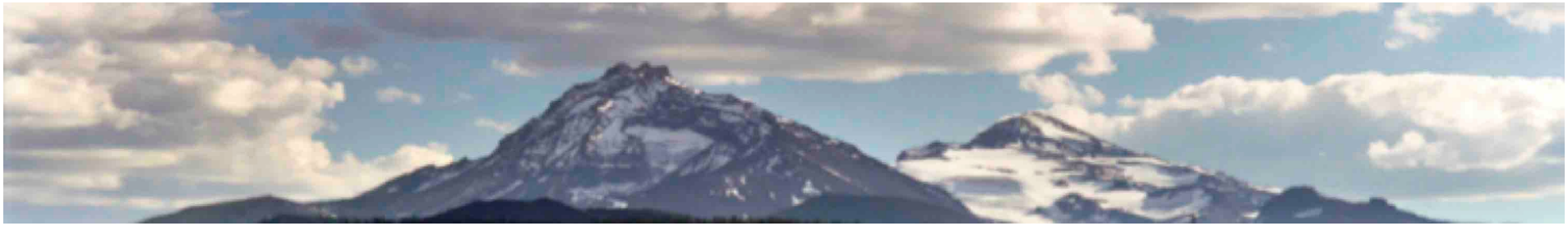


# CIS 415
# Operating Systems
# File System Implementation

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

# *Logistics*

- Read Chapter 14
- Final exam review session
  - Next Monday's office hour (1 pm) over Zoom
  - Attendance is optional
  - Recording will be provided

# *Outline*

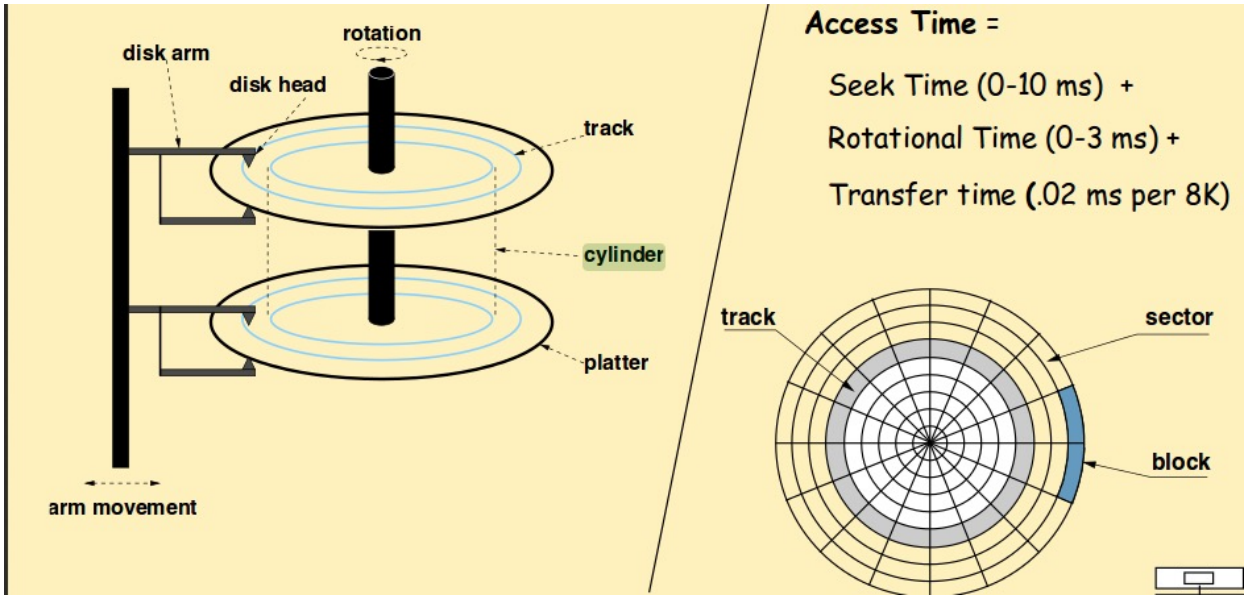
- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance

# *Objectives*

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

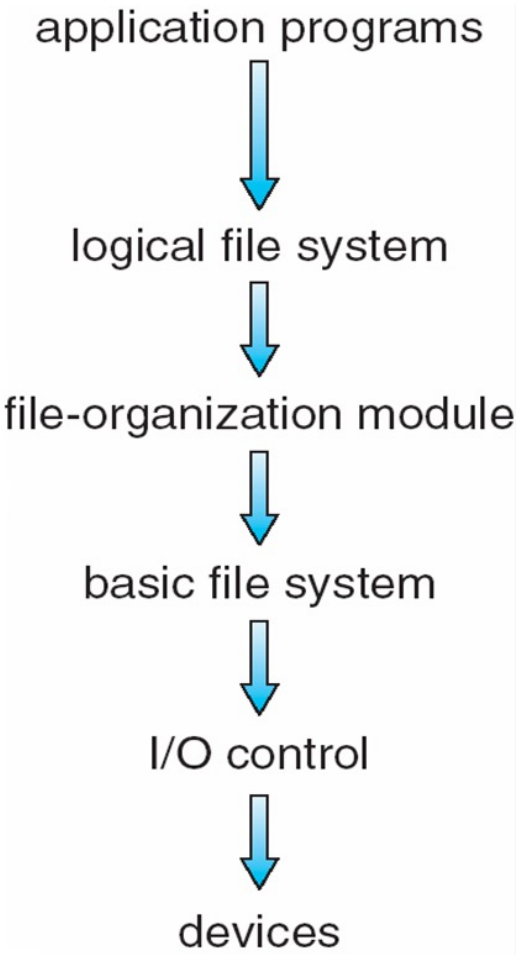
# *How is data stored on a disk?*

- **Sector:** Smallest addressable unit the *hardware* can read/write.
  - Historically 512 bytes; now often 4K bytes
  - Disk controller talks in sectors: "read sector #12345".
- **Block** : Smallest unit *OS/filesystem* uses for allocation.
  - Typically, a power-of-two multiple of the sector size (e.g., 4 KB, 8 KB, 16 KB).
  - The OS talks in blocks: "allocate 1 block for this file", "read block #200".

