# File System Implementation

*Instructor: Dr Tarunpreet Bhatia*

*Assistant Professor*

*CSED, TIET*

## Disclaimer

This is NOT A COPYRIGHT MATERIAL

***Content has been taken mainly from the following books***:

Operating Systems Concepts By Silberschatz & Galvin,

Operating Systems: Internals and Design Principles By William Stallings

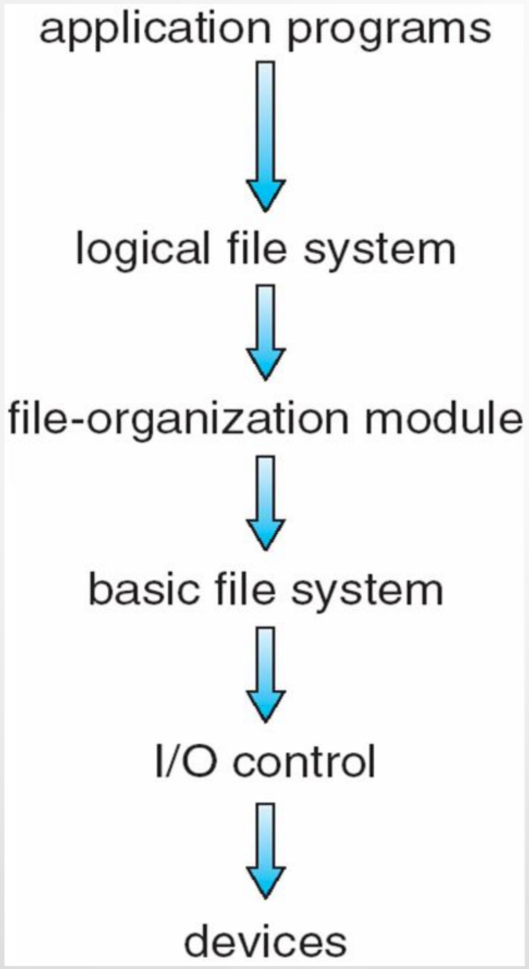
## Topics

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods

# File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device

## Layered File System



## File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
    - Given commands like "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like "retrieve block 123" translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks

## File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance

# File System Layers (Cont.)

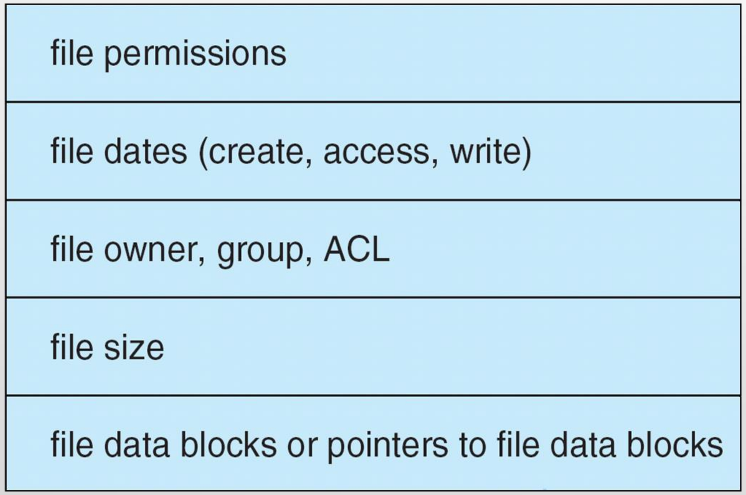
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS;  Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

# File-System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (**superblock, master file table**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

# File-System Implementation (Cont.)

- Per-file **File Control Block** (**FCB**) contains many details about the file
  - inode number, permissions, size, dates
  - NTFS stores into in master file table  using relational DB structures

