Part 8: File Systems

- File System Structure
- Logical File System
  - Files: File Attributes, File Types, File Structure, File Access Methods, File Operations
  - Directories: Directory Structure, Directory Organisation, Directory Operations
  - File Protection in UNIX
  - In-memory File System Data Structures
- File Organization Module
  - Allocation Methods: Contiguous Allocation, Linked Allocation (& FAT), Indexed Allocation (& inode)
  - Disk-Space Management: Block Size, Keeping Track of Free Blocks

File System Structure

- A file system is generally composed of many different levels. For example:
  - *Logical File System*: manages directory structure, responsible for file creation, access, deletion, protection and security.
  - *File-Organisation Module*: allocates storage space for files, translates logical block addresses to physical block addresses, and manages free disk space.
  - *Basic File System*: manages buffers and caches issues generic commands to the appropriate device driver to read and write physical blocks on the disk.
  - *I/O Control*: consists of device drivers and interrupt handlers to transfer information between memory and the disk system.

File Attributes

- A file may have the following attributes
  - *Name* - only information kept in human-readable form.
  - *Type* - needed for systems that support different types.
  - *Location* - pointer to file location on device.
  - *Size* - current file size
  - Protection - controls who can do reading, writing, executing.
  - *Time, Date, and User Identification* - data for protection, security, and usage monitoring.
- These information about files are kept in the directory structure, which is maintained on the disk.

File Types

- A file has a contiguous logical address space which can store many different types of information:

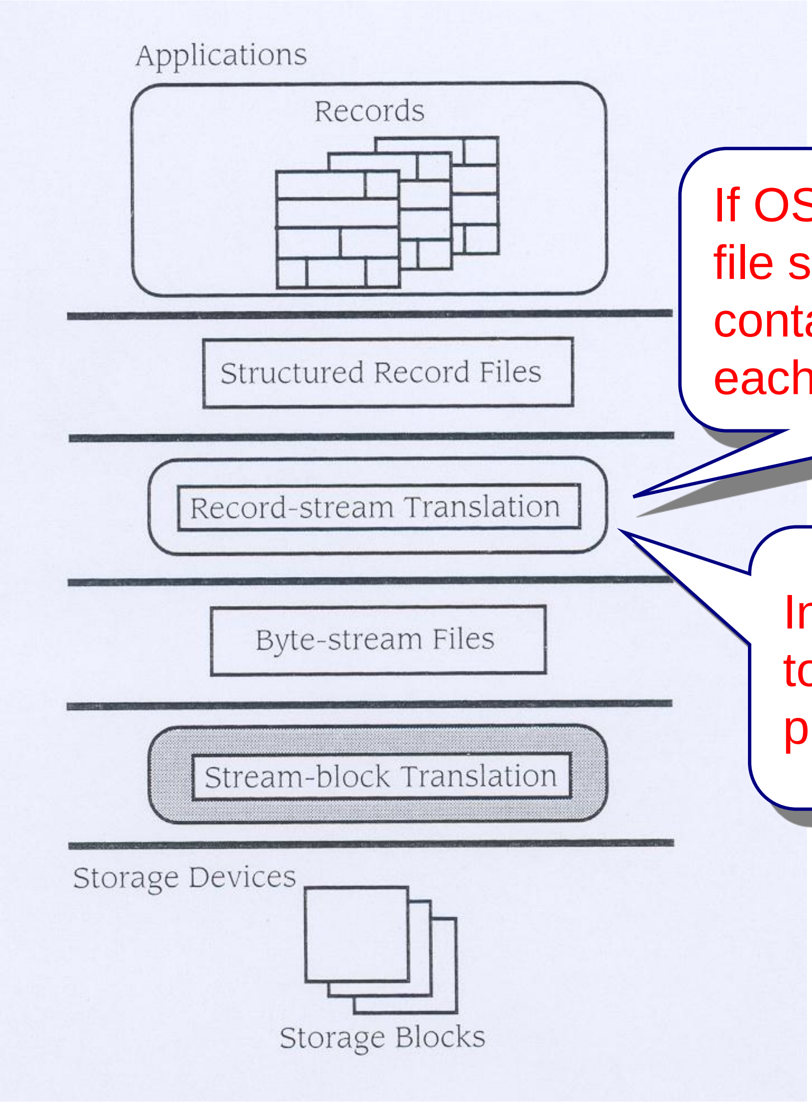
File Type	Usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	complied, machine language, not linked
Source code	c, p, pas, 177, asm, a	source code in various languages
Batch	bat, sh	commands to the command interpreter
Text	txt, doc	textual data documents
Word processor	wp, tex, rrf, etc.	various word-processor formats
Library	lib, a	libraries of routines
Print or view	ps, dvi, gif	ASCII or binary file
Archive	arc, zip, tar	related files grouped into one file, sometimes compressed.

File Structure

- A file consists of a collection of records. These records can be organised or structured to facilitate file access.

- Unstructured: Sequence of Bytes.
  - A file is a stream of bytes. Each byte is individually addressable from the beginning of the file.
  - Used by UNIX and MSDOS (and **assumed in the following discussions**)

File Structure (Cont.)

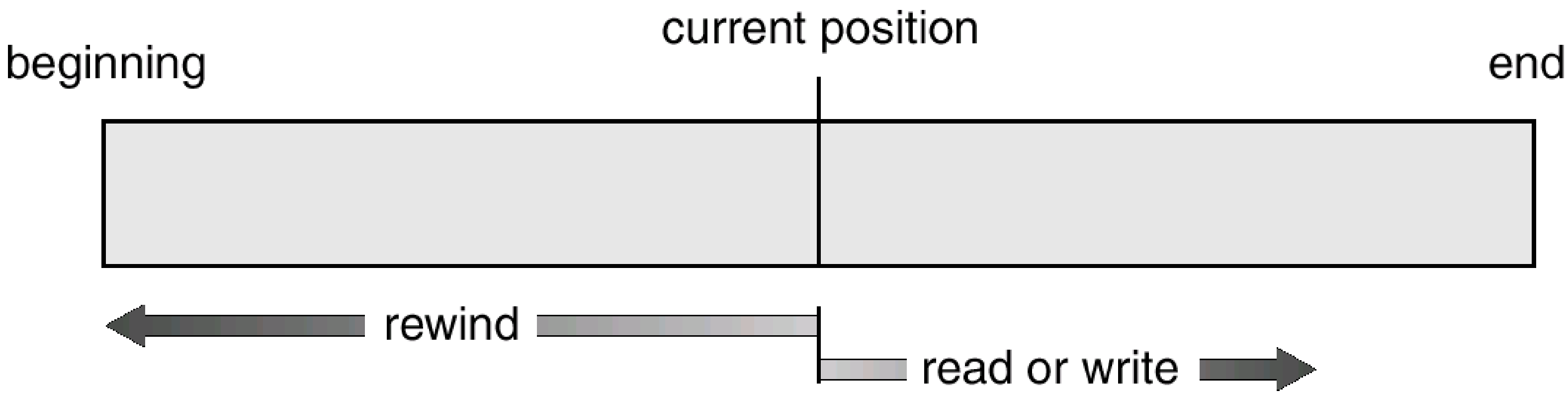


If OS supports multiple file structures, it should contain code to do this for each structure

In general, this task is left to the application programs

File Access Methods

- Sequential Access: Information in the file is processed in order, one record after the other
  - Operations:
    - *read* record from the next position
    - *write* record to the next position
    - *reset* to the beginning



File Access Methods (Cont.)