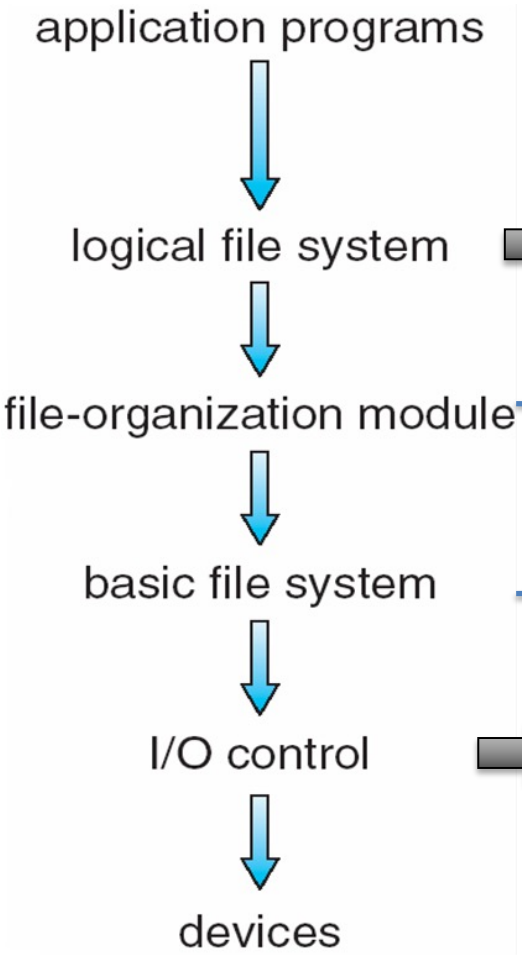


# *Layered File System*

- file system = hierarchical organization
  - Well-defined layers
  - Supports abstraction between layers
- Start with the physical layer and work up
- Use abstraction to support upper-layer functionality while hiding lower-layer complexity



# *File System Layers*




Virtual File System

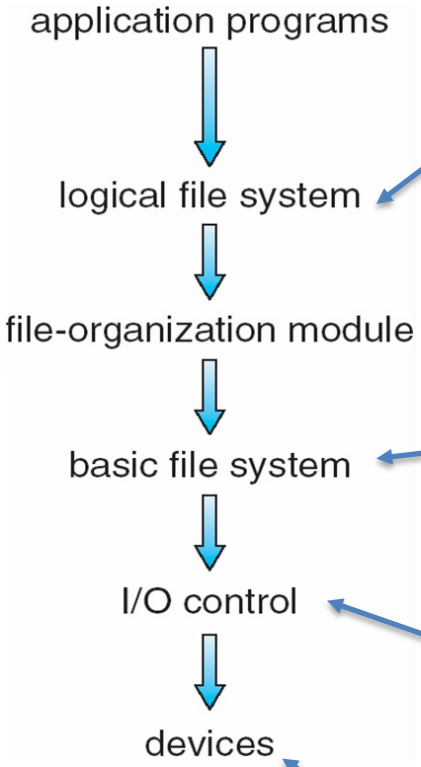
High-level interface and logical components of the file system

Physical File System

OS-level routines and code to manage the physical file system

Device Drivers

# *File System Layers*




Metadata: file-system structure, directories, FCB.

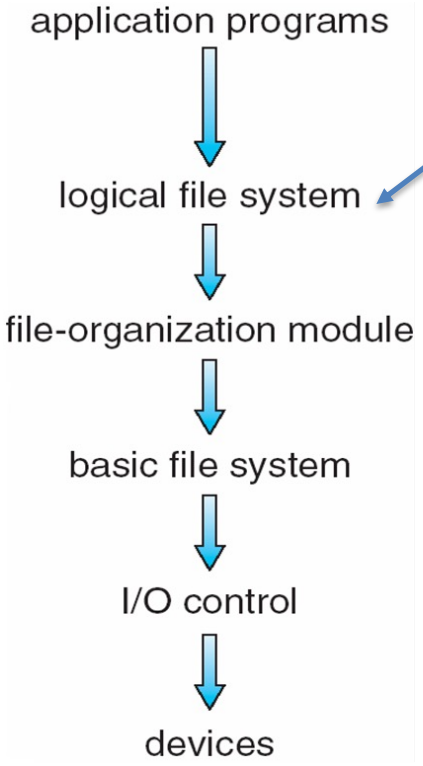
Manages files, logical blocks, and physical blocks, disk allocation, free spaces.

Manages disk at the block level only

Manages memory buffers and caches during operation
- Buffers hold data in transit from I/O device to file layer in OS
- Caches in I/O devices can hold recently requested data

- Low-level operations: "read drive1, cylinder 72, track 2, sector 10, into memory location 1060."

