-STRUCTURED DIRECTORIES (CONT)

- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file

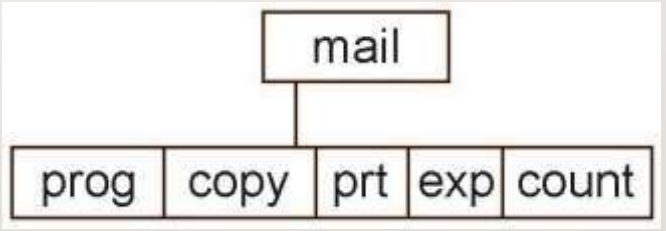
rm <file-name>

- Creating a new subdirectory is done in current directory

mkdir <dir-name>

Example: if in current directory **/mail**

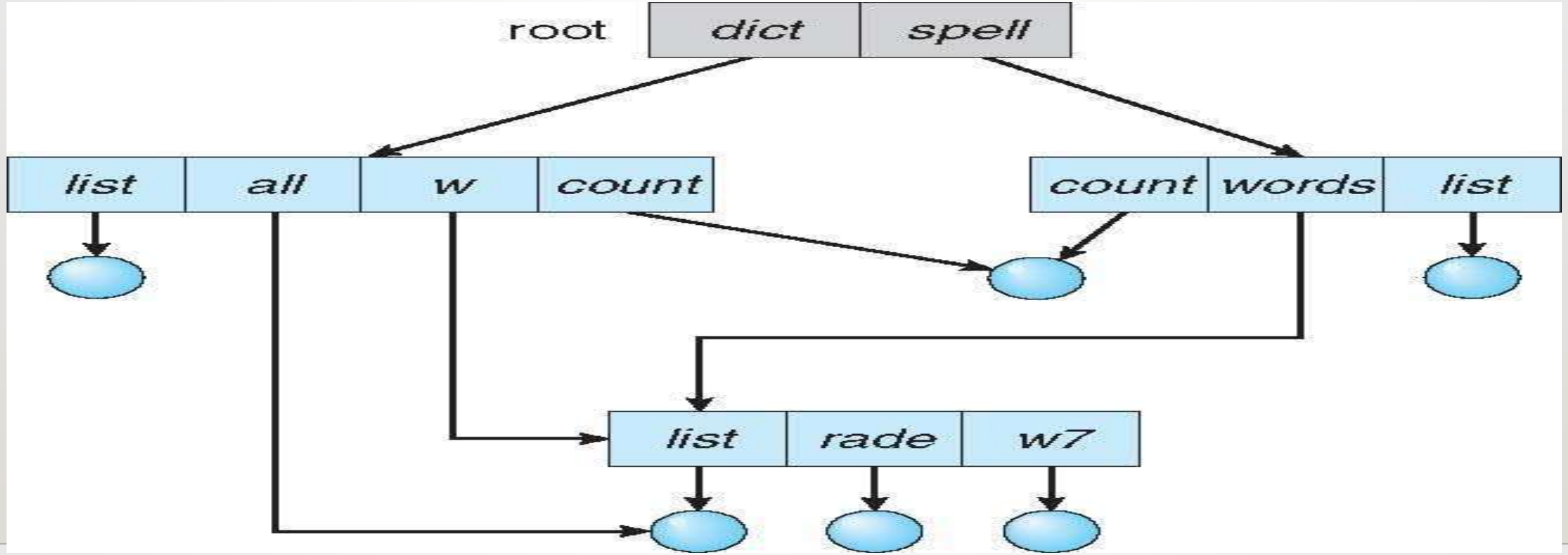
mkdir count



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

ACYCLIC-GRAPH DIRECTORIES