- Direct Access
  - Based on disk model of a file
  - Records of a file can be read in any order (by referencing record number or the key)
  - Operations
    - *read* record at *position* n
    - *write* record at *position* n
    - *position* to n

File Operations

Commands	*Explanation*
Create	allocate disk space; create directory entry with file attributes.
Delete	delete the corresponding directory entry; deallocate disk space.
Open	search the directory structure for file entry; move the content to memory (put in the *open file table*). <u>Information Associated with an open file:</u> - *Current Position Pointer* - *File Open Count* - *Disk Location*
Close	move the content of directory entry in memory to directory structure on disk.
Write	search *open file table* to find the location of file; write data to the position pointed by *Current Position Pointer.*
Read	search *open file table* to find the location of file; read data from the position pointed by *Current Position Pointer.*

Directory Structure

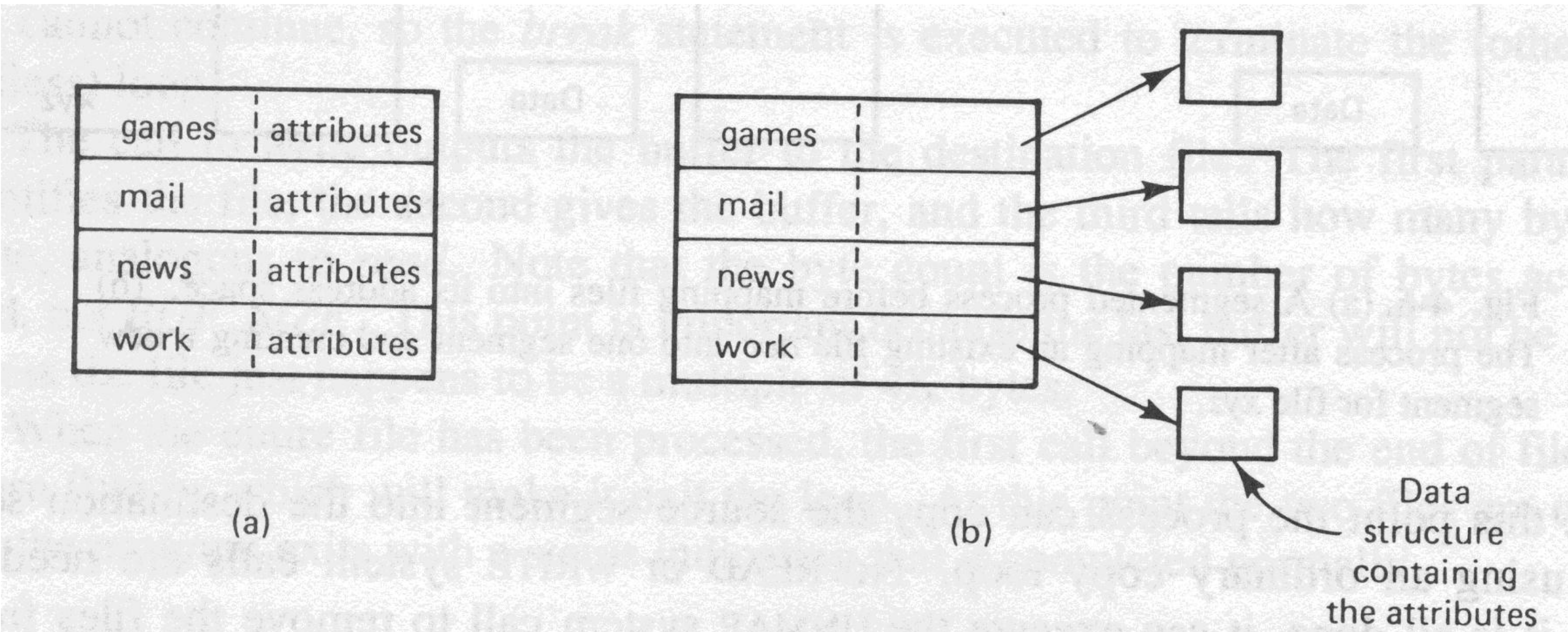
- A directory typically contains a number of entries, one per file. Both the directory and the files reside on disk.

- A directory can be structured in two ways:

  (a) each entry contains a file name and other attributes; or

  (b) each entry contains a file name and a pointer to another data structure where file attributes can be found.

Directory Structure (Cont.)



(File Control Block)

Directory Structure (Cont.)

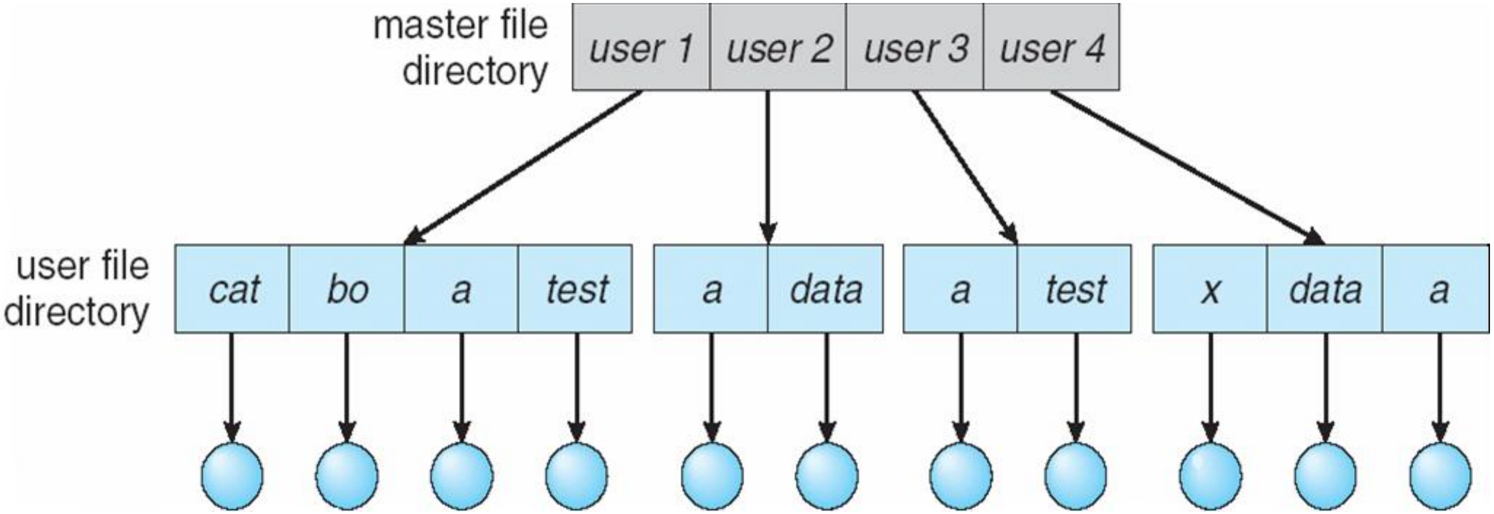
- A directory can be implemented using:
  - *Linear List*: Directory entries are arranged as linear list
    - simple to implement
    - time-consuming to execute: require linear search to find a particular entry
  - *Hash Table*: Directory entries are arranged as linear list with a hash table to facilitate file look-up.
    - decreases directory search time
    - collisions: situations where two file names hash to the same location (solution: each hash table entry contains a list of directory entries