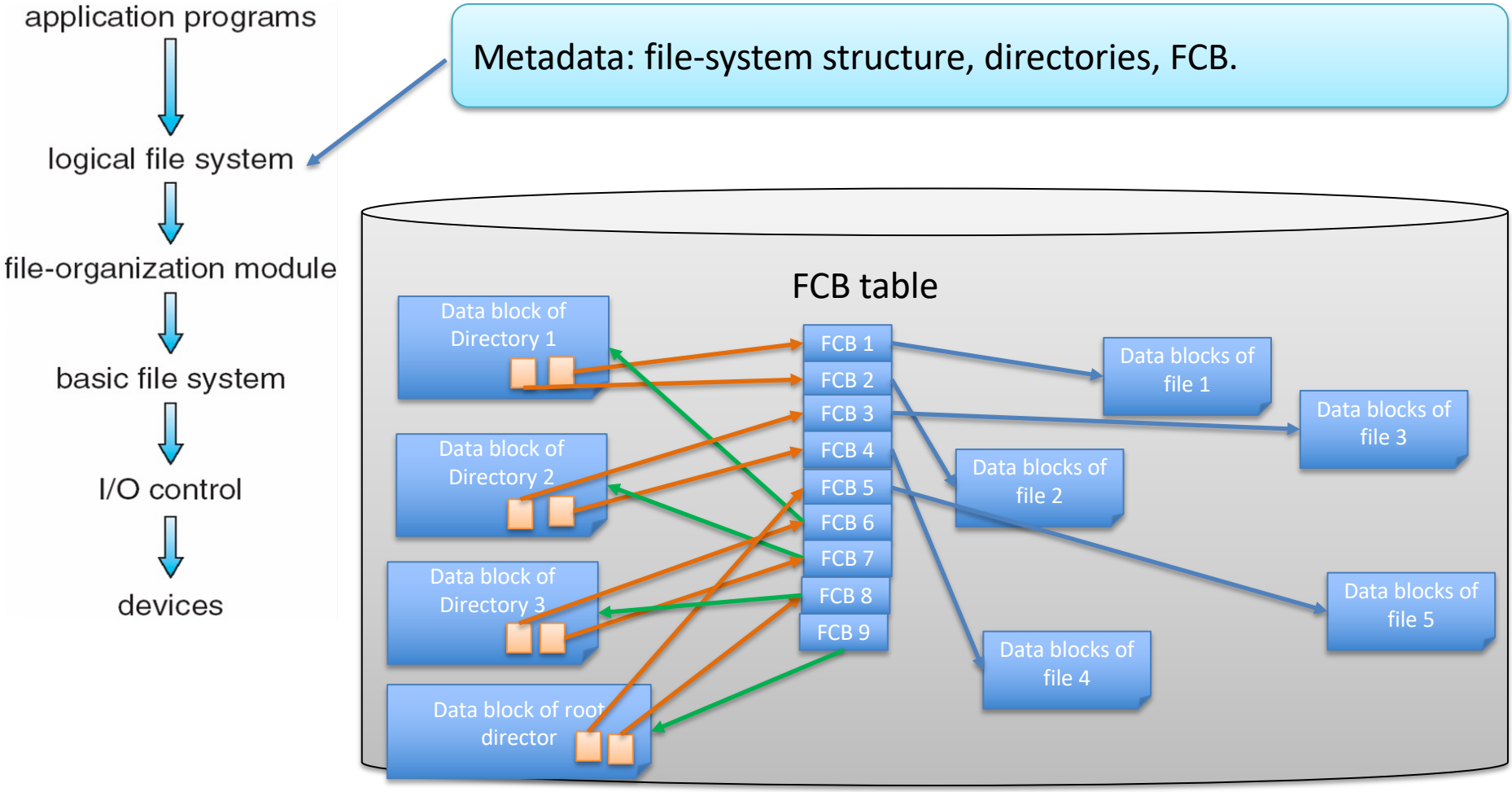
# *Logical file system layer*




Metadata: file-system structure, directories, FCB.

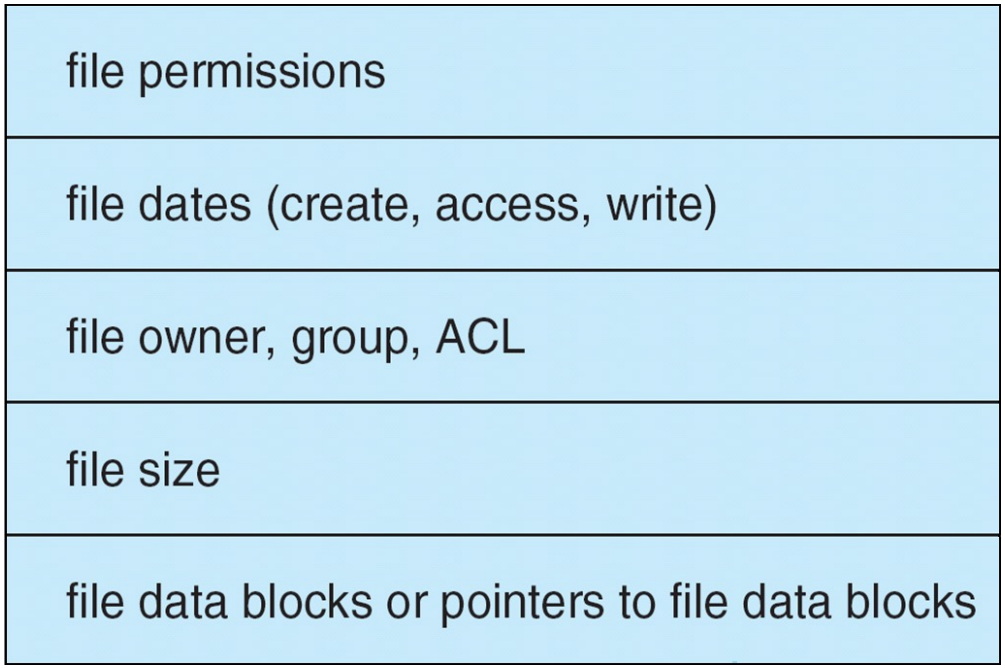
- *Logical file system* manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (FCB)
    - *inodes* in UNIX
  - Directory management
  - Protection

# *Logical file system layer*



# *File Control Block*

- Per-file *File Control Block (FCB)* (stored with the file)
- Logical file system's representation of a file
- Contains many details about the file
  - permissions, size, dates, …
- In Unix, called an *inode*