Have shared subdirectories and files



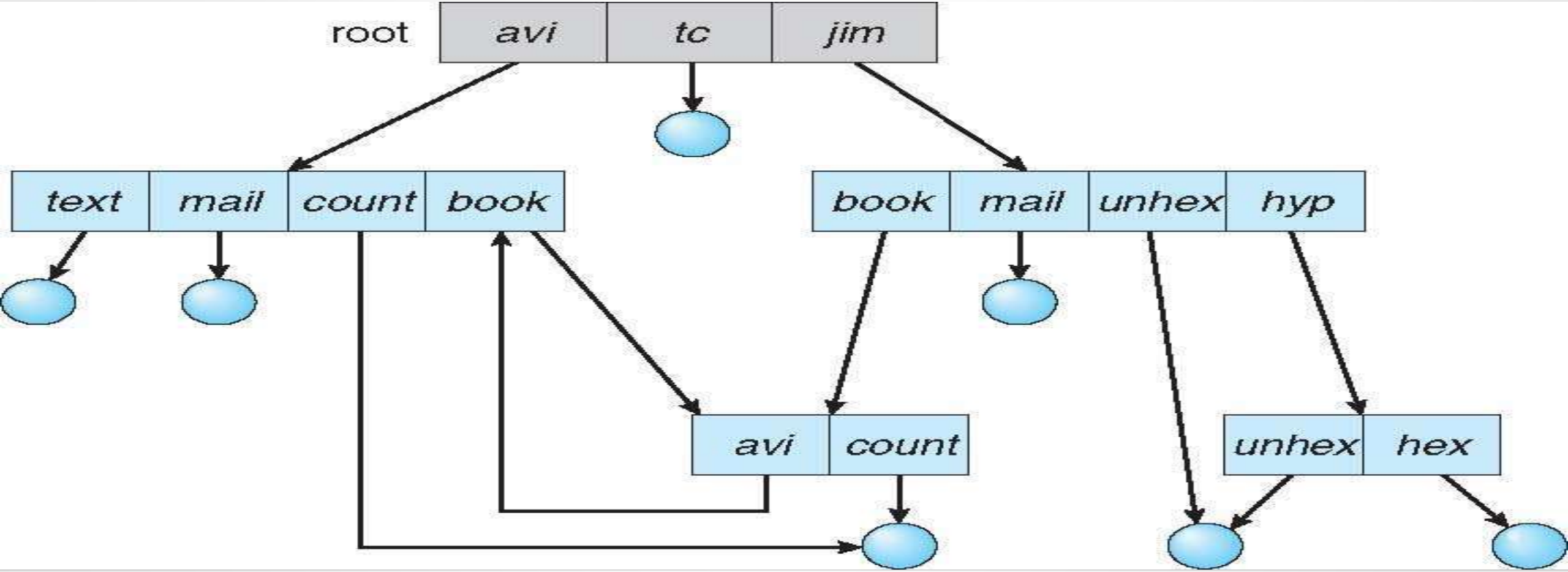
ACYCLIC-GRAPH DIRECTORIES (CONT.)

- Two different names (aliasing)
- If ***dict*** deletes ***list*** ⇒ dangling pointer

  Solutions:

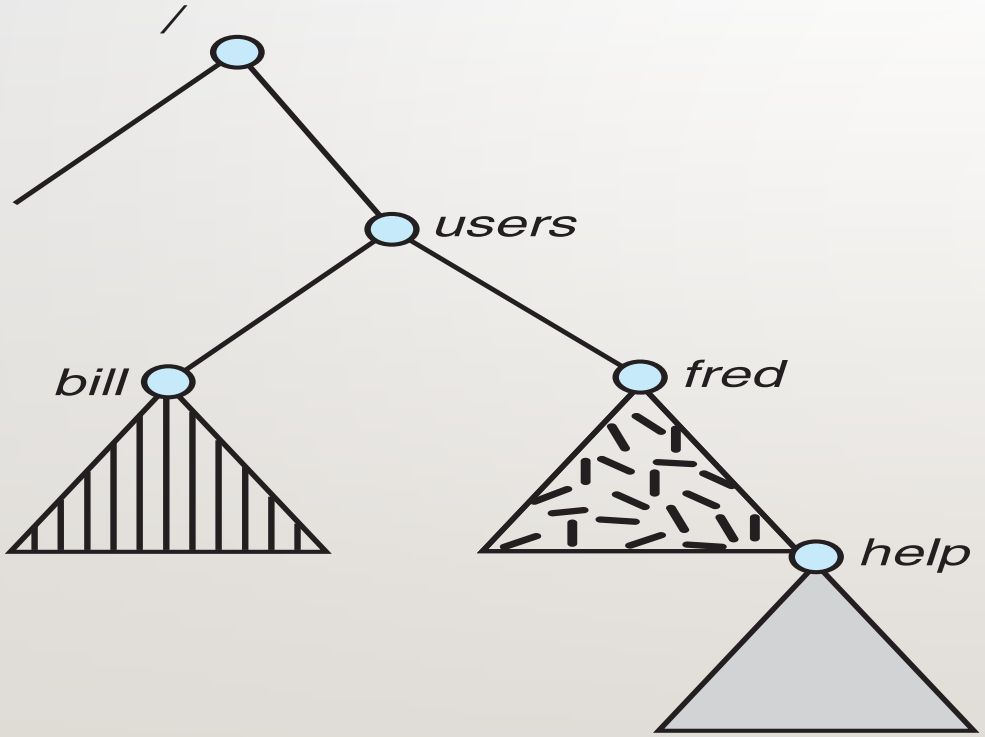
  - Back pointers, so we can delete all pointers
    Variable size records a problem
  - Back pointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