Directory Organisation

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

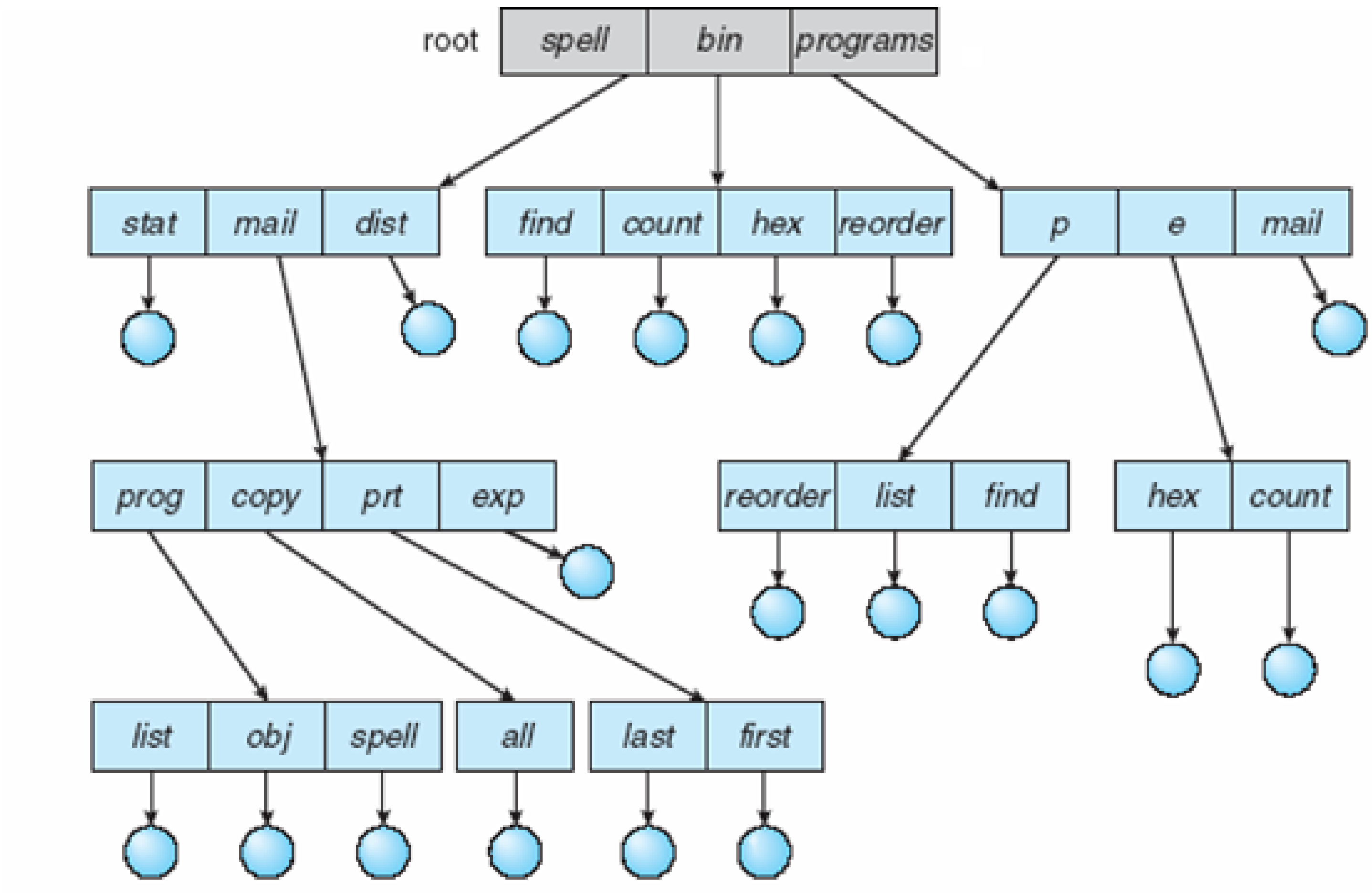
Two-Level Directory

- Keep a separate directory for each user



  - *Efficient Searching*: File can be located according to user name
  - *Naming*: Can have the same file name for different user
  - *Grouping*: No grouping capability

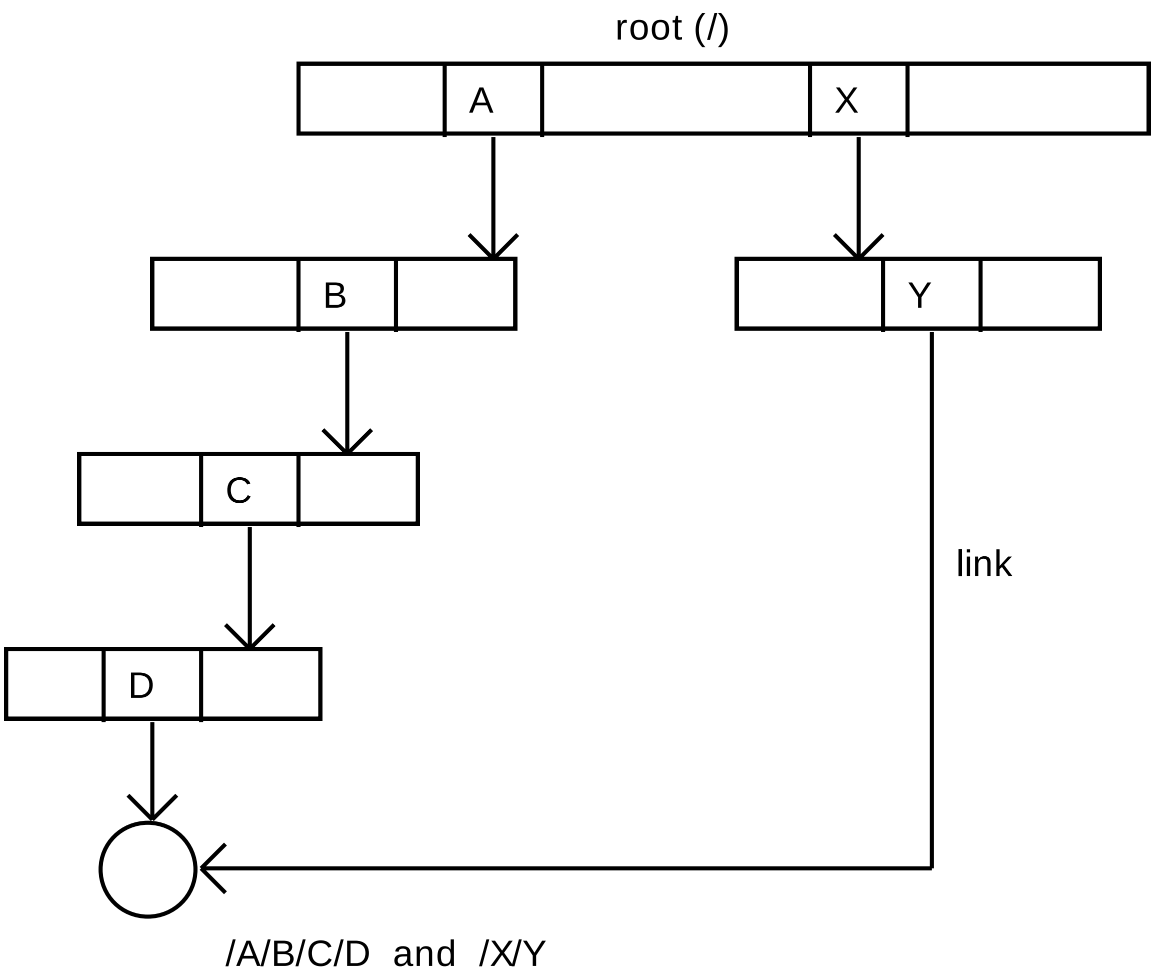
Tree-Structured Directories



Tree-Structured Directories (Cont.)

- Path Name
  - *Absolute Path Name*: begins at the root and follows a path down to the specific file, e.g., /spell/mail/prt/first
  - *Relative Path Name*: Defines a path from the current directory, e.g.,
    - current directory is: /spell/mail
    - relative path name for the above file is: prt/first
- Characteristics
  - *Efficient Searching*: File can be easily located according to the path name.
  - *Naming*: Files can have the same name under different directories.
  - *Grouping*: Files can be grouped logically according to their properties.

Acyclic-Graph Directory



Acyclic-Graph Directory (Cont.)

- An acyclic graph (that is, a graph with no cycles) is a natural generalisation of the tree-structured directory scheme
- *Naming*: A file can have two different names (aliasing)
- *Support for File Sharing*: This is necessary when users want to collaborate

Acyclic-Graph Directory (Cont.)

- Support for File Sharing: Two methods

  1. *Symbolic Link*
     - ❄ create a directory entry *link*, which contains absolute or relative path name of a file
     - ❄ resolve the link by using the path name to locate the real file
     - ❄ slower access than with hard link
  2. *Hard Link*
     - ❄ duplicate all information about a file in multiple directories

Acyclic-Graph Directory (Cont.)

- Problems with File Sharing:
  - In traversing the file system, the shared files may be visited more than once

    Solution: Ignore the link entry when traversing
  - Deleting a shared file may leave dangling pointers to the now-nonexistent file

    Solution:

    ❄ search for dangling links and remove them; or

    ❄ leave the dangling links and delete them only when they are used again; or

    ❄ preserve the file until all references to it are deleted.

Directory Operations

Commands	*Explanation*
Create	create a directory. In UNIX, two entries "." and ".." are automatically added when a directory is created. "." refers to the *current directory*; and ".." refers to its *parent*.
Delete	delete a directory. Only empty directory can be deleted (directory containing only "." and ".." is considered empty).
List	list all files (directories) and their contents of the directory entry in a directory
Search	search directory structure to find the entry for a particular file.
Traverse	access every directory and every file within a directory structure.