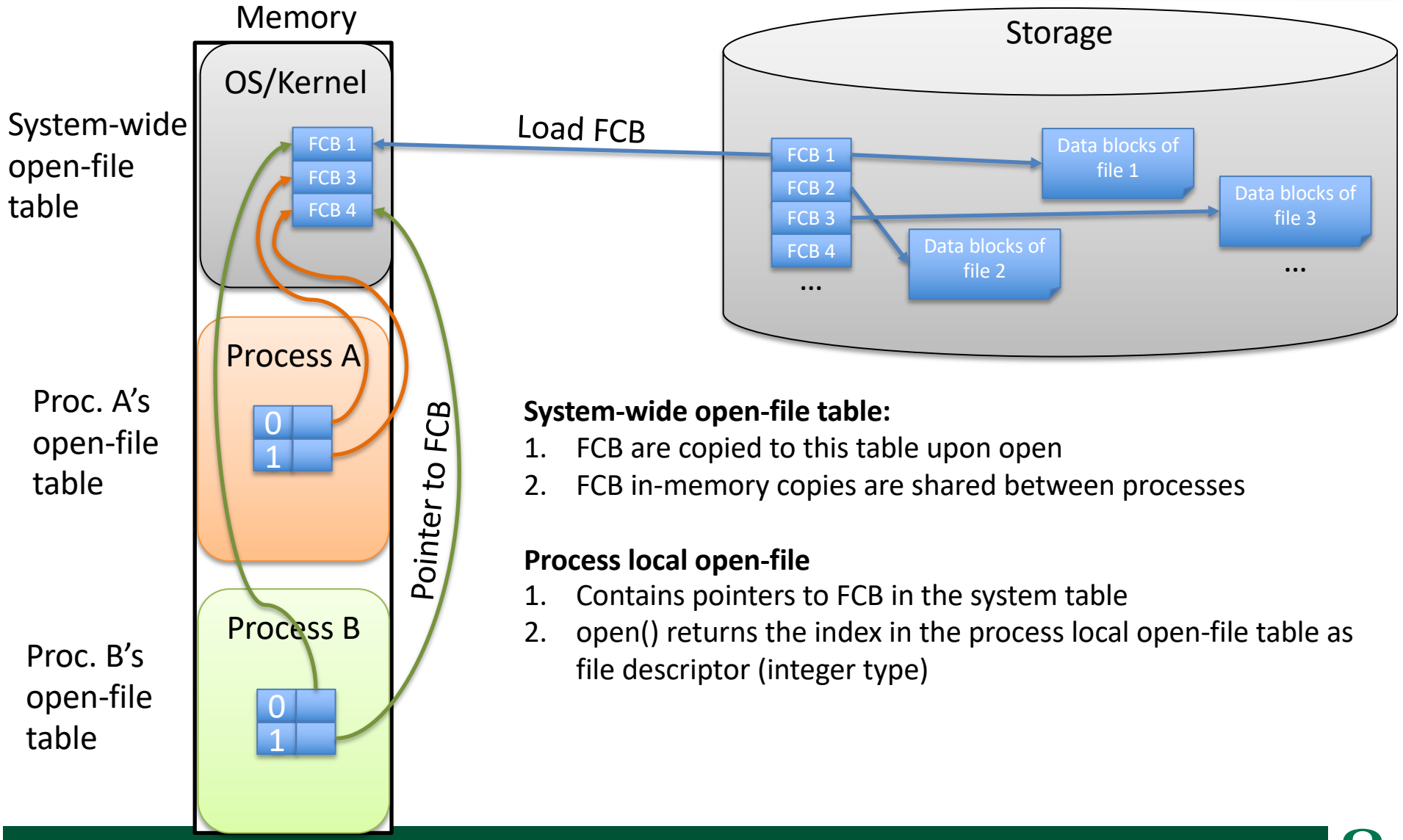



ACL: Access Control List

# *File System Structures – In Memory (1)*

- In-memory structures
  - Partition table
    - current mounted partitions (i.e., that make up file systems)
  - Directory structure
    - information on recently accessed directories
  - System-wide, open file table
    - all currently open files
  - Per-process, open file table
    - all of a given process's open files

# File System Structures – In Memory (2)




**System-wide open-file table:**

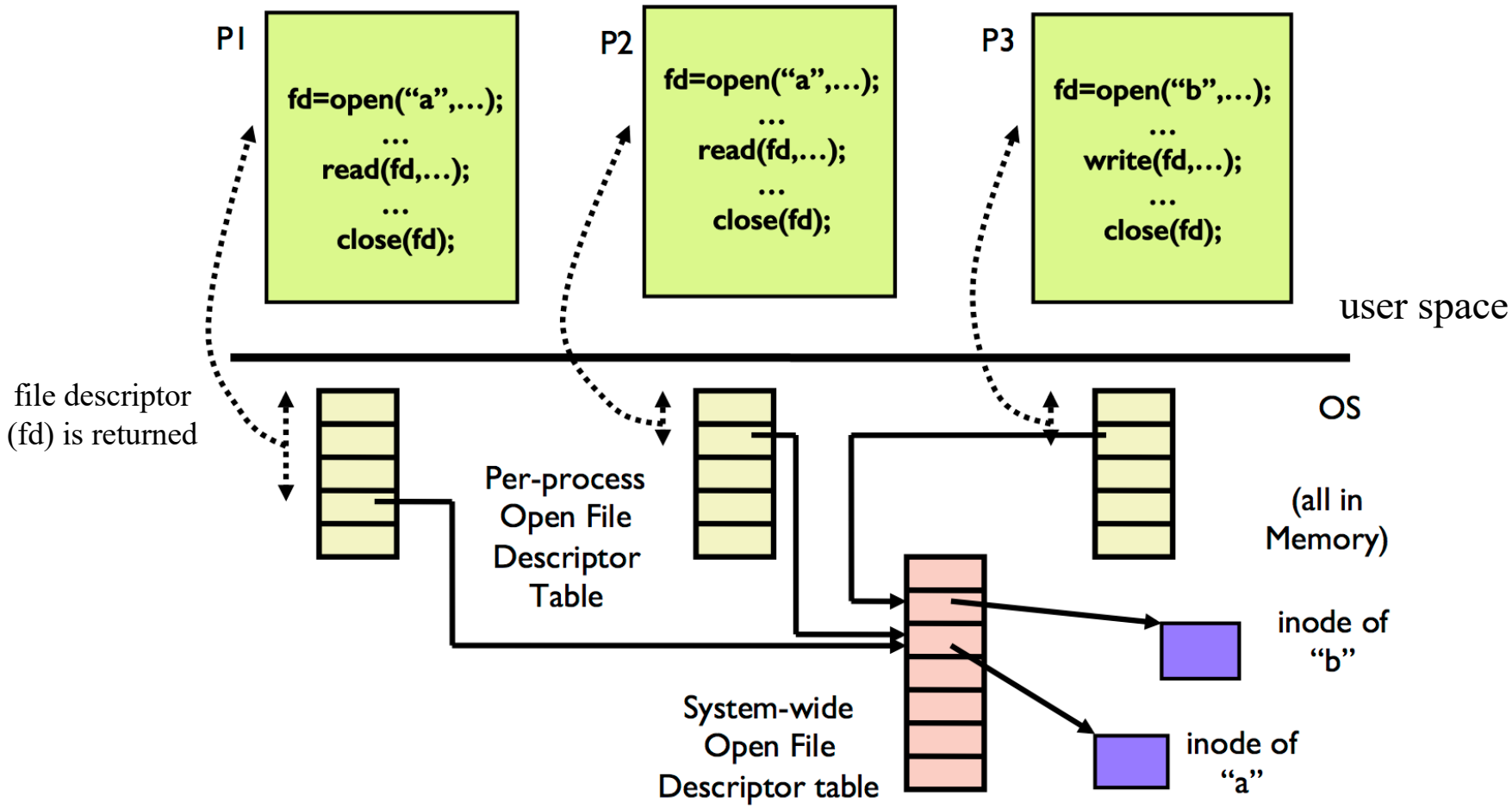
1. FCB are copied to this table upon open
2. FCB in-memory copies are shared between processes