GENERAL GRAPH DIRECTORY

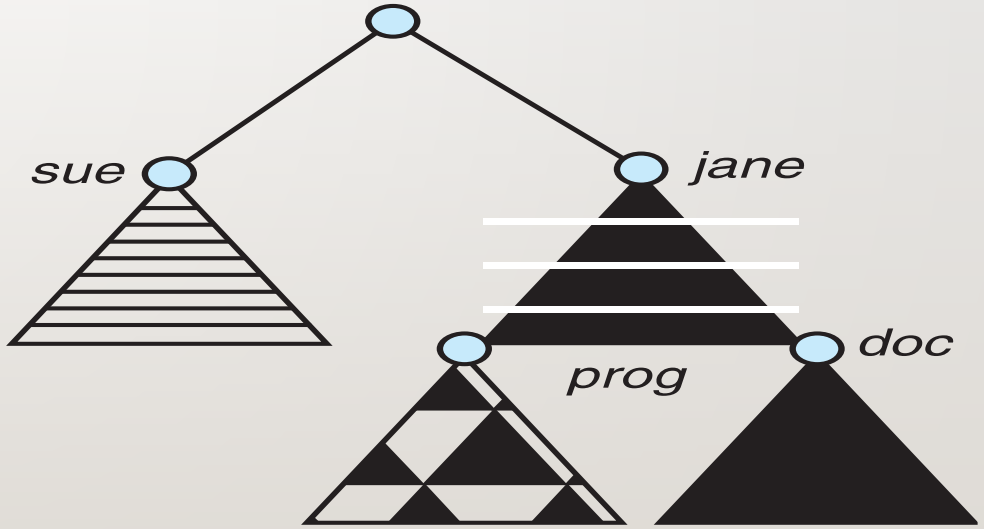


FILE SYSTEM MOUNTING

- A file system must be **mounted** before it can be accessed
- A unmounted file system (i.e., Fig)is mounted at a **mount point**

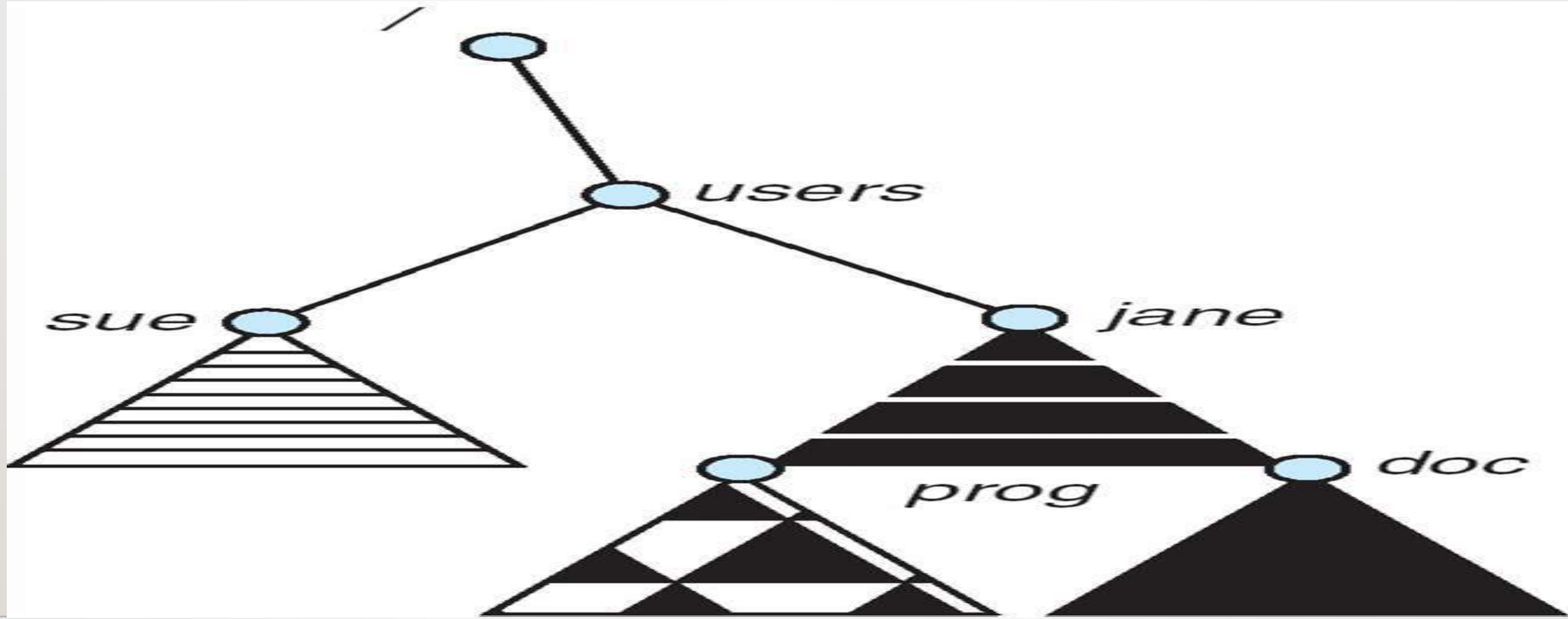


(a)