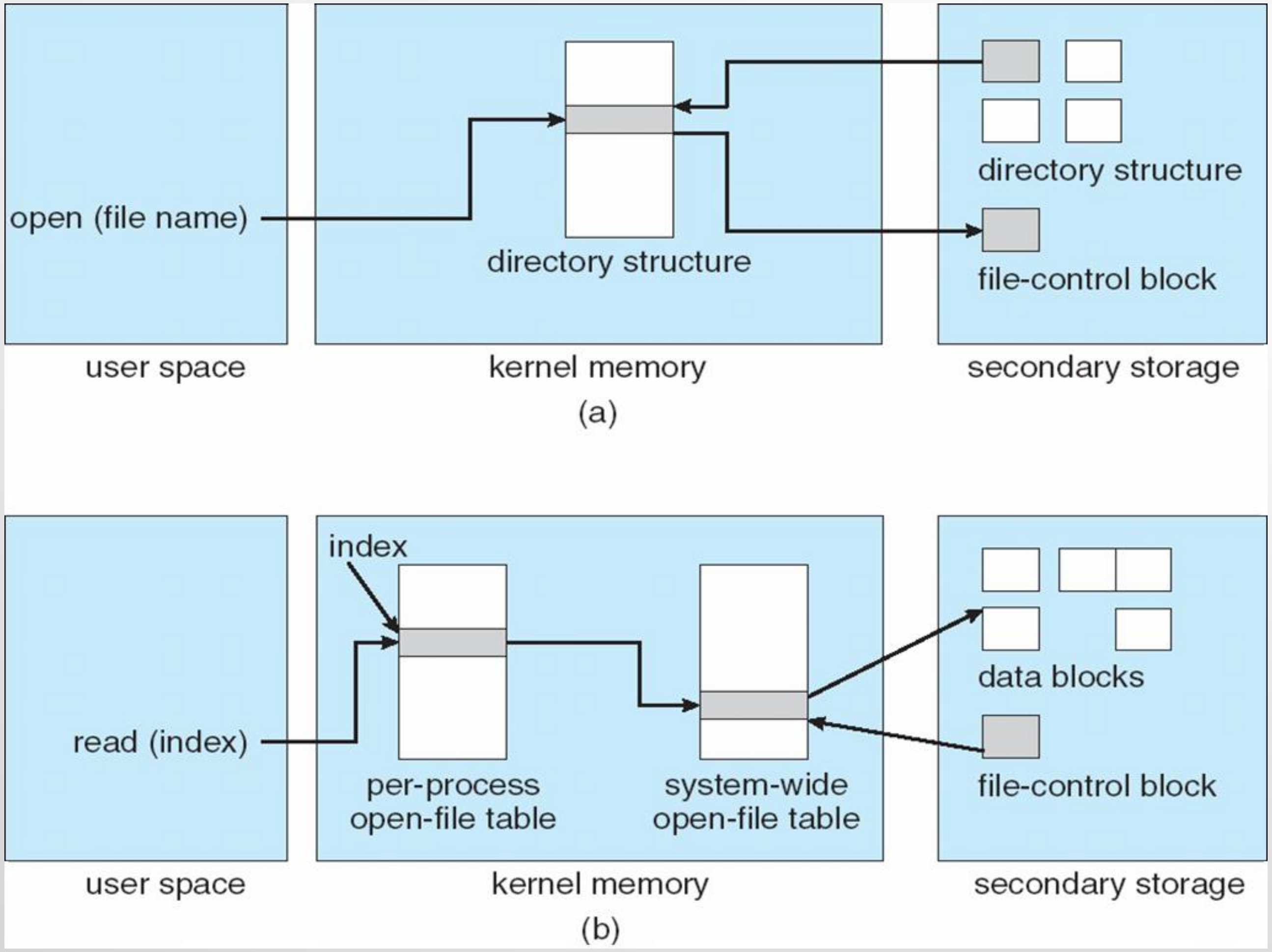


| |
|---|
| file permissions |
| file dates (create, access, write) |
| file owner, group, ACL |
| file size |
| file data blocks or pointers to file data blocks |