**Process local open-file**

1. Contains pointers to FCB in the system table
2. open() returns the index in the process local open-file table as file descriptor (integer type)

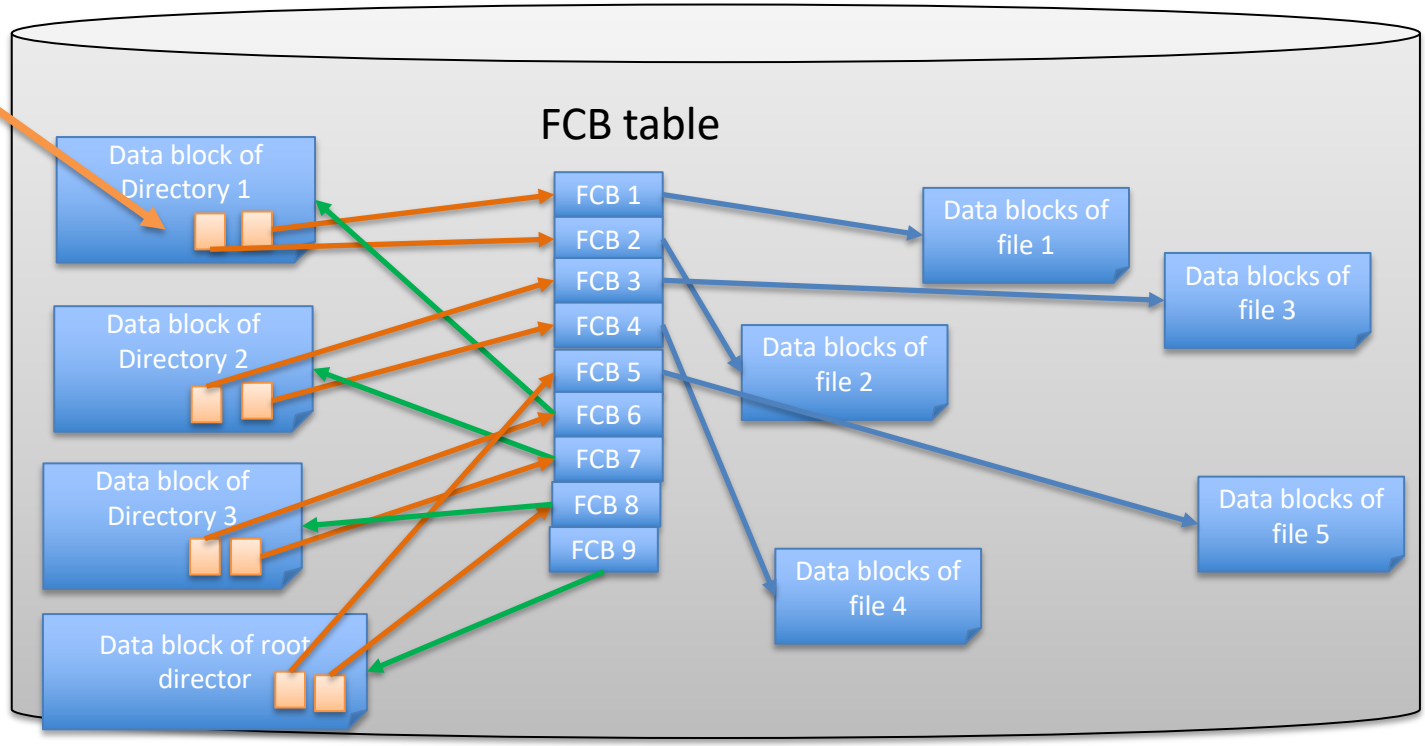
# *File System Structures – In Memory (3)*

- For *open()* syscall



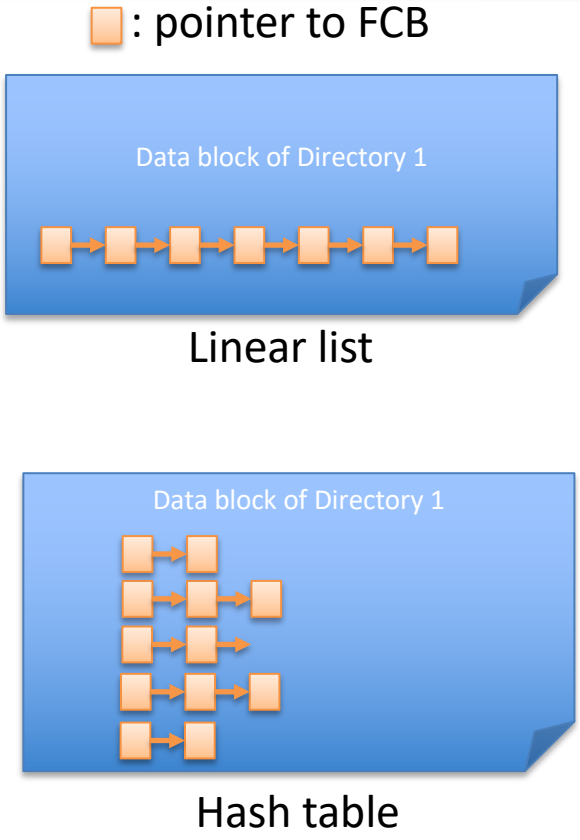
# *Directory Implementation*

How to implement a directory that holds points to FCBs?