File Protection in UNIX

- Model of Access: *read*, *write*, *execute*
- Three Classes of Users:

				RWX
– owner access:	7	$\Rightarrow$		1 1 1
				R - X
– group access:	5	$\Rightarrow$		1 0 1
				- - X
– public access:	1	$\Rightarrow$		0 0 1

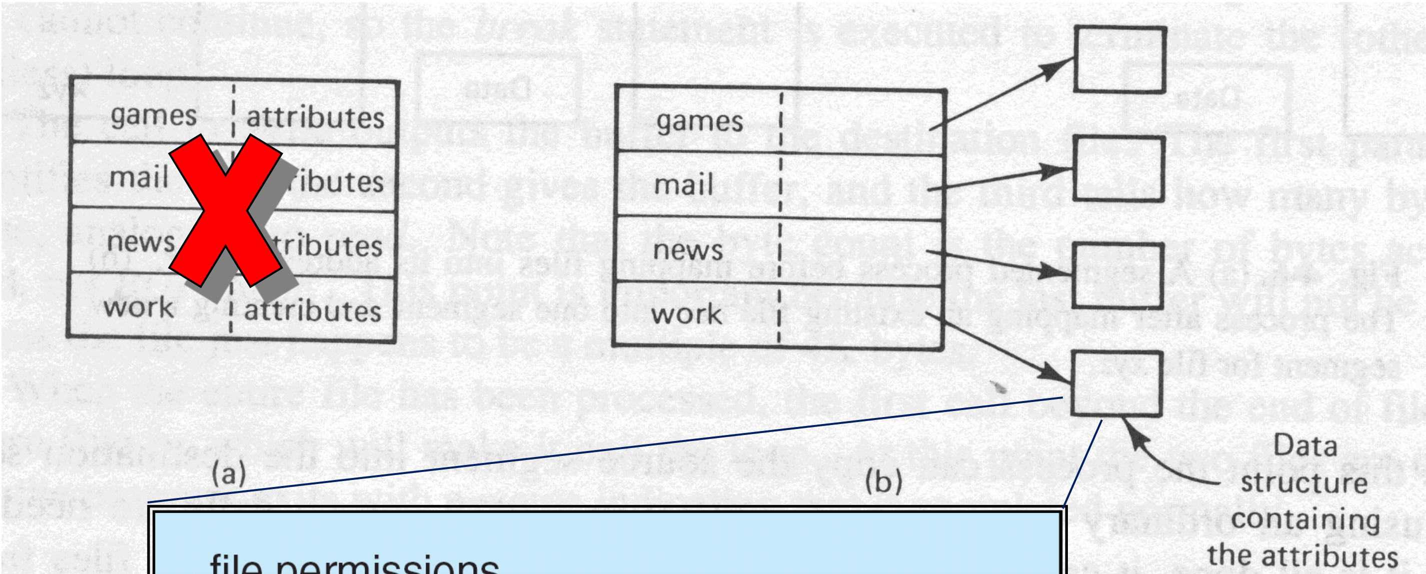
File type	User access	Group access	Public access	Links	Userid	Size	Date	Time	File name
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
-	r w x	r - x	- - x	1	smith	58	Mar 3	3:04	game1

File Protection in UNIX

- Meaning of Permissions for a Directory:
  - To access a directory, the execute permission is essential. No execute permission,
    - can't execute any command on the directory
    - have no access to any file contained in the file hierarchy rooted at that directory
  - No read permission -> can't list the directory
  - No write permission -> can't create or delete files in the directory

File type	User access	Group access	Public access	Links	Userid	Size	Date	Time	File name
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
d	r w x	r - x	- - -	0	smith	5	Feb 2	12:01	games

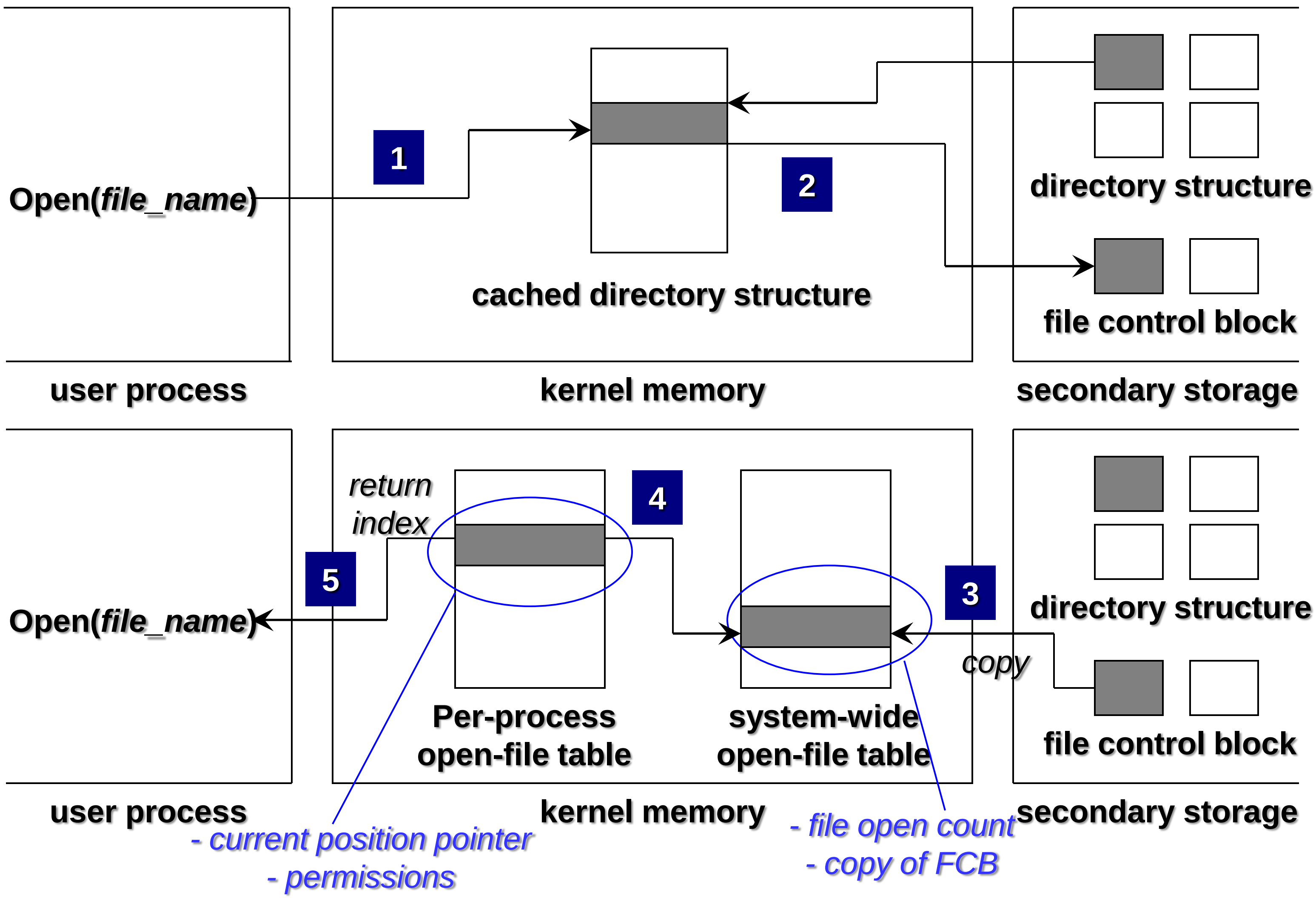
In-memory File System Data Structure



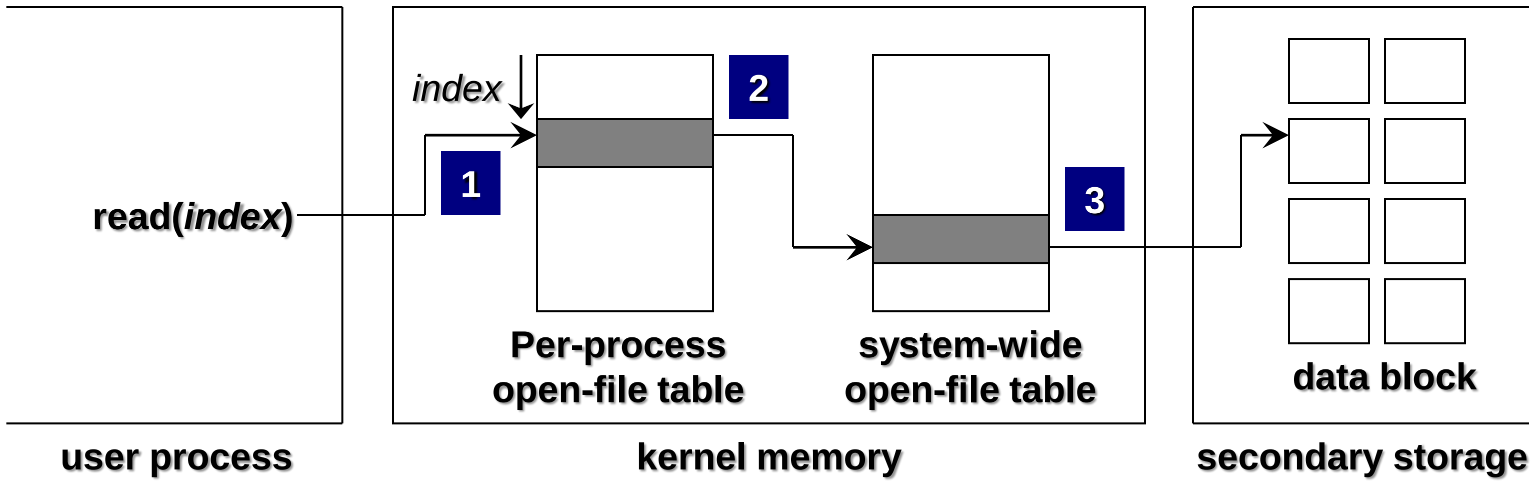
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

File Control Block

In-memory File System Data Structure



In-memory File System Data Structure



File System Structure

- A file system is generally composed of many different levels. For example:
  - *Logical File System*: manages directory structure, responsible for file creation, access, deletion, protection and security.
  - *File-Organisation Module*: allocates storage space for files, translates logical block addresses to physical block addresses, and manages free disk space.
  - *Basic File System*: manages buffers and caches issues generic commands to the appropriate device driver to read and write physical blocks on the disk.
  - *I/O Control*: consists of device drivers and interrupt handlers to transfer information between memory and the disk system.