## In-Memory File System Structures

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

# In-Memory File System Structures



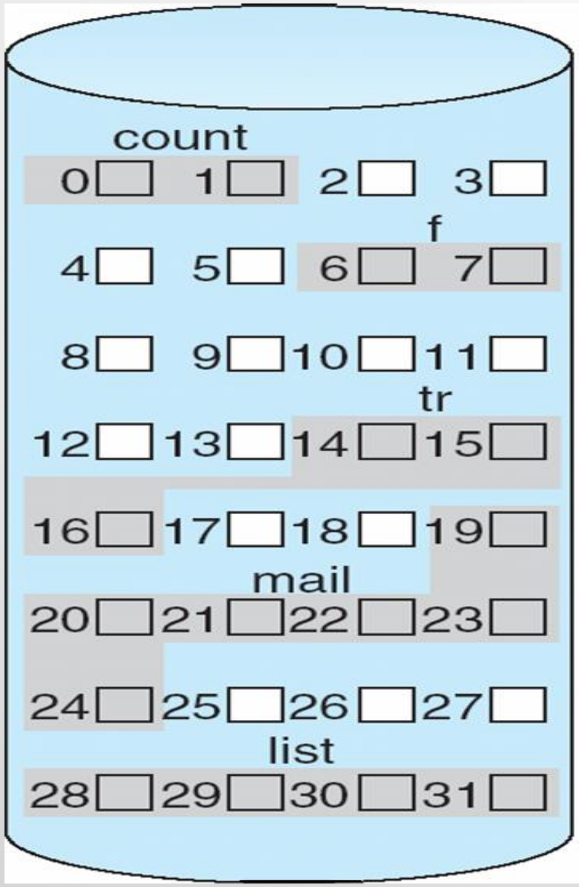
# Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

# File Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies 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 for file, knowing file size, external fragmentation, need for compaction

## Contiguous Allocation



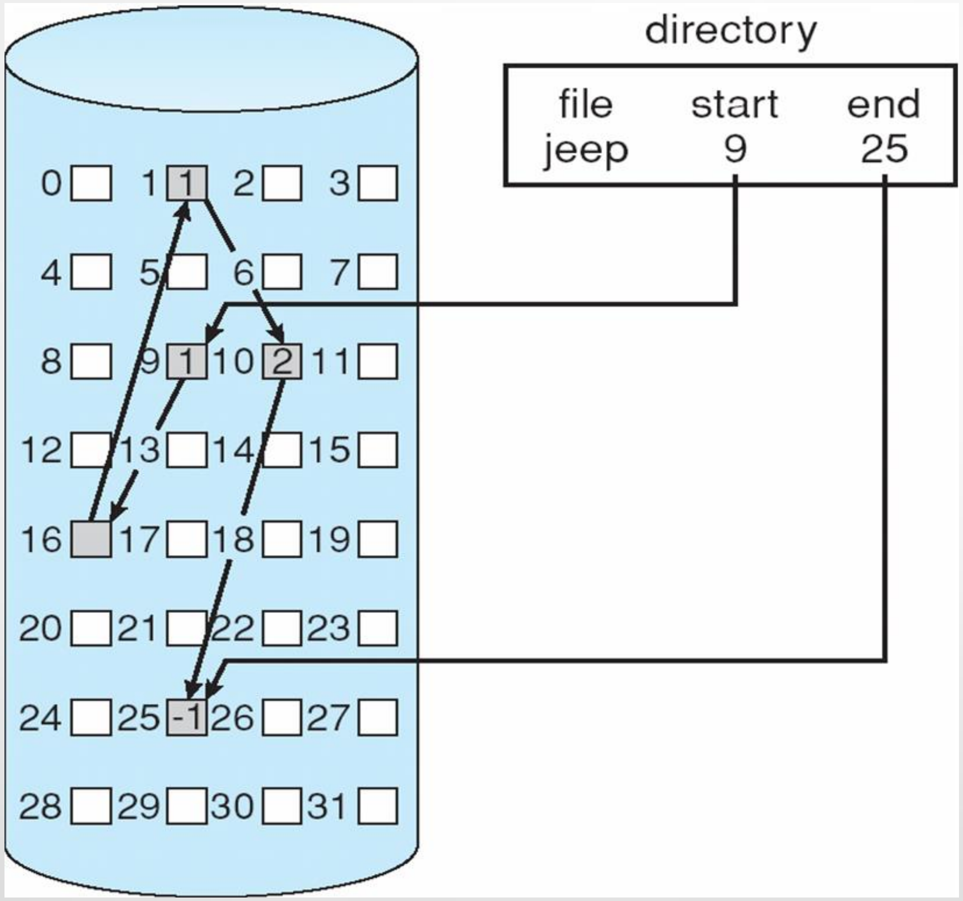
directory

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

# Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation and no compaction
  - Each block contains pointer to next block
  - Free space management system called when new block needed
  - File can continue to grow as long as free blocks are available.
  - Used effectively only for sequential access not direct acces to files.
  - Space required for pointers.
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem