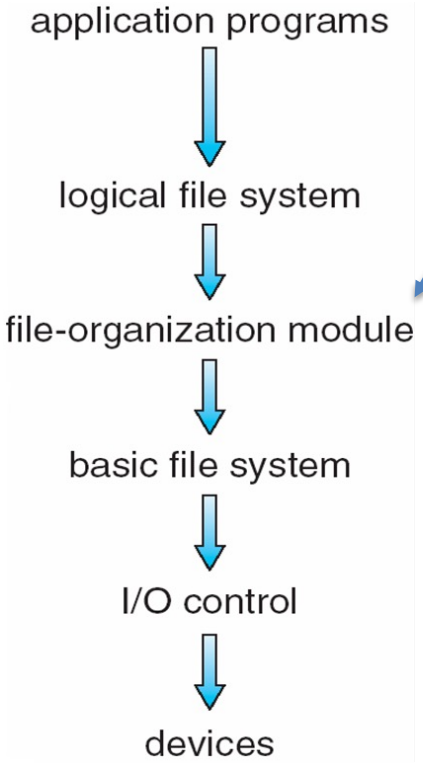


# *Directory Implementation*

- Linear list
  - Simple to program
  - Time-consuming to execute
    - linear search time
    - could keep ordered via linked list or use tree
- Hash table
  - 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



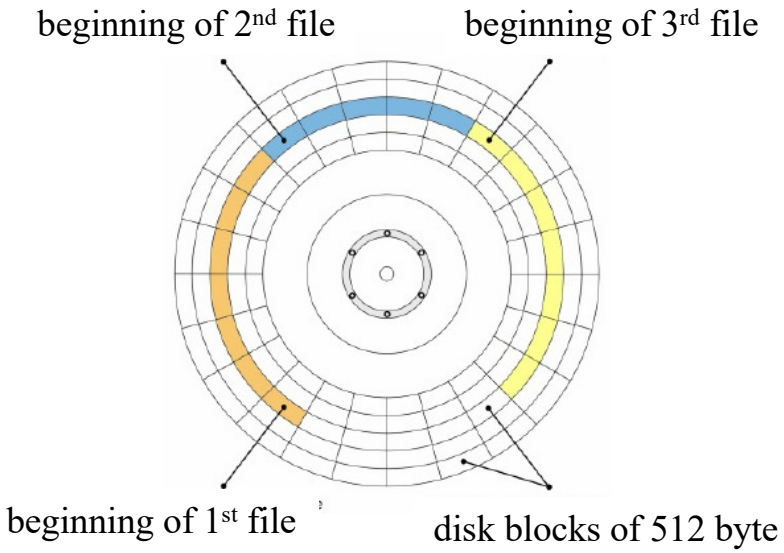
# *File organization module*




Manages files, logical blocks, and physical blocks, disk allocation, free spaces.

# *File System Implementation – Allocation*

- View the disk as a logical sequence of blocks
- A block is the smallest unit of allocation.
- Issues:
  - How do you assign the blocks to files?
  - Given a file, how do you find its blocks?




Implementation options:

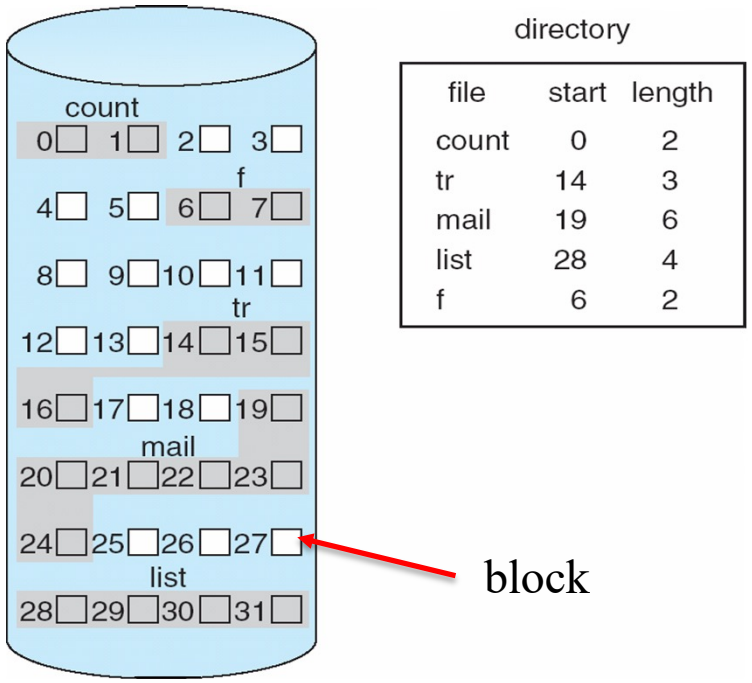
- Contiguous
- Non-contiguous
  - linked
  - indexed

# *Contiguous Allocation*

Sequential access: good
Direct (random) access: good
Fragmentation: high
Storage overhead: low

- Each file occupies a set of contiguous blocks
- Advantages:
  - Simple: starting location + length (number of blocks)
  - Best performance in most cases
- Disadvantages:
  - File size has to be known a priori
    - need to find space for the file of that size
  - External fragmentation
  - Need for compaction

How to better handle fragmentation without compaction?



