CSI 4500 Operating Systems

File System

Today's Objectives

- File System Motivation
  - To explain the function of file systems
  - To describe the interfaces to file systems
  - To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
  - To explore file-system protection
- To describe the details of implementing
  - Local file systems
  - Directory structures
- To discuss block allocation and free-block algorithms and trade-offs

Storing Information

- So far…
  - We have discussed processor, memory, I/O
- How do we make stored information usable?
- Applications can store information in the process address space
- Why permanent storage?
  - Size is limited to size of virtual address space
    - May not be sufficient for search engines, banking, etc.
  - The data is lost when the application terminates
    - Even when computer crashes!
  - Multiple process might want to access the same data
    - Imagine a telephone directory part of one process

File Systems

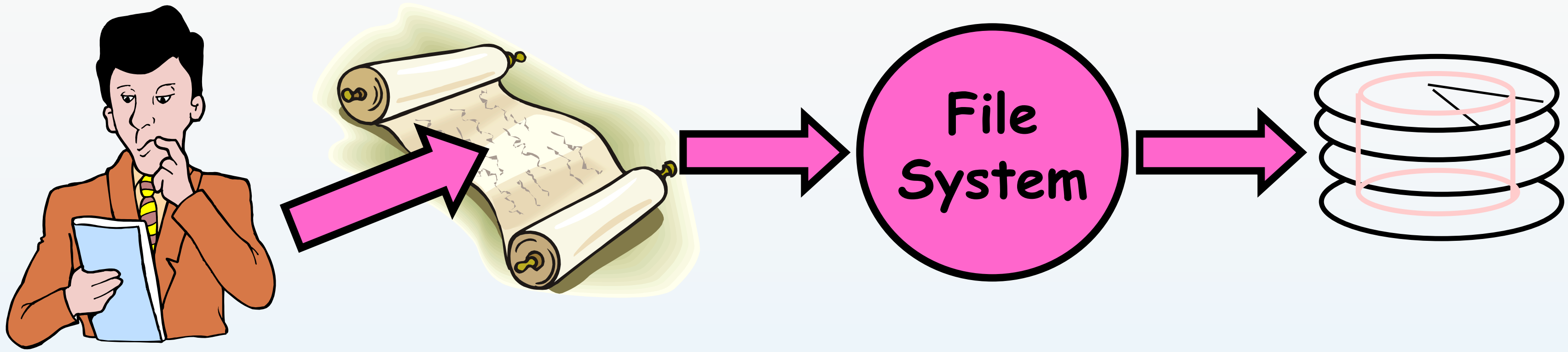
- 3 criteria for long-term information storage:
  - Should be able to store very large amount of information
  - Information must survive the processes using it
  - Should provide concurrent access to multiple processes
- Solution:
  - Store information on disks in units called **files**
  - Files are persistent, and only owner can explicitly delete it
  - Files are managed by the OS

- File Systems: How the OS manages files!

File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - **Disk Management**: collecting disk blocks into files
  - **Naming**: Interface to find files by name, not by blocks
  - **Protection**: Layers to keep data secure
  - **Reliability/Durability**: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view: Durable Data Structures
  - System's view (system call interface): Collection of Bytes (UNIX)
    - Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - Block size $\geq$ sector size; in UNIX, block size is 4KB

Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - 1) Fetch block 2) Modify portion 3) Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks (i.e. systems view inside OS)

File System Patterns

- How do users access files?
  - Sequential Access
    - bytes read in order ("give me the next X bytes, then give me next, etc")
  - Random Access
    - read/write element out of middle of array ("give me bytes i - j")
- What are file sizes?
  - Most files are small (for example, .login, .c, .o, .class files, etc)
  - Few files are large (for example, core files, etc.)
  - Large files use up most of the disk space and bandwidth to/from disk
    - May seem contradictory, but a few enormous files are equivalent to an immense # of small files

File Concept

- A **File** is a named collection of related information that is recorded on secondary storage.
  - Contiguous logical address space
- File represents:
  - Data
    - Numeric, character, binary
    - Payroll records, graphic images, sound records, etc.
  - Program
    - Source programs, object programs, executable programs

File Attributes

- **File Specific information maintained by OS**
  - File size, modification date, creation time, etc.
  - Varies a lot across different OSes
  - Information about files are kept in the directory structure, which is maintained on the disk
- **Some Examples:**
  - **Name** – only information kept in human-readable form
  - **Identifier** – unique tag (number) identifies file within file system
  - **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

File Operations

- File is an Abstract Data Type
- Some basic file operations:
  - Create a file
    - Find space in FS, add an new entry in the directory
  - Write a file
    - Search file name in the directory to find its location
    - Information to be written
    - Write pointer
  - Read a file
    - Search file name in the directory to find its location
    - Location to put the read results
    - Read pointer
  - Reposition within file
    - Current-file-position pointer
  - Delete
    - Delete entry from the directory
  - Truncate
    - Keep file attributes except the file length

Open and Close Files

- **Open(F_i)** – search the directory structure on disk for entry F_i, and move the content of entry to memory
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information
- **Close (F_i)** – move the content of entry F_i in memory to directory structure on disk
  - Remove from the open-file table

File Naming

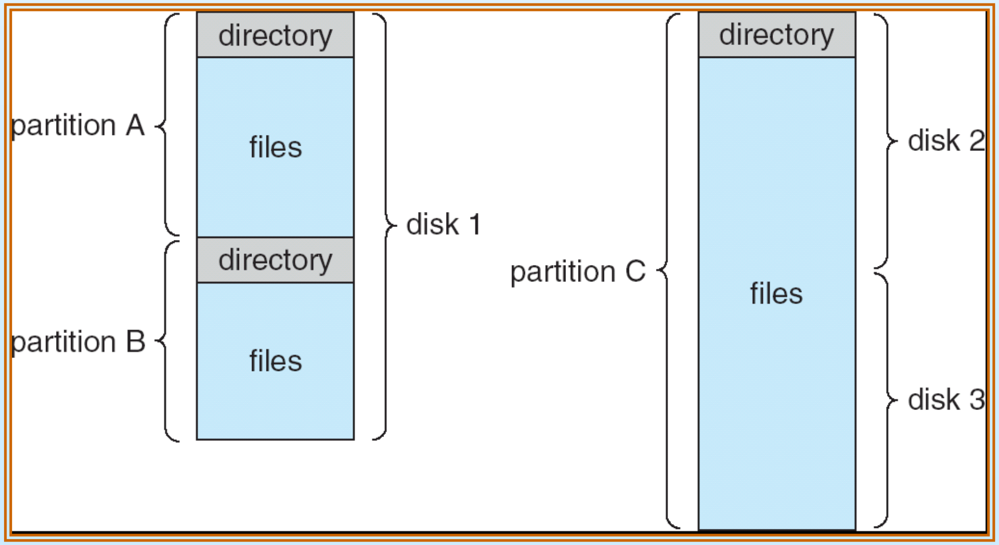
- Motivation: Files abstract information stored on disk
  - You do not need to remember block, sector, …
  - We have human readable names
- How does it work?
  - Process creates a file, and gives it a name
    - Other processes can access the file by that name
  - Naming conventions are OS dependent
    - Usually names as long as 255 characters is allowed
    - Digits and special characters are sometimes allowed
    - MS-DOS and Windows are not case sensitive, UNIX family is