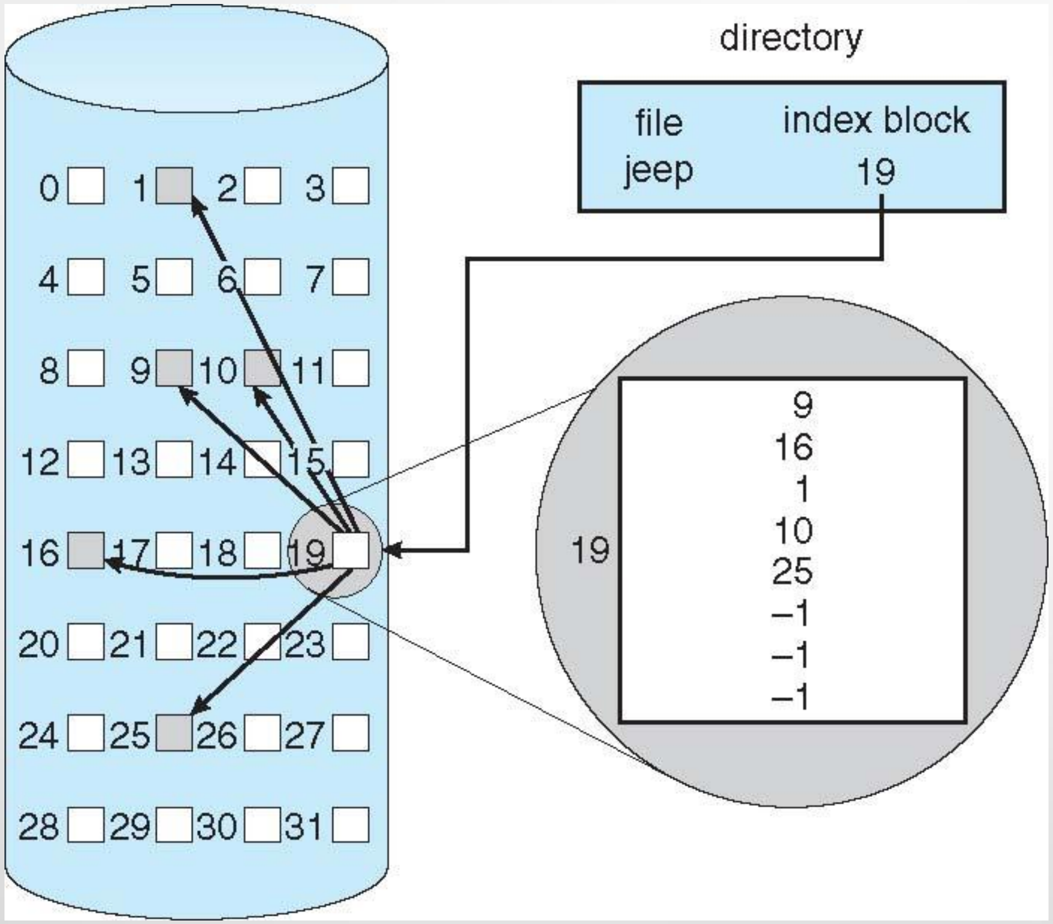
# Linked Allocation



## Indexed Allocation

- Each file has its own **index block**(s) of pointers to its data blocks
- Need index table
- Random access
- Direct access without external fragmentation, but have overhead of index block