Allocation Methods

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

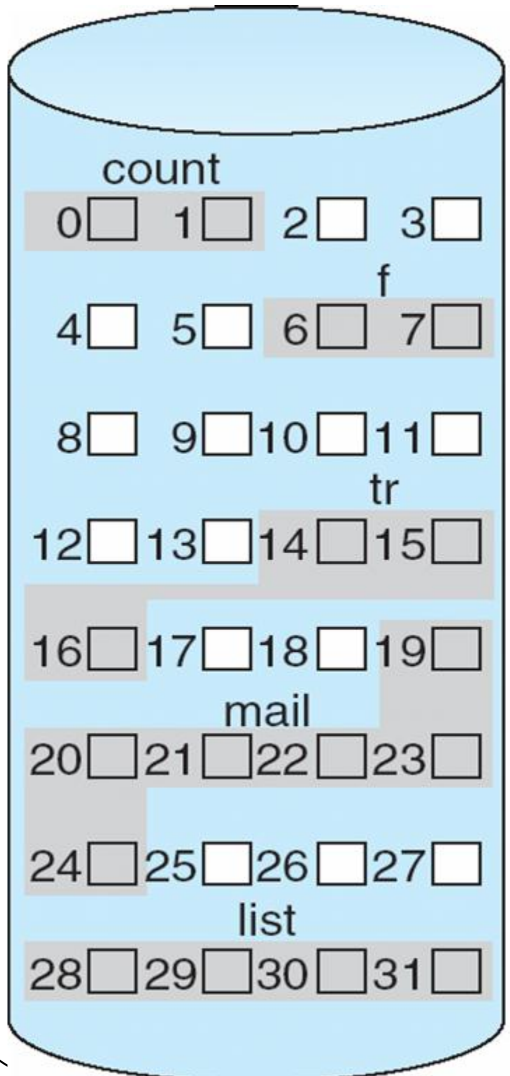
Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
  - Simple: only starting location (block #) and length (number of block) are required.
  - Support random access
- Problems:
  - Waste of space (similar to dynamic storage allocation of main memory)
  - Finding hole big enough using First-fit (faster) or Best-fit may result in external fragmentation
  - File space constricted by size of hole, so may later have to move to a bigger hole
  - If instead needed file space is overestimated $\Rightarrow$ internal fragmentation

Contiguous Allocation (Cont.)

- Logical to Physical Address Mapping:
  - Suppose block size is 512 bytes
  - Logical address $\Rightarrow$ $Q \times 512 + R$
  - Block to be accessed = Q + starting address
  - Displacement into block = R

Contiguous Allocation (Cont.)



directory

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

Logical Address $= 2333$

$= Q \times 512 + R$

$= 4 \times 512 + 285$

Physical Address = 285th byte of block 23

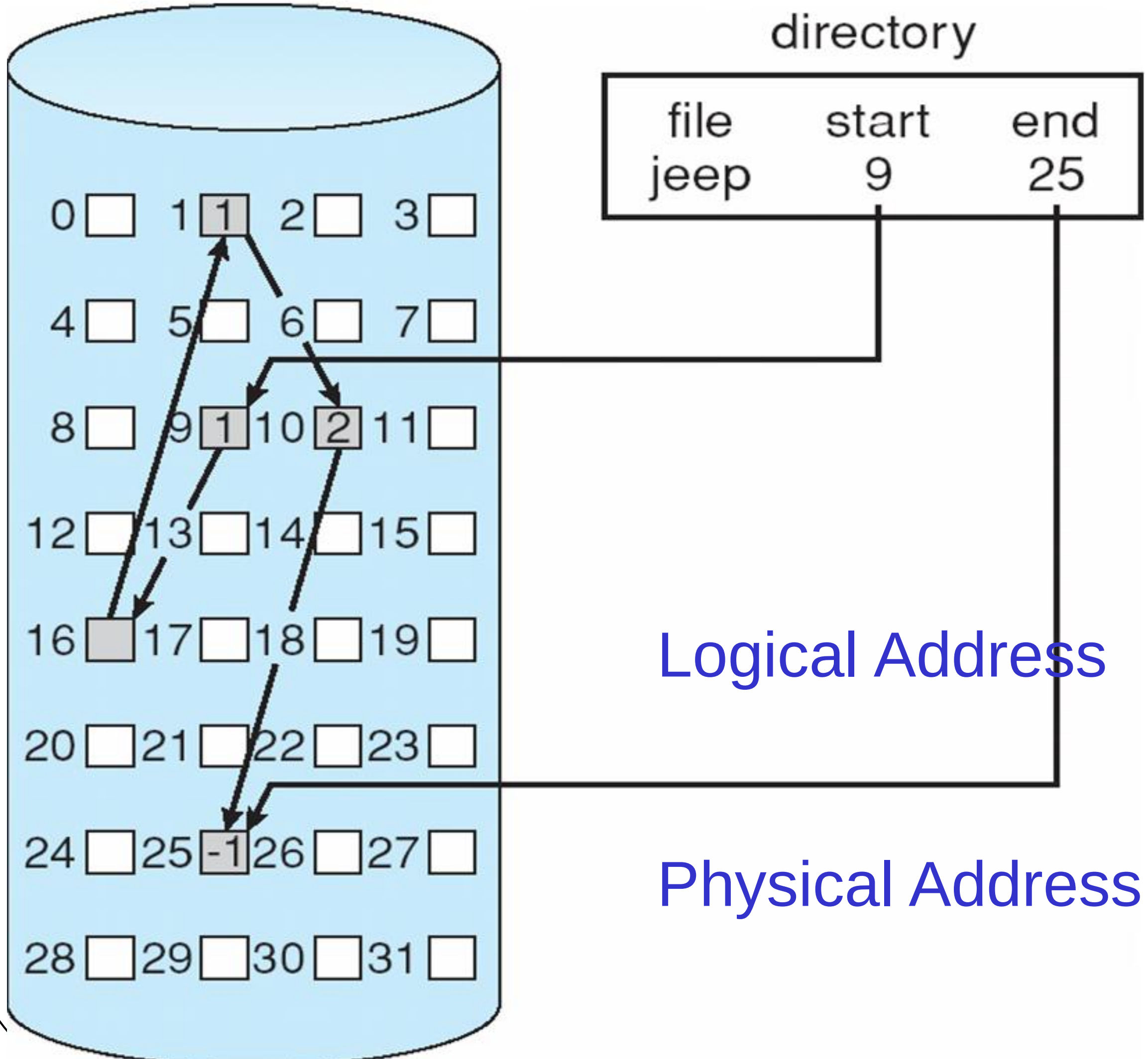
Linked Allocation

- Each file is a linked list of disk blocks; blocks may be scattered anywhere on the disk.
  - Simple: need only starting address
  - No waste of space
  - No constraint on file size: blocks can be allocated as needed
  - Problem: random access not supported

Linked Allocation (Cont.)