File Structure

- Logical record vs physical block
  - Packing a number of logical records into physical block
- Some Example Structures:
  - None - sequence of words, bytes
    - Unix defines all files to be simply streams of bytes.
  - Simple record structure
    - Lines, Fixed length, Variable length
  - Complex Structures
    - Formatted document, Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters

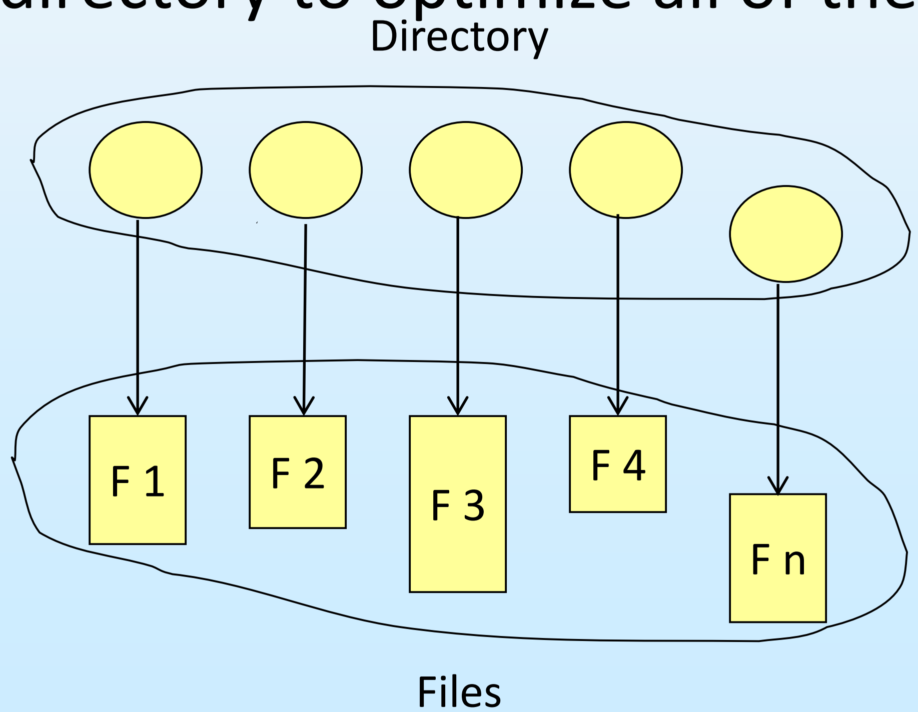
A Typical File-system Organization

- Could use entire disk space for a FS, but
  - A system could have multiple FSes
  - Want to use some disk space for swap space
- Storage Structure
  - A disk is divided into **partitions/slices**.
  - **Directory** records information
    - Name, location, size, and type



Directory Structure

- A collection of nodes containing information about all files.
  - Both the directory structure and the files reside on disk.
  - Symbol table
- How to structure the directory to optimize all of the following operations:
  - Search a for file
  - Create a file
  - Delete a file
  - List directory
  - Rename a file
  - Traversing the FS

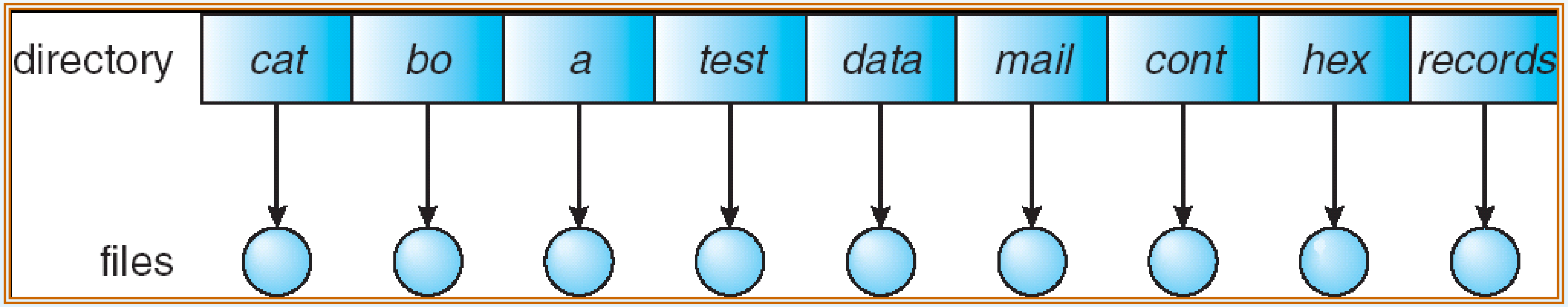


Organize the Directory (Logically) to Obtain

- 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 Java programs, all games, …

Single-Level Directory

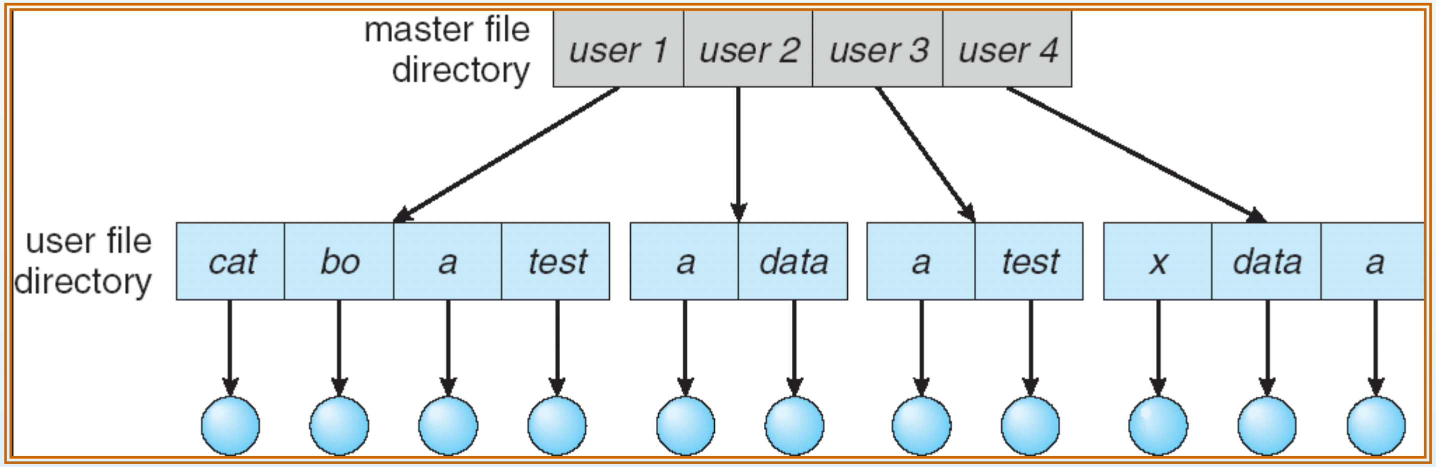
- A single directory for all users
  - Called root directory