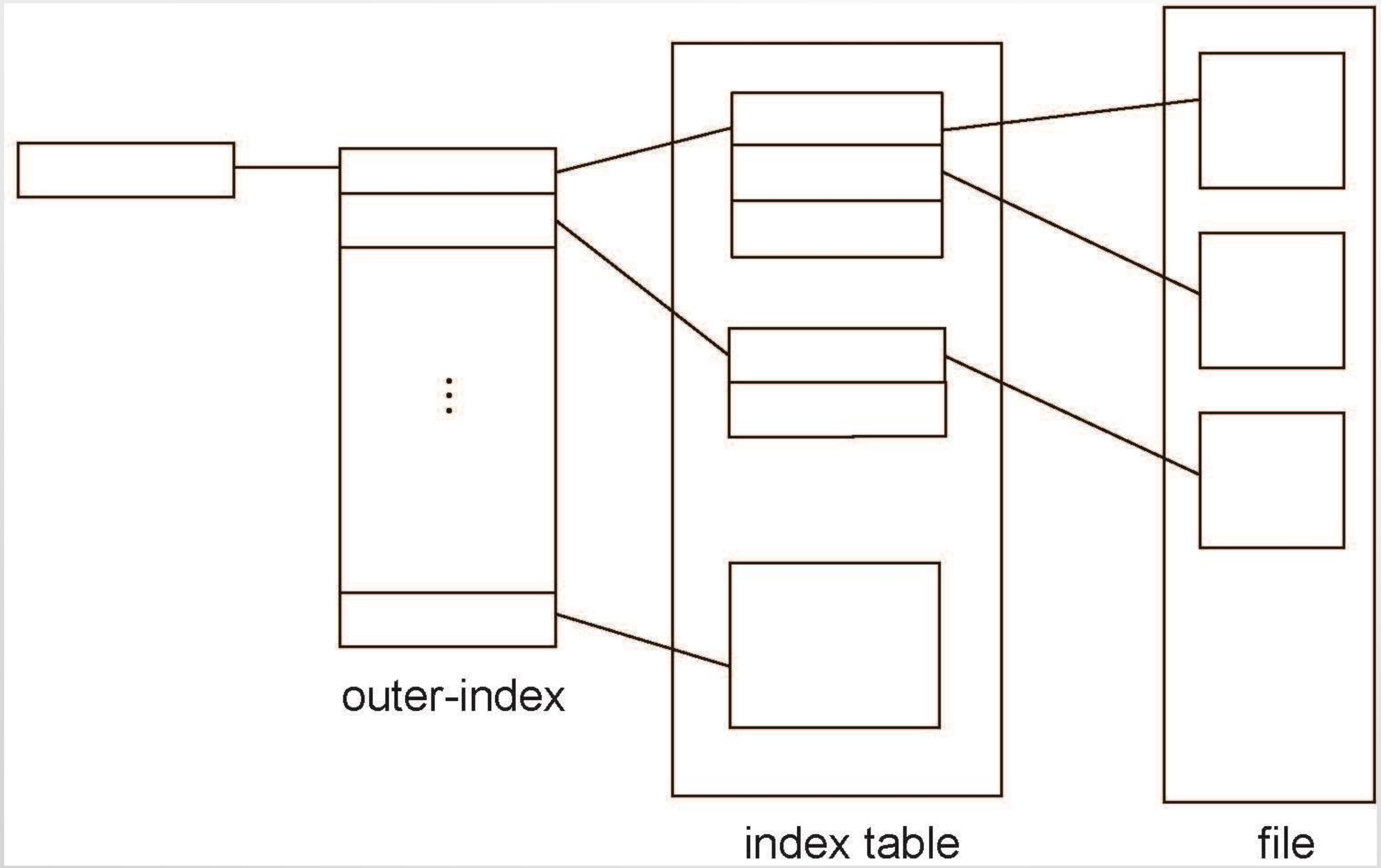
# Example of Indexed Allocation



## Mechanisms to deal with index block size

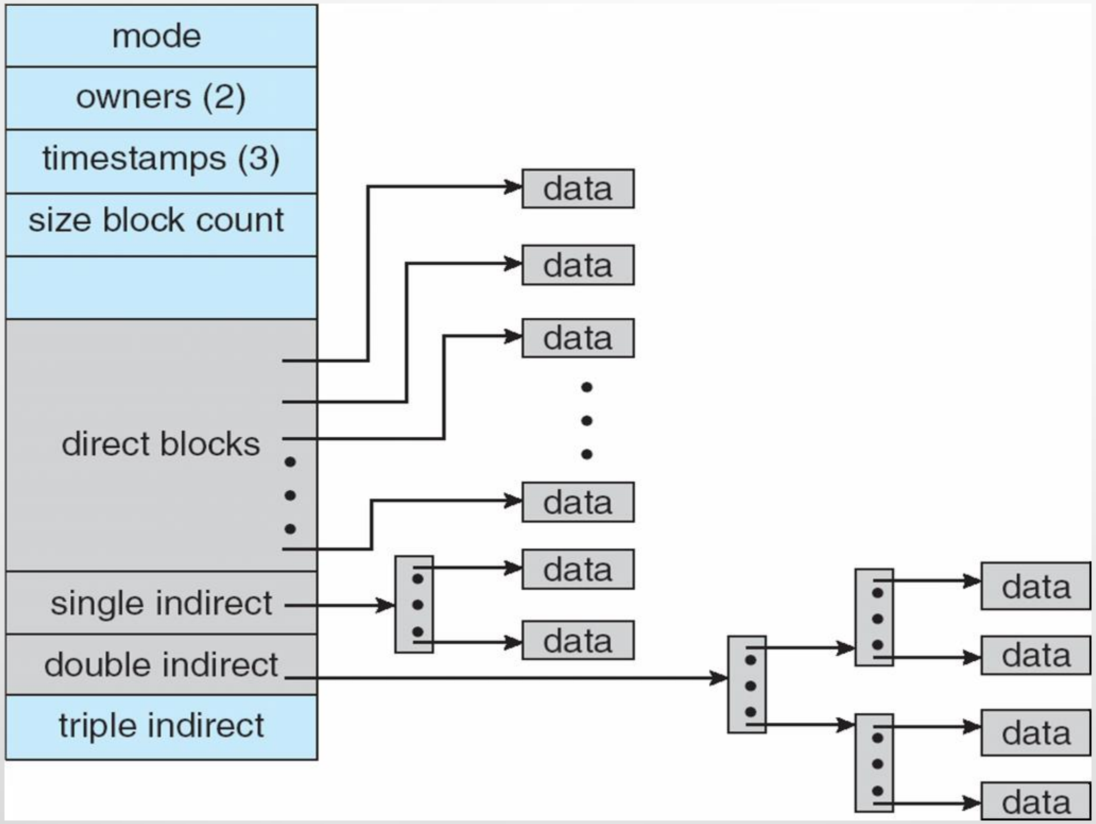
- Linked scheme
- Multilevel index
- Combined scheme

# Multilevel Indexed Allocation – Mapping (Cont.)



## Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

## Performance

- Best method depends on file access type
  - Contiguous great for sequential and direct
- Linked good for sequential, not direct
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

## Question 1

Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

## Question 2

Consider a system in which a directory entry can store up to 16 disk block address. For files not larger than 16 blocks, the 16 addresses serve as the file's index table. For files larger than 16 blocks, the address point to indirect blocks which in turn point to 256 file blocks each. A block is 1024 bytes. How big can a file be?

## Question 3

Consider a file currently consisting of 101 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations (read and write) are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In calculating I/O operations, if the location of the first block of a file changes, do not include the output operations needed to rewrite that revised information to the directory entry on the disk. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory and 51th block is middle block.

a. The block is added at the beginning.

b. The block is added after the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