(b)

MOUNT POINT



FILE SHARING

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
    Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

FILE SHARING – REMOTE FILE SYSTEMS

- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - Standard operating system file calls are translated into remote calls

FILE SHARING – FAILURE MODES

- All file systems have failure modes
  - For example corruption of directory structures or other non-user data, called **metadata**
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve **state information** about status of each remote request
- **Stateless** protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

FILE SYSTEM IMPLEMENTATION

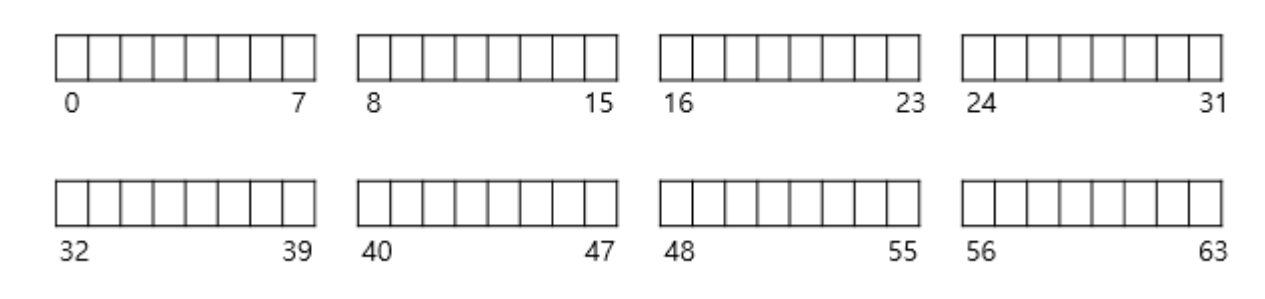
THE WAY TO THINK

- There are two different aspects to implementing a file system
  - **Data structures**
    - What types of on-disk structures are utilized by the file system to organize its data and metadata?
  - **Access methods**
    - How does it map the calls made by a process as open(), read(), write(), etc.
    - Which structures are read during the execution of a particular system call?

OVERALL ORGANIZATION

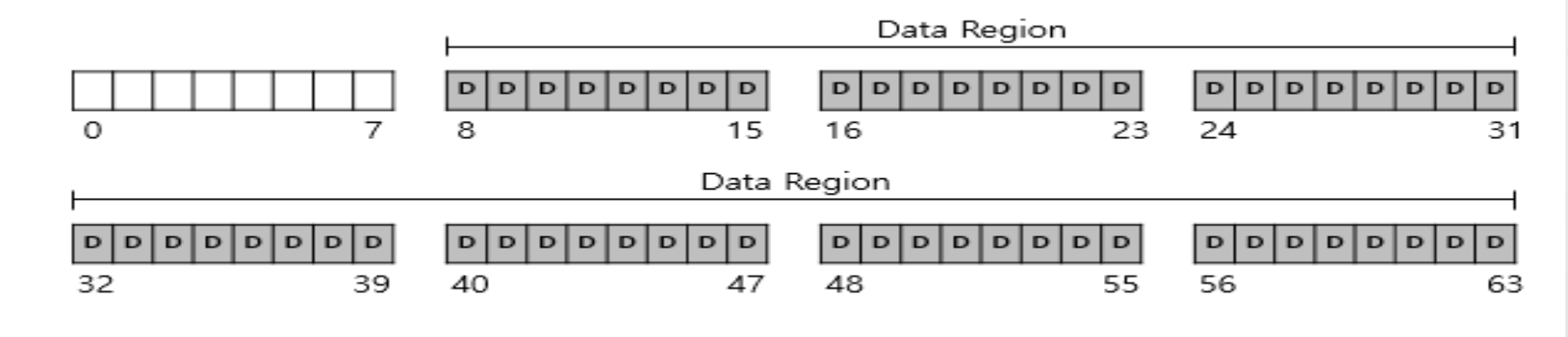
- Let's develop the overall organization of the file system data structure.

- Divide the disk into **blocks**.
  - Block size is 4 KB.
  - The blocks are addressed from *0 to N -1*.