- Pros: simplicity, ability to quickly locate files
- Cons: inconvenient naming
  - Naming problem
    - File name must be unique
    - remembering all
  - Grouping problem

Two-Level Directory

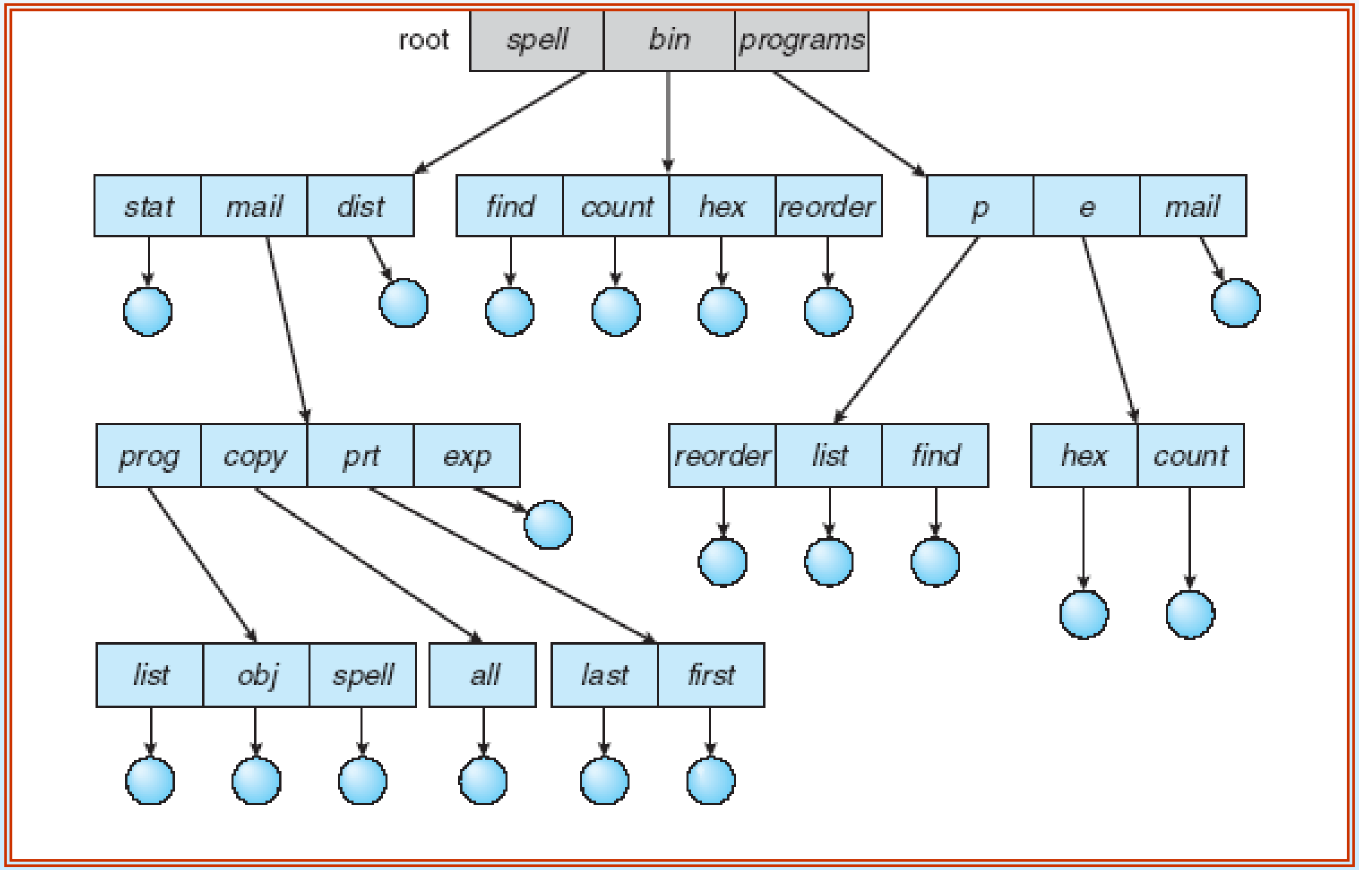
- Separate directory for each user



- Solve name collision, but what if user has lots of files
- Files need to be addressed by path names
  - Allow user's access to other user's files
  - Need for a search path (for example, locating system files)
    - Efficient searching ?

Tree-Structured Directories

- Directory is now a tree of arbitrary height
  - Directory contains files and subdirectories
  - A bit in directory entry differentiates files from subdirectories



Tree-Structured Directories (Cont)