# Question 4

A file system uses 256-byte physical blocks and block numbering starts from 1. Now, assume that the last physical block read and the directory entries (corresponding to contiguous, linked and indexed strategies) are residing in the main memory. Further, assume the directory entries in indexed allocation contain pointers for 127 file blocks with an additional pointer to the next index block. In addition to the last block read, assume that the index block that contains pointer to the last block read resides in the main memory.

How many physical blocks must be read for the cases given below in context of contiguous, linked and indexed allocation??

i) Last block read: 100; block to be read: 600

ii) Last block read: 500; block to be read: 200

iii) Last block read: 20; block to be read: 21

iv) Last block read: 21; block to be read: 20

# Question 5

Consider a file system which uses an inode table to organize the files on disk. Each inode consists of a user id (2 bytes), three time stamps (4 bytes each), protection bits (2 bytes), a reference count (2 bytes), a file type (2 bytes) and the file size (4 bytes). The file system also stores the first 436 data bytes of each file in the inode table. Additionally, the inode contains 64 direct data block addresses as well as a single indirect index block and a double indirect index block. Each index entry takes 4 bytes, index table takes up an entire disk block, and a disk block is of 1024 bytes. Assume directory entry is already in main memory. Answer the following questions and show intermediate calculations.

- What is the maximum data size (in MB up-to 2 decimal places) for a file in this system?
- How many disk accesses does it take to read data at relative location 2999885 within a file, assuming no caching?