DATA REGION IN FILE SYSTEM

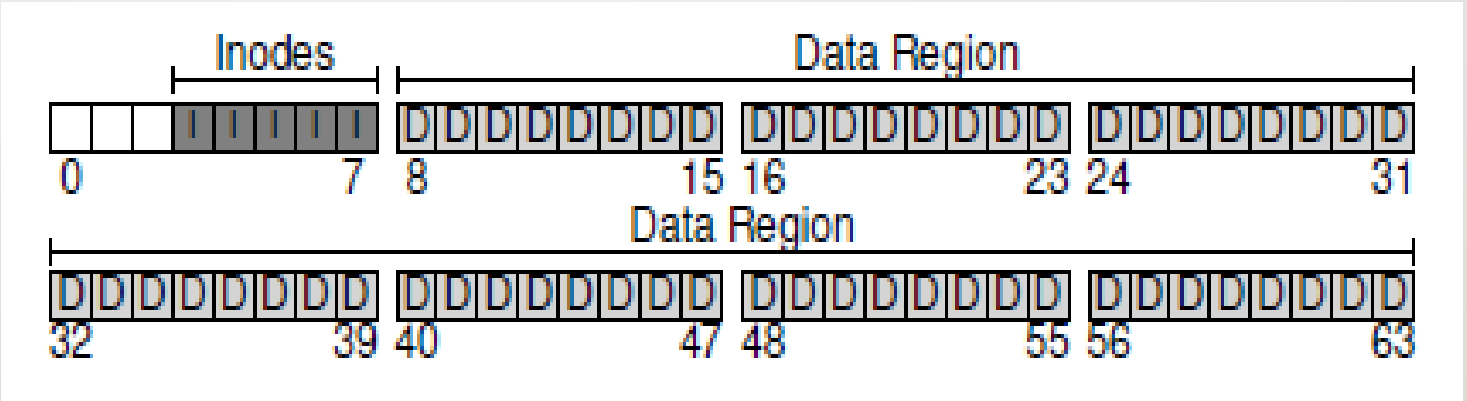
- Reserve **data region** to store user data



- The file system has to track which data blocks comprise a file, its size, its owner, etc.

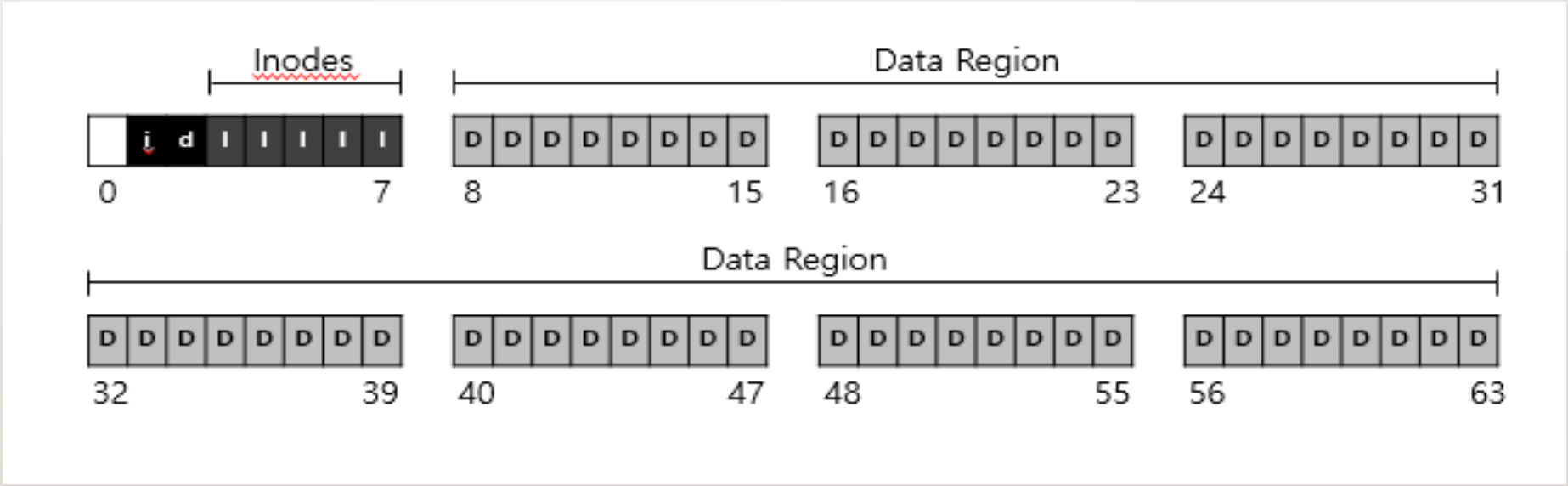
INODE TABLE IN THE FILE SYSTEM

- Reserve some space for **inode table**
  - This holds an array of on-disk inodes.
  - Ex) inode tables : 3 ~ 7, inode size : 256 bytes
    - 4-KB block can hold 16 inodes.
    - The file system contains 80 inodes. (maximum number of files)