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

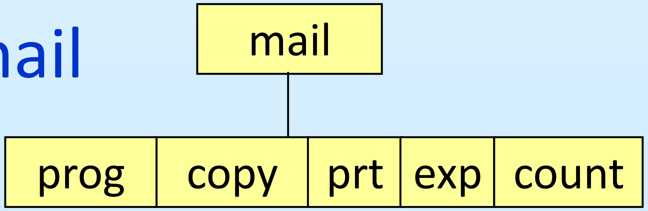
  rm <file-name>

- Creating a new subdirectory is done in current directory

  mkdir <dir-name>

Example: if in current directory /mail

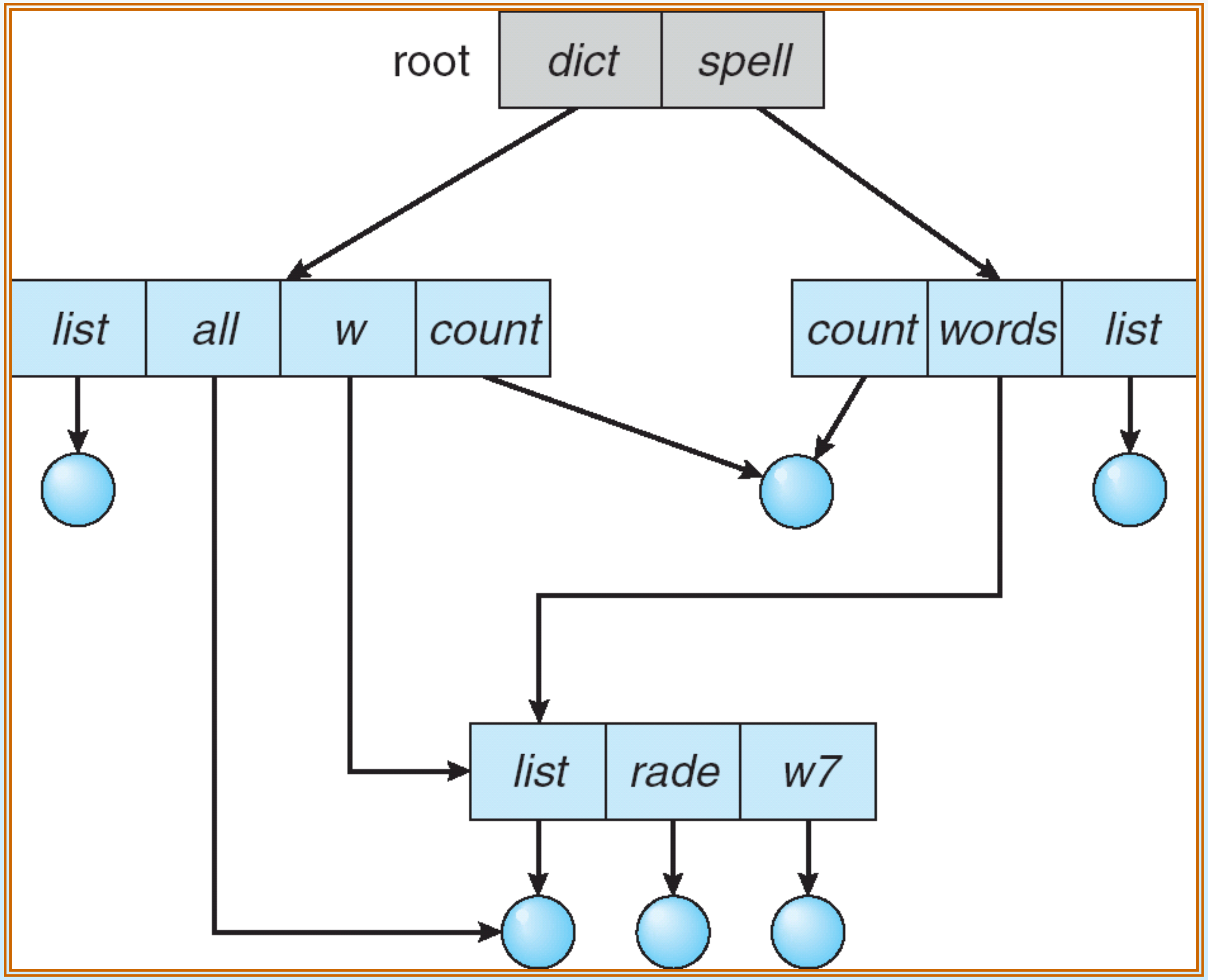
mkdir count



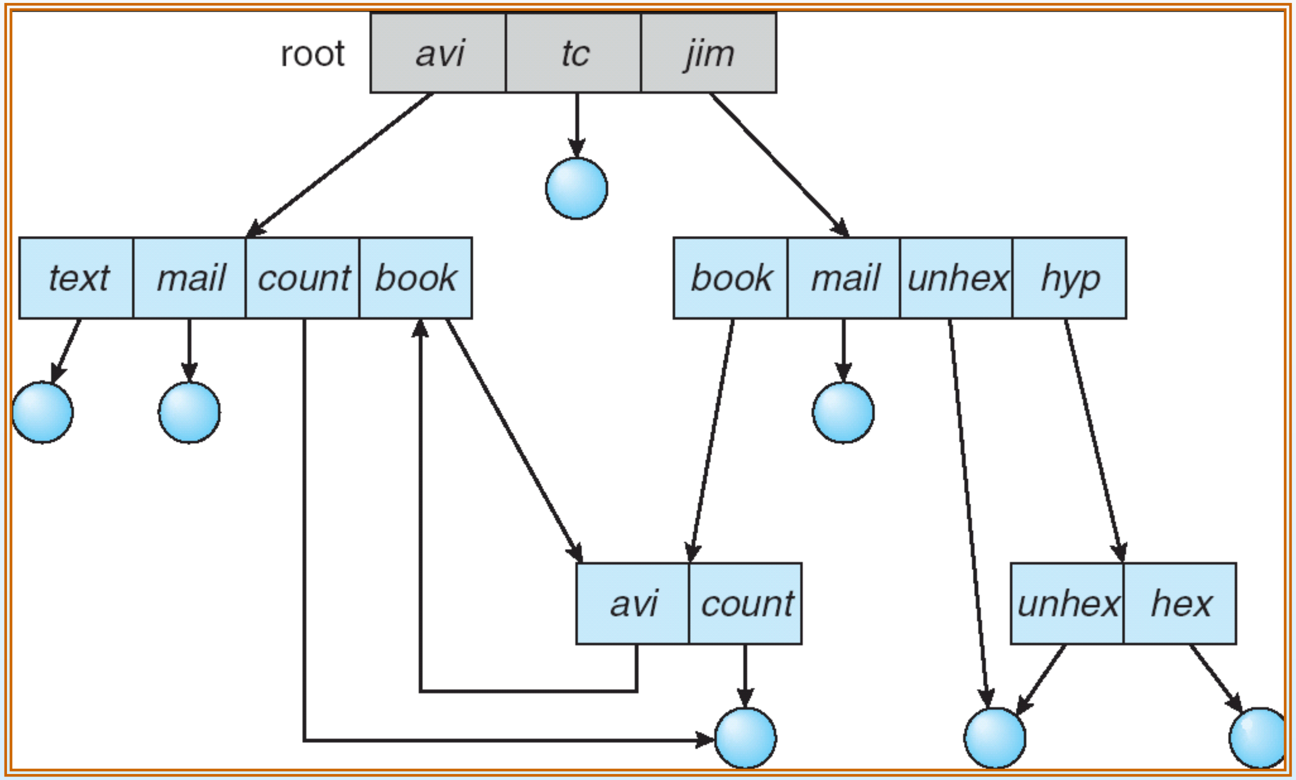
Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files



General Graph Directory



- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

File Sharing

- Sharing of files on multi-user systems is desirable
  - **User IDs** identify users, allowing permissions and protections to be per-user
  - **Group IDs** allow users to be in groups, permitting group access rights
- 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

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) **owner access**	7	$\Rightarrow$	1 1 1
			RWX
b) **group access**	6	$\Rightarrow$	1 1 0
			RWX
c) **public access**	1	$\Rightarrow$	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.