- Logical to Physical Address Mapping:
  - Suppose block size is 512 bytes, and first 4 bytes is reserved for the *pointer* to the next block in the list.
  - Logical Address $\Rightarrow Q \times 508 + R$
  - Block to be accessed is the (Q+1)th block in the linked chain of blocks representing the file
  - Displacement into block = $R + 4$

- Allocate as needed, link together; e.g., file starts at block 9



Logical Address $= 2333$

$= Q \times 508 + R$

$= 4 \times 508 + 301$

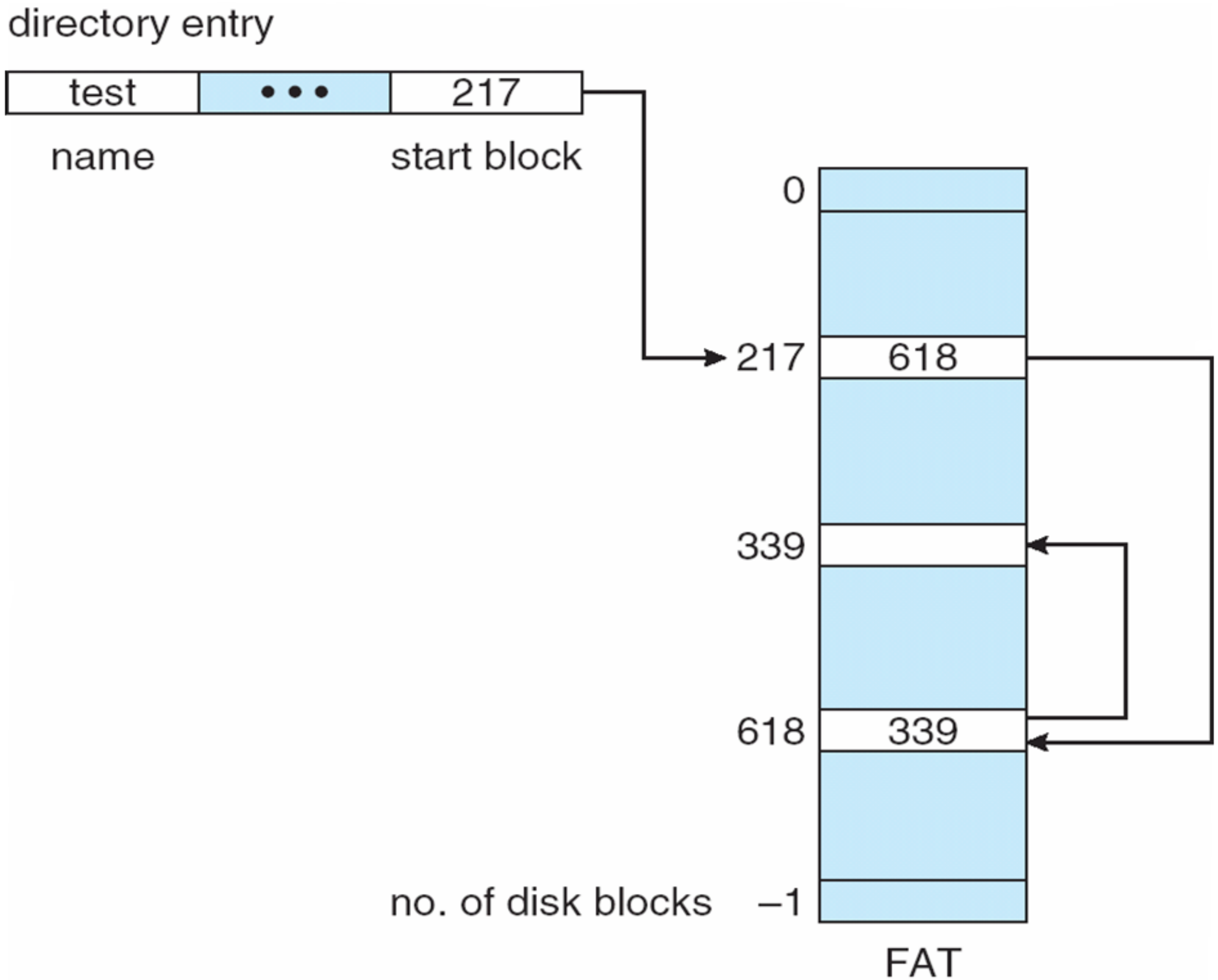
Physical Address Block = 25

Displacement = 301+4

Linked Allocation: File Allocation Table

- Disk space allocation used by MS-DOS and OS/2
- File Allocation Table (FAT) has one entry for each disk block, and is indexed by block number.
- The FAT entry contains either a "free", or a special end-of-file value, or the block number of the next block in the file.
- The directory entry contains the block number of the first block of the file.
- Random access can be optimised.

FAT (Cont.)



directory entry
test
• • •
217
name
start block
0
217
618
339
618
339
no. of disk blocks
–1
FAT

Indexed Allocation

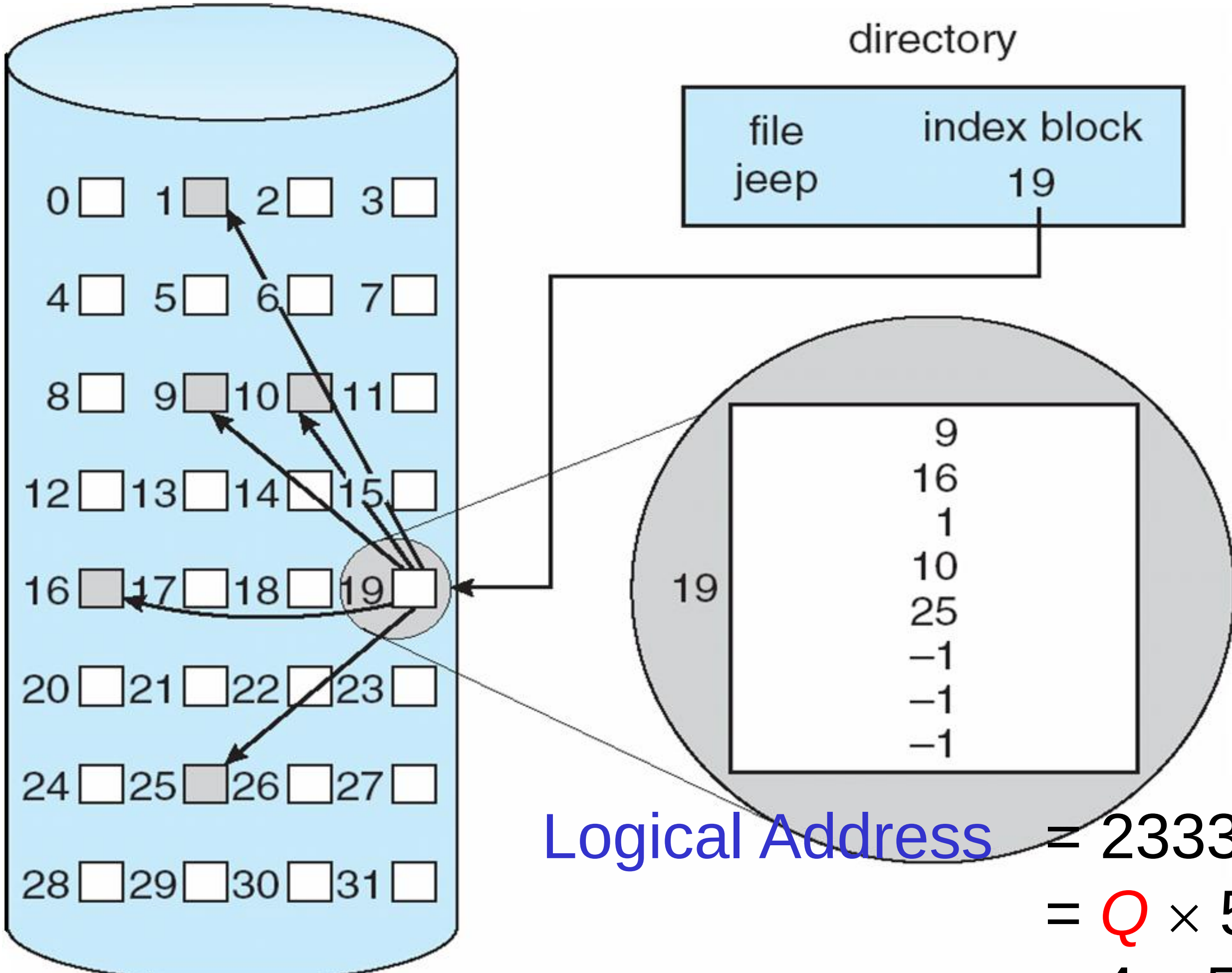
- Each file has an index block which contains all pointers to the allocated blocks. Directory entry contains the block number of the index block.
  - Support random access.
  - Dynamic storage allocation without external fragmentation (similar to the allocation of main memory using paging scheme).
  - *Problem*: Overhead of keeping index blocks and address mapping.

Indexed Allocation (Cont.)

- Logical to Physical Address Mapping
  - Suppose maximum size of a file is 128K bytes and block size is 512 bytes.
  - 2 blocks are needed for index table (4 bytes are used for each pointer)
  - Logical Address $\Rightarrow Q \times 512 + R$
  - Displacement into index table = Q
  - Displacement into block = R

Why are 2 blocks needed for index table in this case ?

Example of Indexed Allocation



Logical Address = 2333

= $Q \times 512 + R$

= $4 \times 512 + 285$

Physical Address = 285th byte of block 25

Indexed Allocation: UNIX inode

- For each file or directory, there is an *inode* (index block).
- The inode contains:
  - *file attributes*
  - 12 pointers point to *direct blocks* (data blocks)
  - 3 pointers point to *indirect blocks*. They are pointers to
    - ❄*single indirect block* (index block);
    - ❄*double indirect block*; and
    - ❄*triple indirect block*

inode (Cont.)