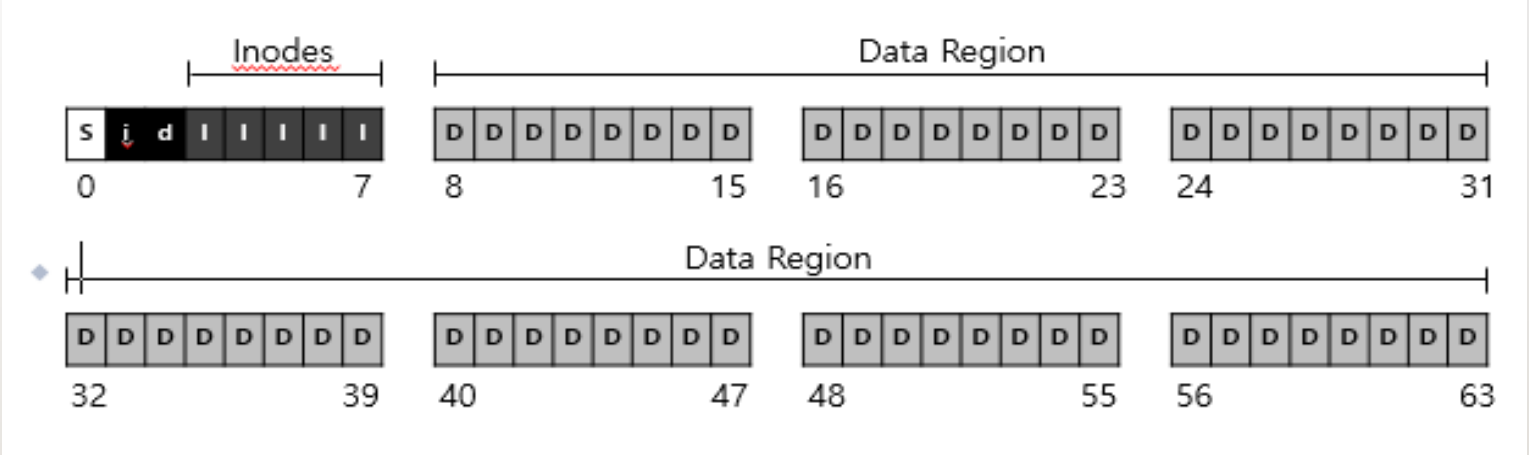
ALLOCATION STRUCTURES

- This is to track whether inodes or data blocks are free or allocated.
- Use **bitmap**, each bit indicates free(0) or in-use(1)
  - **data bitmap**: for data region for data region
  - **inode bitmap**: for inode table



SUPERBLOCK

- Super block contains this **information** for **particular file system**
  - Ex) The number of inodes, begin location of inode table. etc



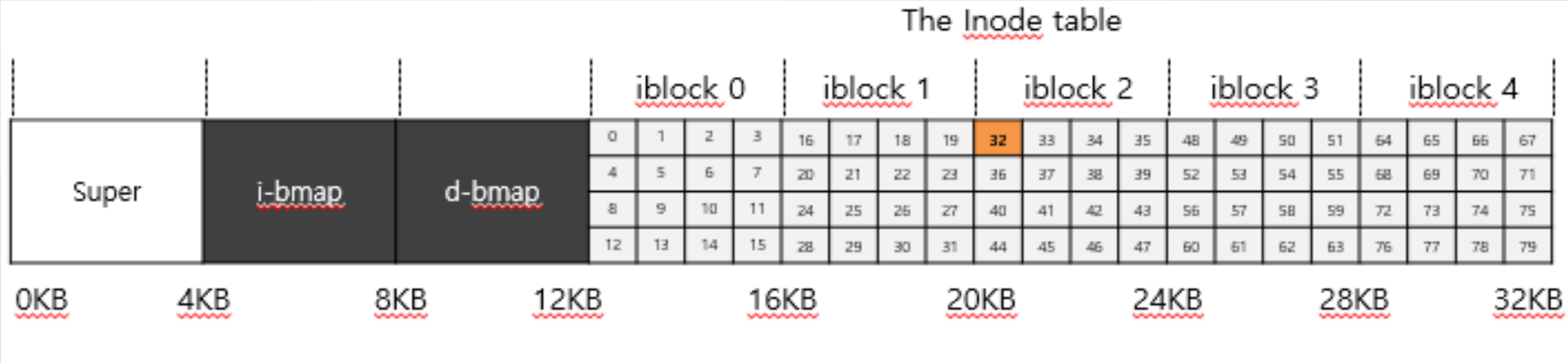
- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

FILE ORGANIZATION:THE INODE

- Each inode is referred to by inode number.
  - by inode number, File system calculate where the inode is on the disk.
  - Ex) inode number: 32
    - Calculate the offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB

FILE ORGANIZATION: THE INODE (CONT.)