owner group public

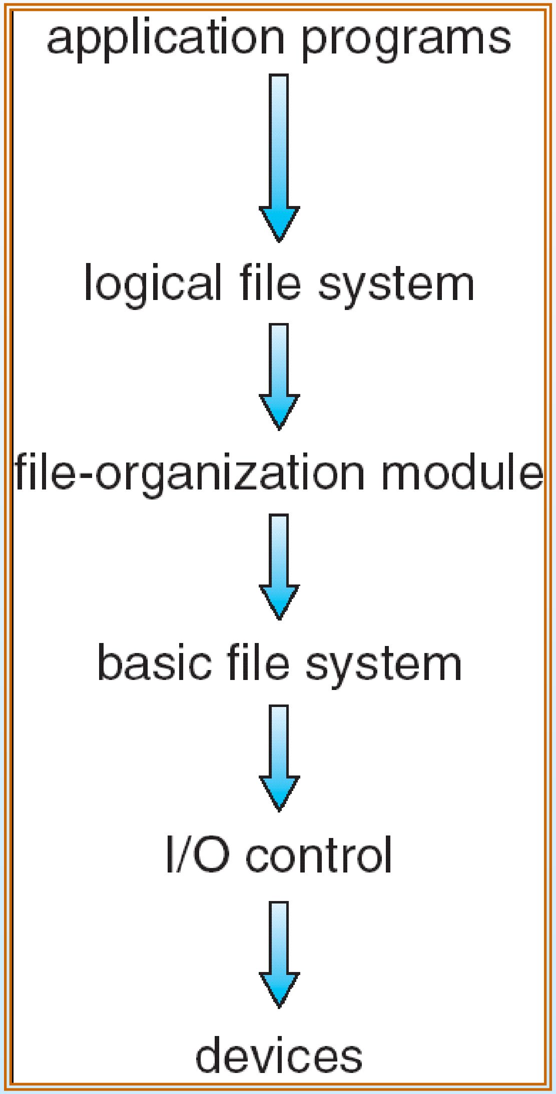
chmod 761 game

Attach a group to a file

chgrp G game

File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks)
- File system organized into layers



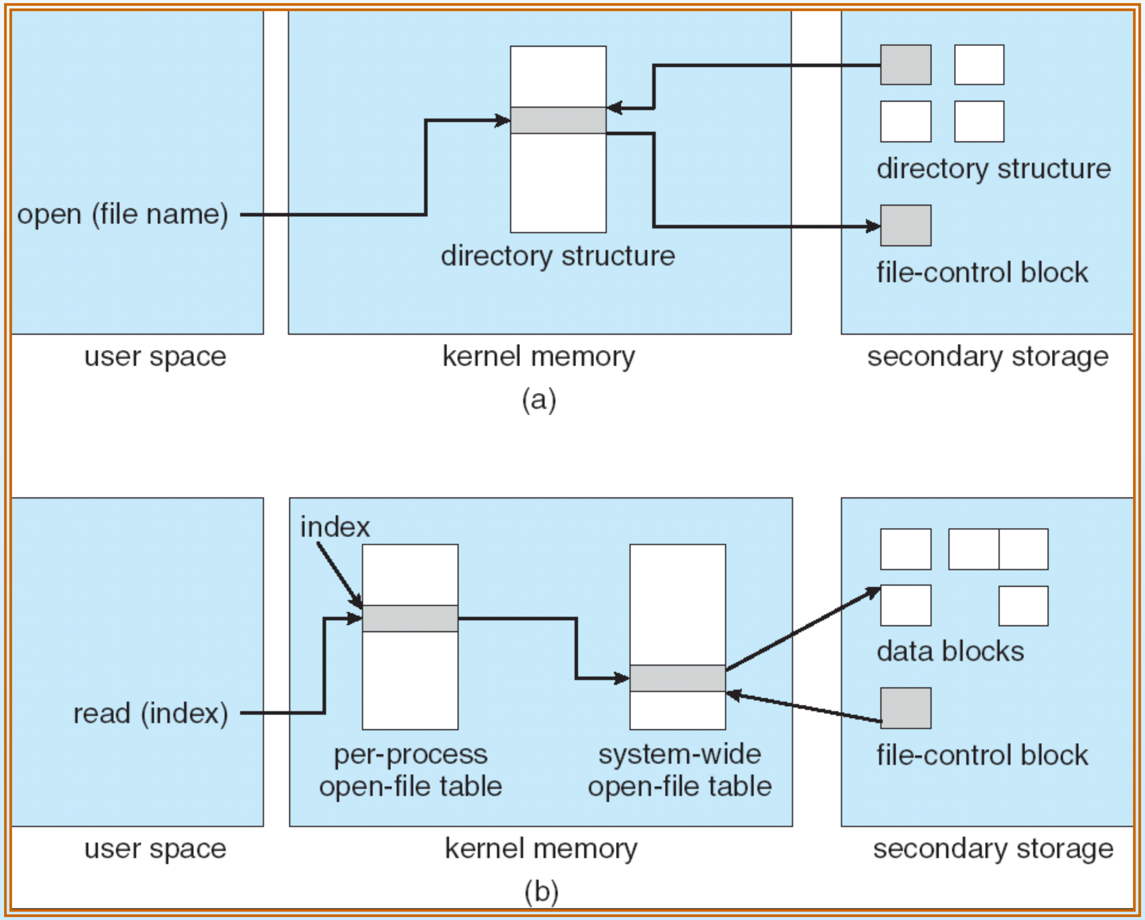
File-System Structure - FCB

- **File control block** – storage structure consisting of information about a file