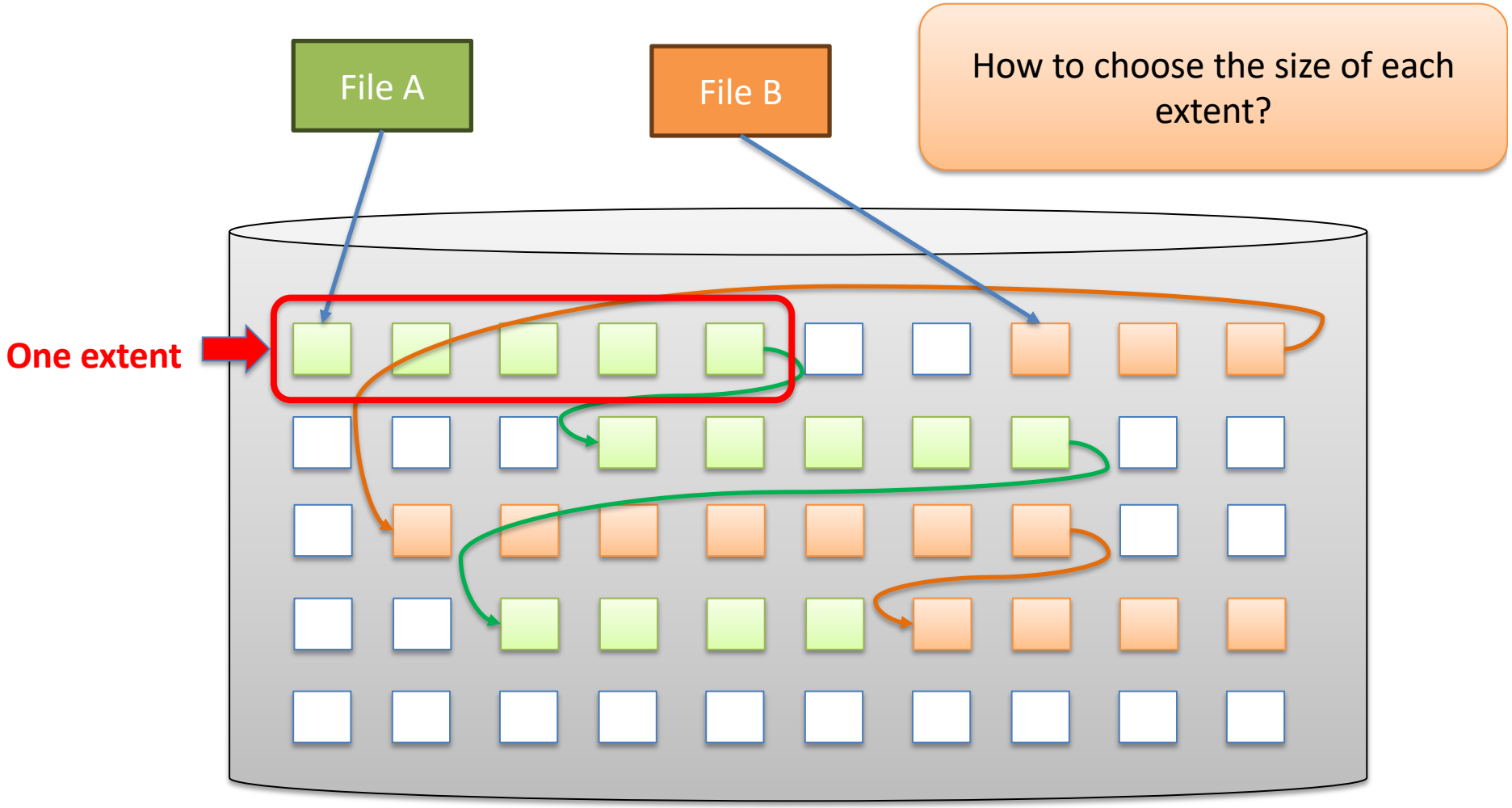

directory

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

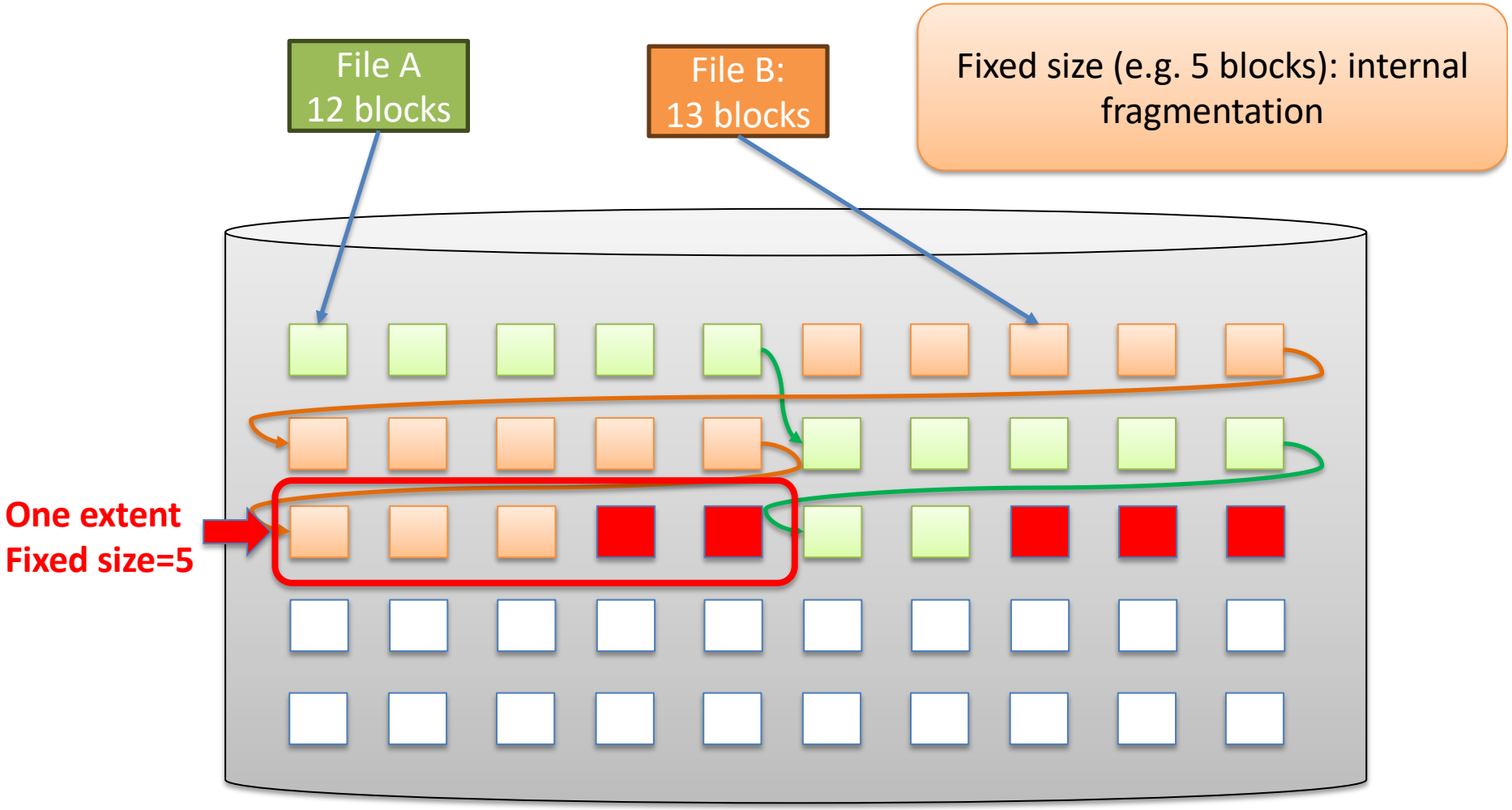
## *Modified contiguous allocation: Extent-Based Systems*