- Disk are not byte addressable, sector addressable.
- Disk consist of a large number of addressable sectors, (512 bytes)
  - Ex) Fetch the block of inode (inode number: 32)
    - Sector address iaddr of the inode block:
    - blk : (inumber * sizeof(inode)) / blocksize
    - sector : (blk * blocksize) + inodeStratAddr) /sectorsize



FILE ORGANIZATION: THE INODE (CONT.)

- inode have all of the information about a file
  - File type (regular file, directory, etc.),
  - Size, the number of blocks allocated to it.
  - Protection information(who ones the file, who can access, etc).
  - Time information.
  - Etc.

FILE ORGANIZATION: THE INODE (CONT.)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

THE MULTI-LEVEL INDEX

- To support bigger files, we use a multi-level index.
- **Indirect pointer** points to a block that contains more pointers.
  - Inode have a fixed number of direct pointers (12) and a single indirect pointer.
  - If a file grows large enough, an indirect block is allocated, and the inode's slot for an indirect pointer is set to point to it.
    - $(12 + 1024)$ x 4 K or 4144 KB

THE MULTI-LEVEL INDEX (CONT.)

- Double indirect pointer points to a block that contains indirect blocks.
  - Allow file to grow with an additional 1024×1024 or 1 million 4KB blocks.
- Triple indirect pointer points to a block that contains double indirect blocks.
- Multi-Level Index approach to pointing to file blocks.
  - Ex) twelve direct pointers, a single and a double indirect block.
    - over 4GB in size $(12+1024+1024^2) \times 4KB$
- Many file system use a multi-level index.
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system.
  - Linux EXT4 use extents instead of simple pointers.

THE MULTI-LEVEL INDEX (CONT.)

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file system remain -50% full
Directories are typically small	Many have few entries; most have 20 or fewer

File System Measurement Summary

DIRECTORY ORGANIZATION

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files ."dot" for current directory and .."dot-dot" for parent directory
  - For example, `dir` has three files (`foo, bar, foobar`)

```
inum | reclen | strlen | name
   5     4     2         .
   2     4     3         ..
  12     4     4         foo
  13     4     4         bar
  24     8     7         foobar
```

FREE SPACE MANAGEMENT

- File system tracks which inode and data block are free or not.
- To manage free space, we have two simple bitmaps.
  - When a file is newly created, it is allocated inode by searching the inode bitmap and updating the on-disk bitmap.
  - Pre-allocation policy is commonly used to allocate contiguous blocks.

ACCESS PATHS: READING A FILE FROM DISK

- Issue an `open("/foo/bar", O_RDONLY)`,
  - Traverse the pathname and thus locate the desired inode.
  - Begin at the root of the file system `(/)`
    - In most Unix file systems, the root inode number is 2
  - Filesystem reads in the block that contains inode number 2.
  - Look inside of it to find a pointer to data blocks (contents of the root).
  - By reading in one or more directory data blocks, It will find the "foo" directory.
  - Traverse recursively the path name until the desired inode ("bar")
  - Check final permissions, allocate a file descriptor for this process, and return the file descriptor to the user.

ACCESS PATHS: READING A FILE FROM DISK (CONT.)

- Issue read() to read from the file.
  - Read in the first block of the file, consulting the inode to find the location of such a block.
    - Update the inode with a new last accessed time.
    - Update in-memory open file table for file descriptor, the file offset.

- When file is closed:
  - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No dis I/Os take place.

ACCESS PATHS: READING A FILE FROM DISK (CONT.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read			read				
				read						
					read		read			
read()					read					
					write			read		
read()					read					
					write				read	
read()					read					
					write					read

File Read Timeline (Time Increasing Downward)

ACCESS PATHS: WRITING TO DISK

- Issue write() to update the file with new contents.
- File may allocate a block (unless the block is being overwritten).
  - Need to update data block, and data bitmap.
  - It generates five I/Os:
    - one to read the data bitmap
    - one to write the bitmap (to reflect its new state to disk)
    - two more to read and then write the inode
    - one to write the actual block itself.
  - To create files, it also allocates space for the directory, causing high I/O traffic.