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

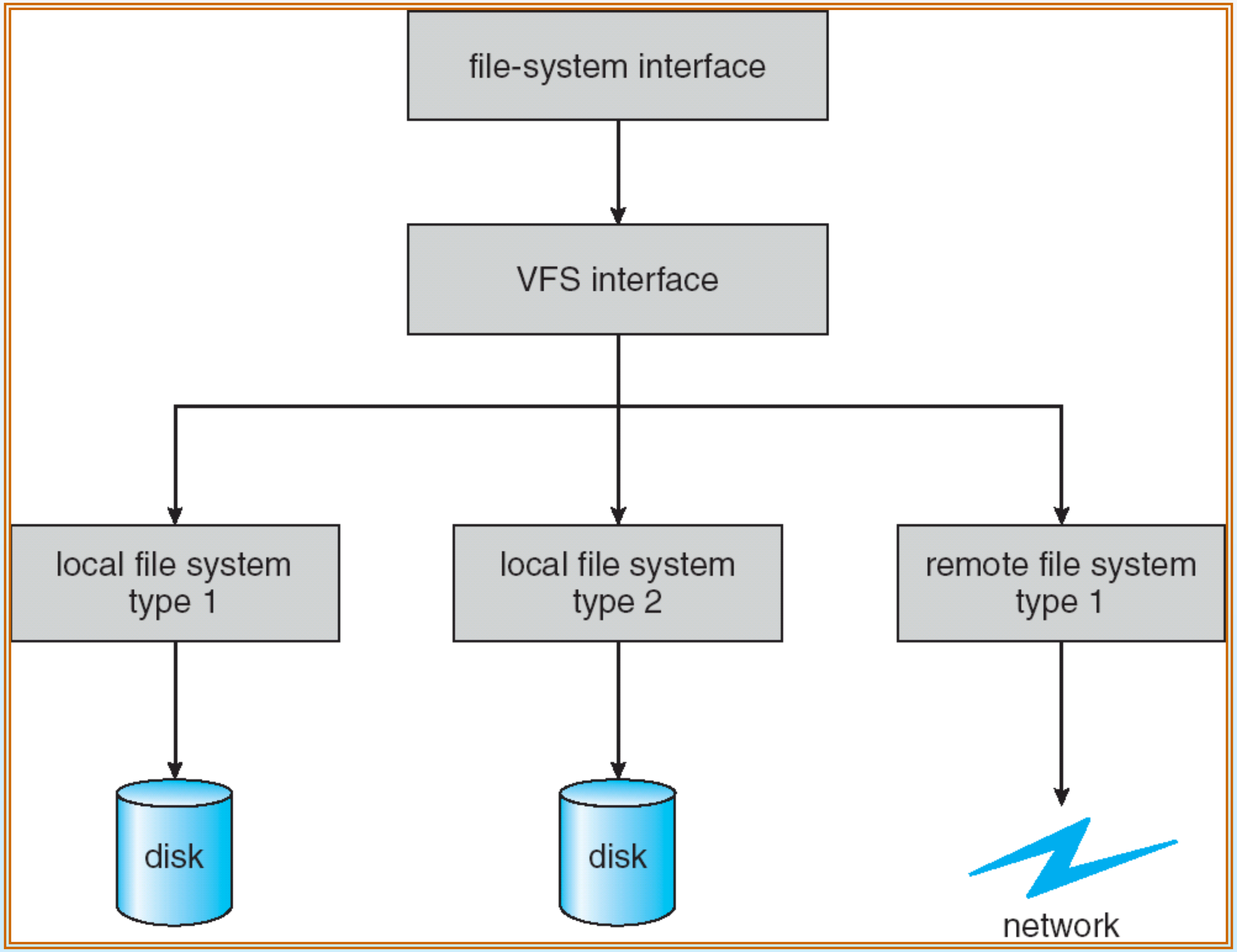
- The following figures illustrates the necessary file system structures provided by the operating systems.



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
    - Create vs Delete
- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size
- Other data structures
  - Sorted list, B-Tree, etc.

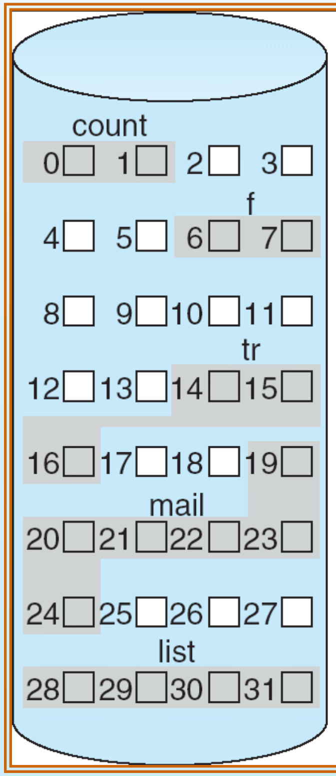
Oakland UNIVERSITY

Disk Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
  - **Contiguous allocation**
  - **Linked allocation**
  - **Indexed allocation**

Contiguous Allocation

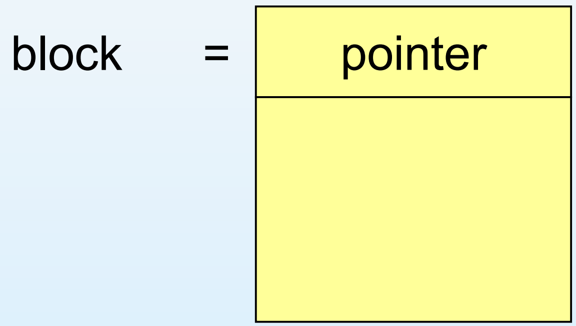
- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
  - Sequential access (b+1)
  - Direct access (b+i)
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow easily