- Assume 4-byte file pointer and 4K bytes block. Maximum file size:
  - direct access: $12 \times 4K = 48K$ bytes
  - single indirect access: $48K + 2^{22}$ bytes
  - double indirect access: $48K + 2^{22} + 2^{32}$ bytes > 4 gigabytes

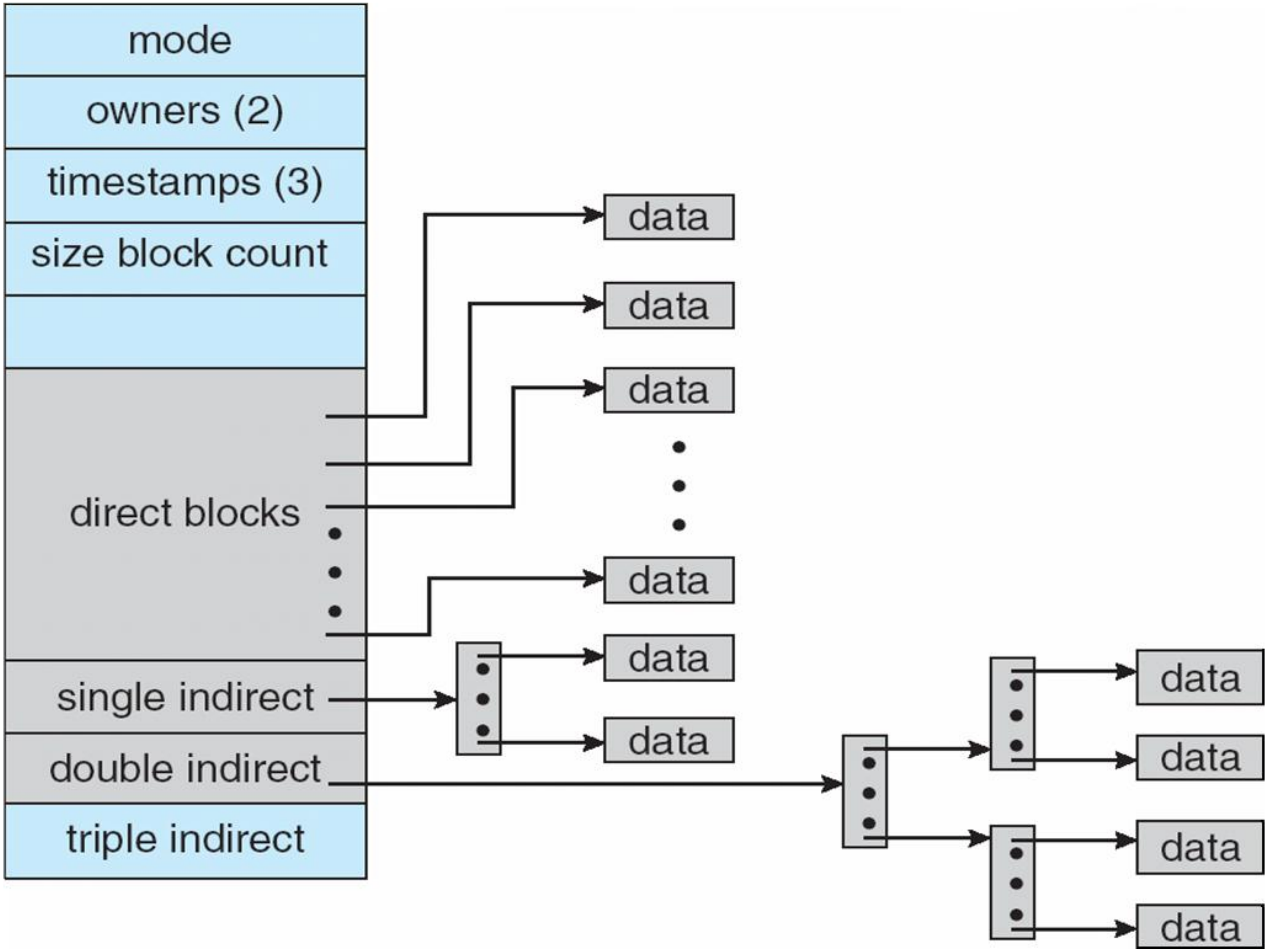
$$\frac{2^{12}}{2^{2}} \times 4K = 2^{22}$$

block size

# of pointers per block

$$2^{10} \times 2^{10} \times 4K = 2^{32}$$

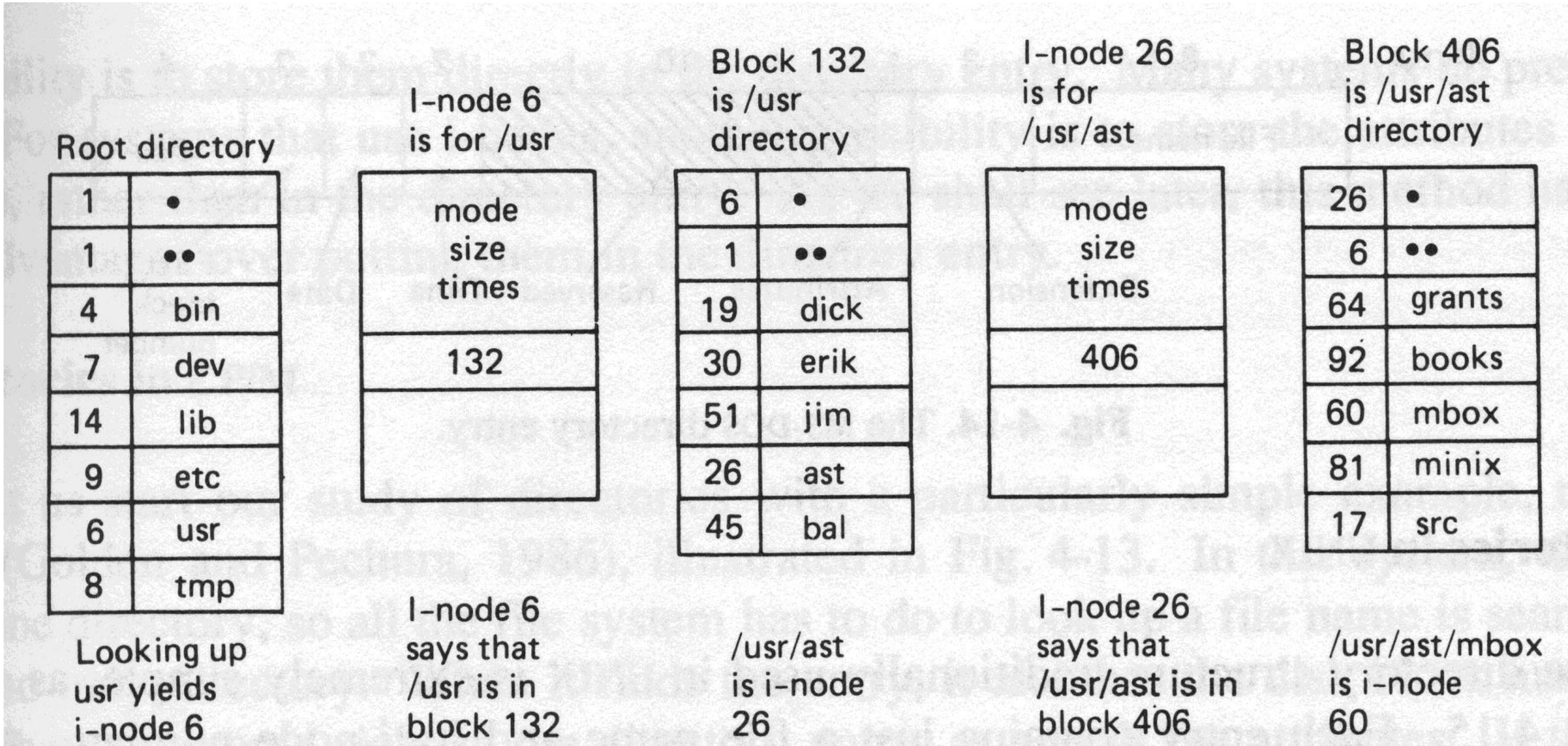
inode (Cont.)


mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

inode (Cont.)