ACCESS PATHS: WRITING TO DISK (CONT.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

File Creation Timeline (Time Increasing Downward)

CACHING AND BUFFERING

- Reading and writing files are expensive, incurring many I/Os.
  - For example, long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one read its data.
    - perform hundreds of reads just to open the file.

- To reduce I/O traffic, file systems aggressively use system memory(DRAM) to cache.
  - Early file systems use fixed-size cache to hold popular blocks.
    - Static partitioning of memory can be wasteful;
  - Modem systems use a dynamic partitioning approach, a unified page cache.
- Read I/O can be avoided by large cache.

CACHING AND BUFFERING (CONT.)

- Write traffic has to go to disk for persistence, Thus, cache does not reduce write I/Os.
- File system uses write buffering for write performance benefits.
  - delaying writes (file system batch some updates into a smaller set of I/Os).
  - By buffering a number of writes in memory, the file system can then schedule the subsequent I/Os.
  - By avoiding writes
- Some applications force flush data to disk by calling fsync() or direct I/O.

PROTECTION

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

ACCESS LISTS AND GROUPS

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

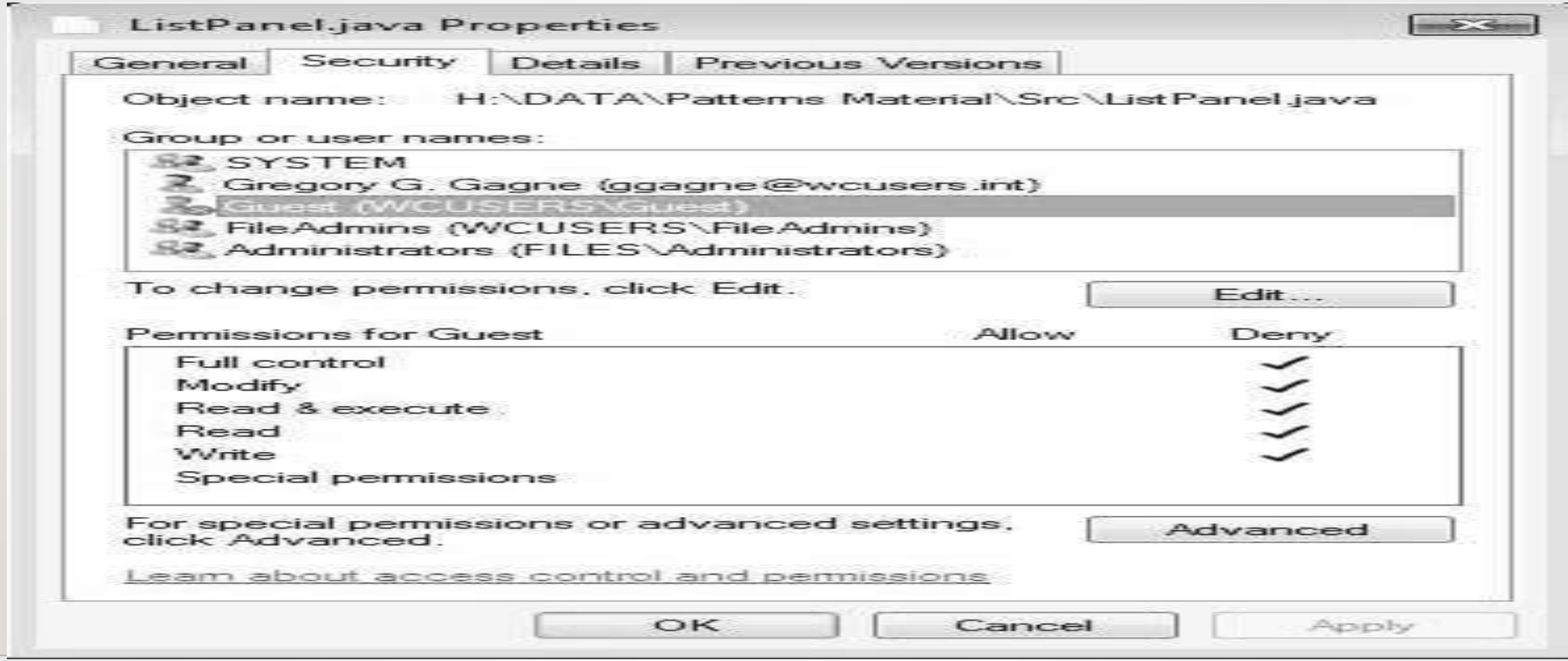
			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	1 1 0
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

Attach a group to a file

```
chgrp      G      game
```

WINDOWS 7 ACCESS-CONTROL LIST MANAGEMENT



UNIX DIRECTORY LISTING

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

THANK YOU

Team - Operating System