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

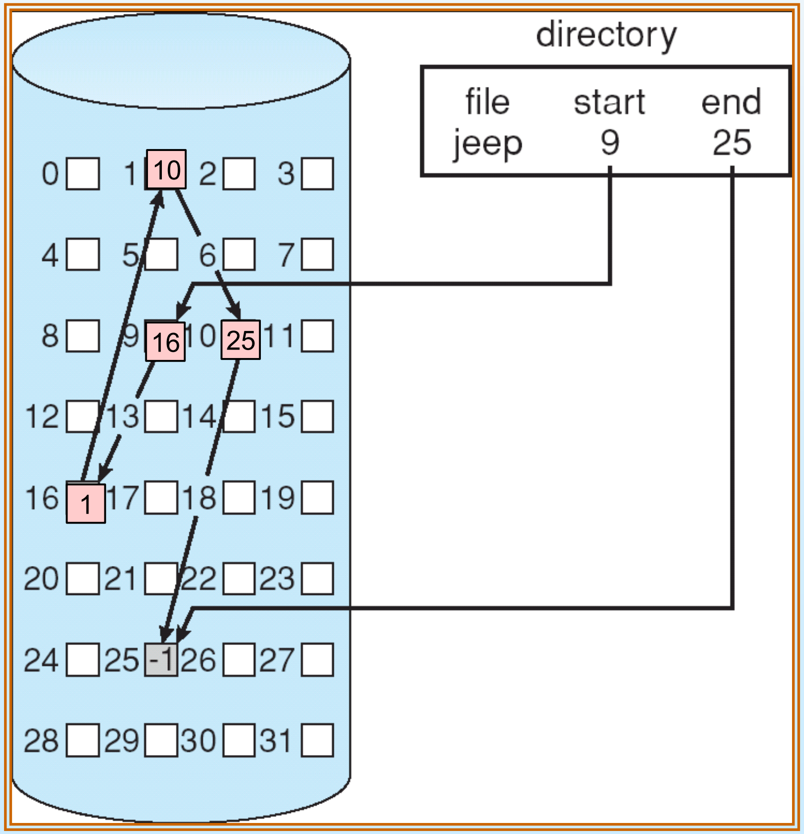
Linked Allocation

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



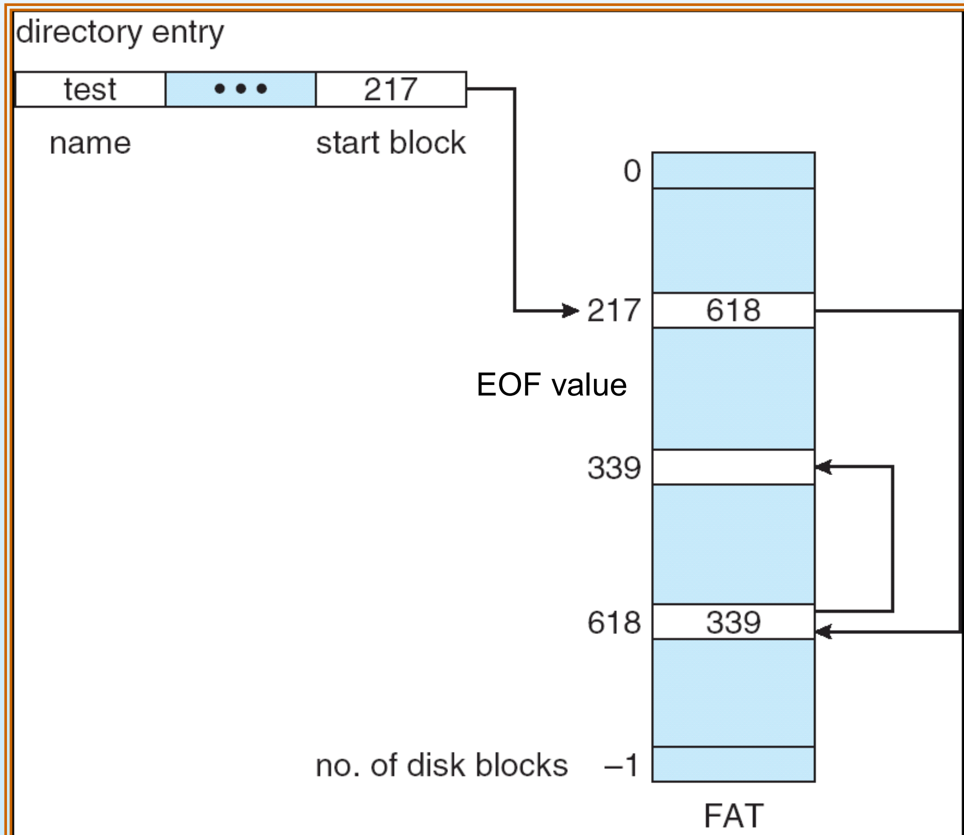
Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system
  - No waste of space
  - Really?
- Cluster based allocation
- No random access
  - Why?
- Reliability Issue



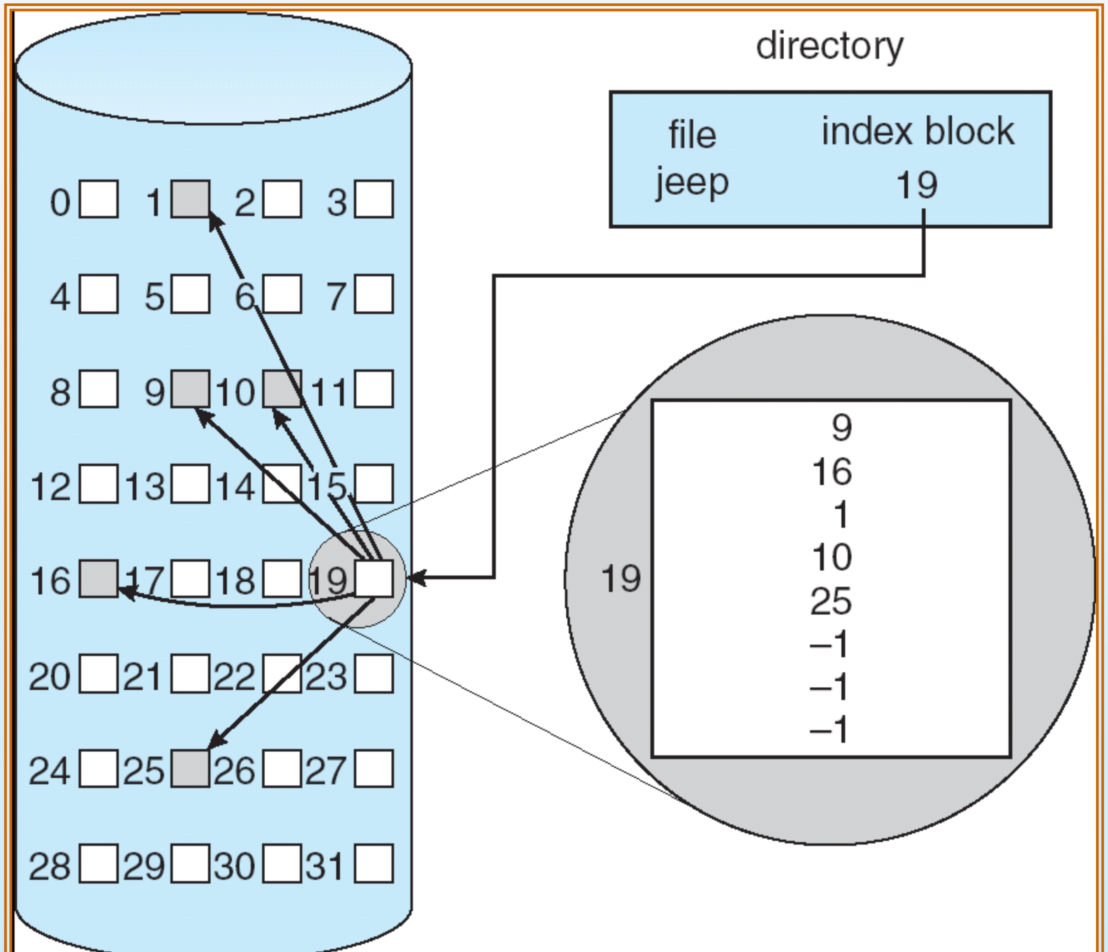
File-Allocation Table

- A variation of the Linked Allocation.
  - A section of disk at the beginning of each volume is set aside.
  - Simple and efficient
  - Entry represents disk block.
  - EOF vs 0 values
    - 0 means unused blocks
- Improved random-access time.
- Disk head seeks increases
  - Why?