- Each directory entry in UNIX contains an inode number and a file name.
- Each inode has a fixed location in disk.
- File look-up is straightforward. For example, looking up /usr/ast/mbox:

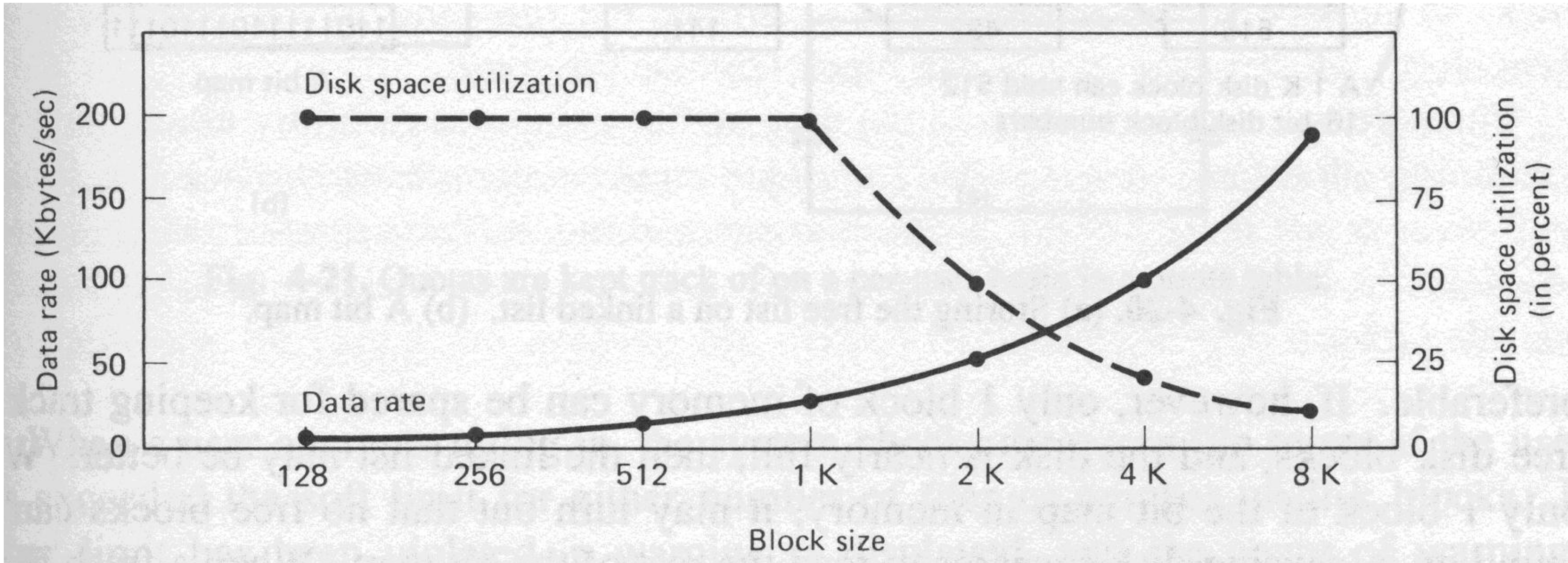
inode (Cont.)


Root directory
1 •
1 ••
4 bin
7 dev
14 lib
9 etc
6 usr
8 tmp
Looking up usr yields i-node 6
I-node 6 is for /usr
mode size times
132
I-node 6 says that /usr is in block 132
Block 132 is /usr directory
6 •
1 ••
19 dick
30 erik
51 jim
26 ast
45 bal
/usr/ast is i-node 26
I-node 26 is for /usr/ast
mode size times
406
I-node 26 says that /usr/ast is in block 406
Block 406 is /usr/ast directory
26 •
6 ••
64 grants
92 books
60 mbox
81 minix
17 src
/usr/ast/mbox is i-node 60

Disk-Space Management

- Block size affects both data rate and disk space utilisation
  - Big block size: file fits into few blocks $\Rightarrow$ fast to find & transfer blocks, but wastes space if file does not occupy the entire last block
  - Small block size: file may consist of many blocks $\Rightarrow$ slow data rate
  - Trade-off between time and space utilisation has to be compromised

Disk-Space Management (Cont.)



Disk-Space Management (Cont.)

- Keeping Track of Free Blocks: Similar to the issue of *Keeping Track of Memory Usage* in memory management under variable partition multiprogramming. Methods are:
  - Bit Map or Bit Vector
  - Linked List

Bit Map or Bit Vector

- Each block is represented by 1 bit:
  0 $\Rightarrow$ block is free; 1 $\Rightarrow$ block is allocated

- Bit map requires extra space, e.g.,
  - block size = 2^9 bytes
  - disk size = 2^{34} bytes (16 gigabyte)
  - bit map size = $(2^{34}/2^9)/8 = 2^{22}$ bytes

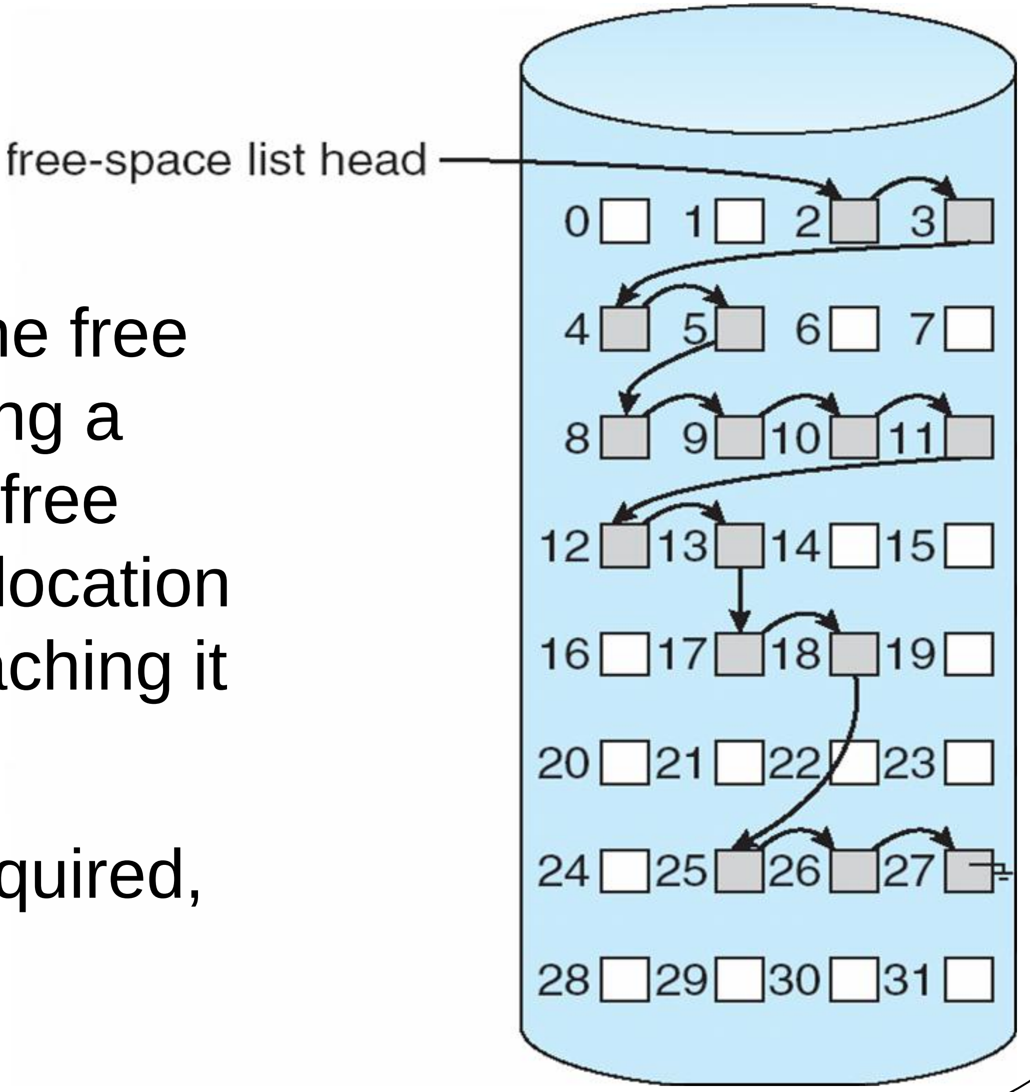
Bit Map (Cont.)

- Bit map is usually kept in a fixed place on disk. May be brought into memory for efficiency - needs to write back to disk occasionally for consistency and security.

- Easy to locate free blocks, but inefficiency unless the entire map is kept in memory.

Linked List

- 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.
- No extra space required, but not efficient.



Summary

- Logical File System
  - Files
    - ❄ e.g., file structures,
  - Directories
    - ❄ e.g., support for file sharing
  - In-memory Data Structures
- File Organization Module
  - Allocation Methods: contiguous, linked and indexed allocation
  - Disk Space Management