Indexed Allocation

- Brings all pointers together into the *index block.*

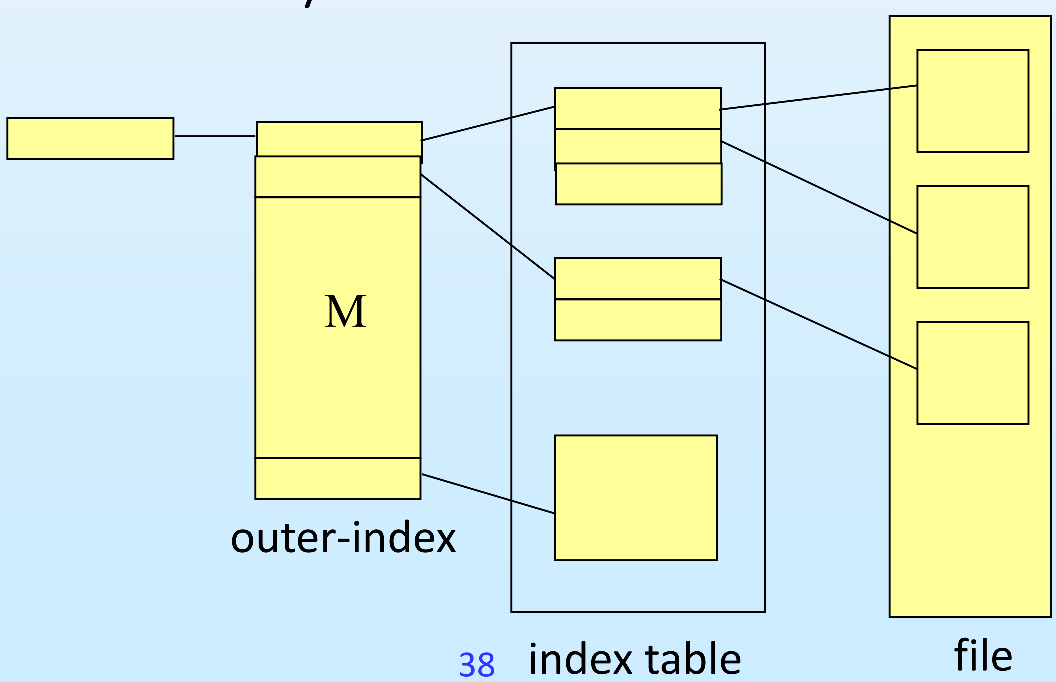


Indexed Allocation (Cont.)

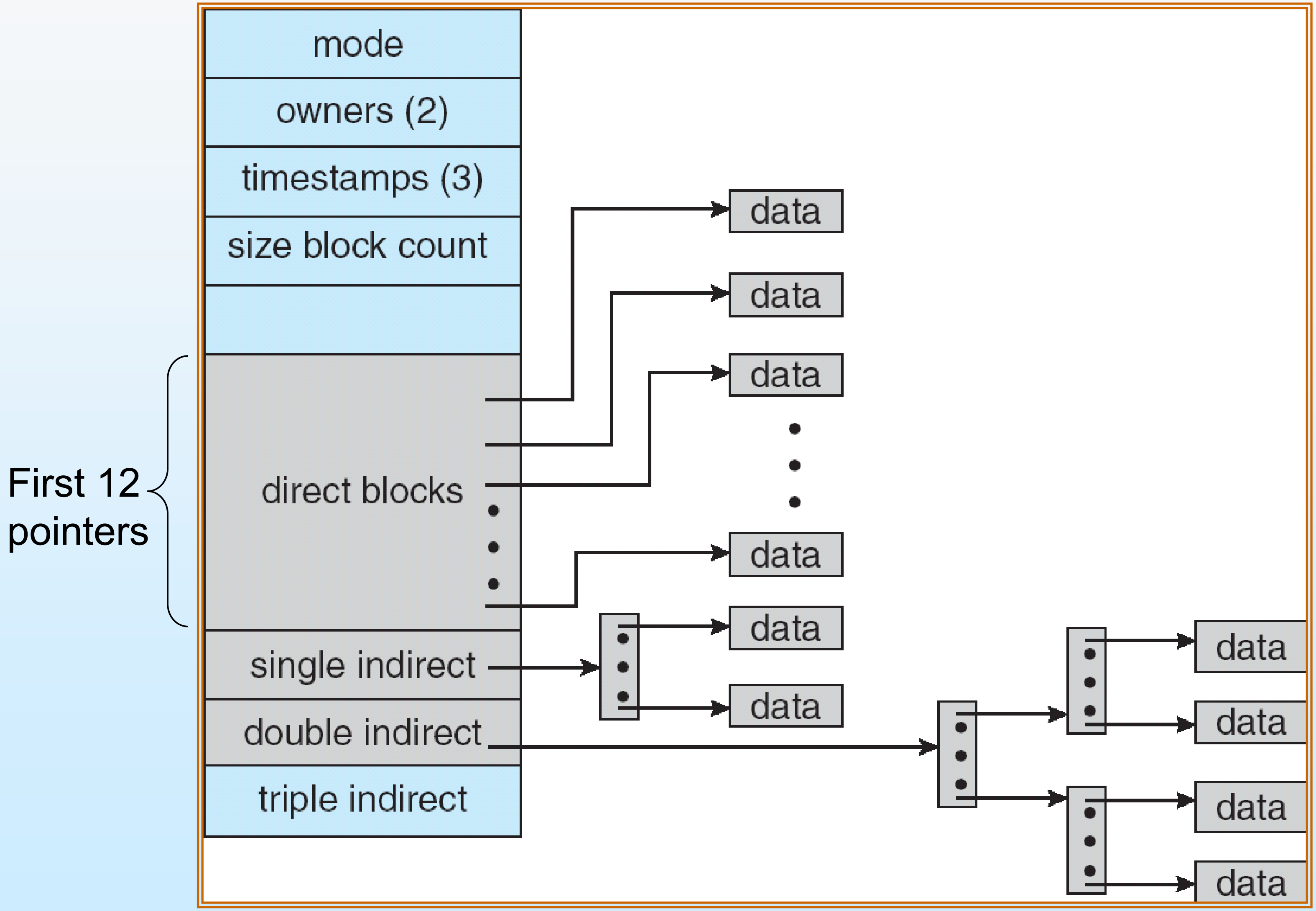
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of at least one index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words.
  - How many blocks we need for the index table?
- Mapping from logical to physical in a file of unbounded length (block size of 512 words).

How large the index block should be?

- Linked scheme – Link blocks of index table (no limit on size).
  - Last word of the index block is a *nil* or a pointer.
- Two-level index
  - What is the maximum size of the file? Assume the disk block is 512 Bytes and a pointer needs 4 bytes.



Combined Scheme: UNIX (4KB per block)



Performance Comparison

- Contiguous Allocation
  - One access to get a disk block
  - Maximum length has to be determined at the beginning of creation
- Linked Allocation
  - Direct Access needs *i* disk reads to access to i^{th} block
  - It is fine with sequential access
- Indexed Allocation
  - Index block requires considerable space
  - Index structure, file size determine the performance
- Combined approach
  - Using contiguous allocation for small file
  - Using indexed allocation for large file