- One extent: one contiguous allocation
- Extent-based systems allocate disk blocks in extents and then links extents with each other
  - If an extent is not enough for a file, another extent is allocated and linked
  - A file consists of one or more extents
  - Internal fragmentation can be a problem
  - If extents are different sizes, external fragmentation could occur
- Why not just link blocks?

# *Extent-Based Systems*



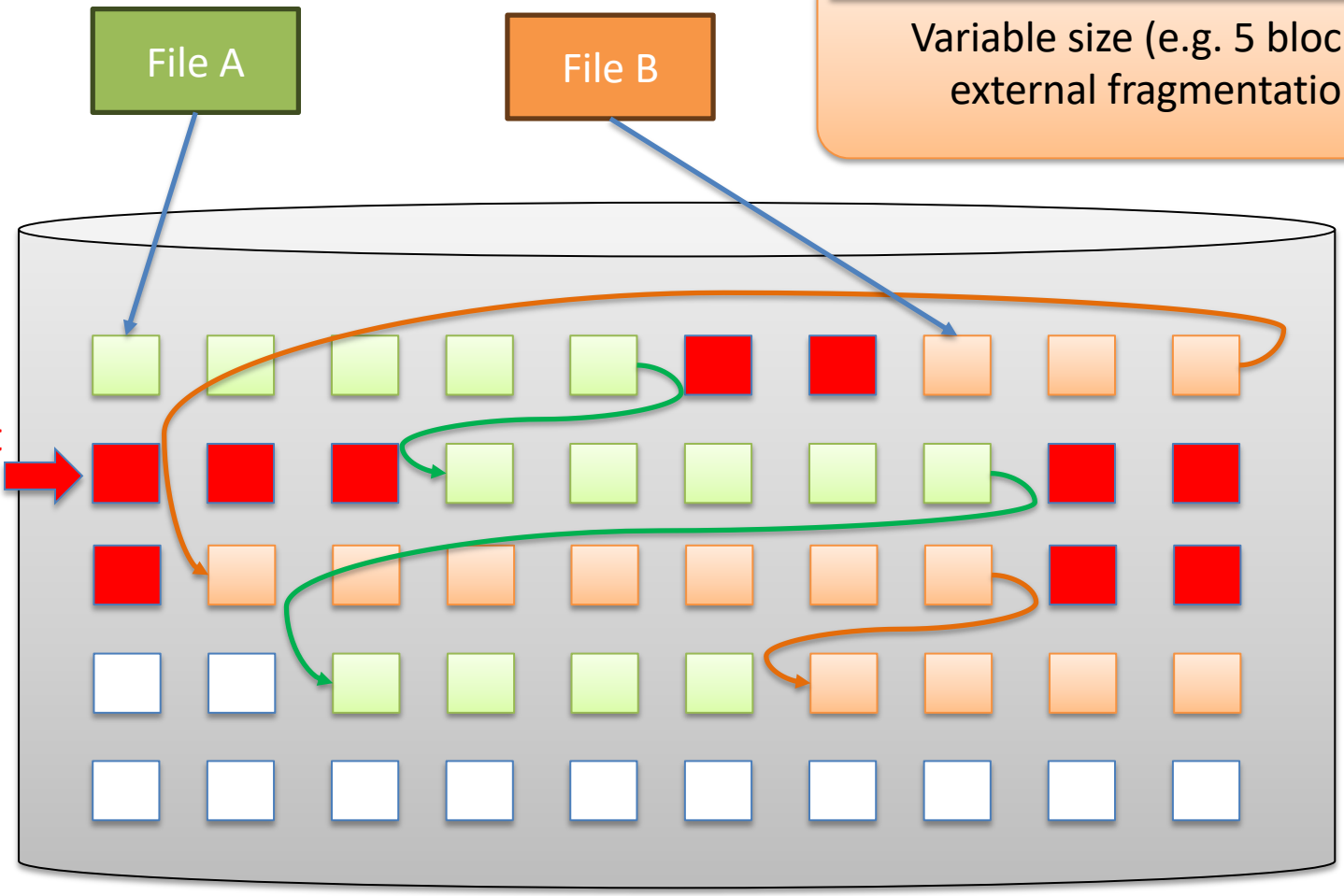
# *Extent-Based Systems*



# *Extent-Based Systems*

Sequential access: good
Direct (random) access: moderate
Fragmentation: moderate
Storage overhead: low

Variable size (e.g. 5 blocks): external fragmentation

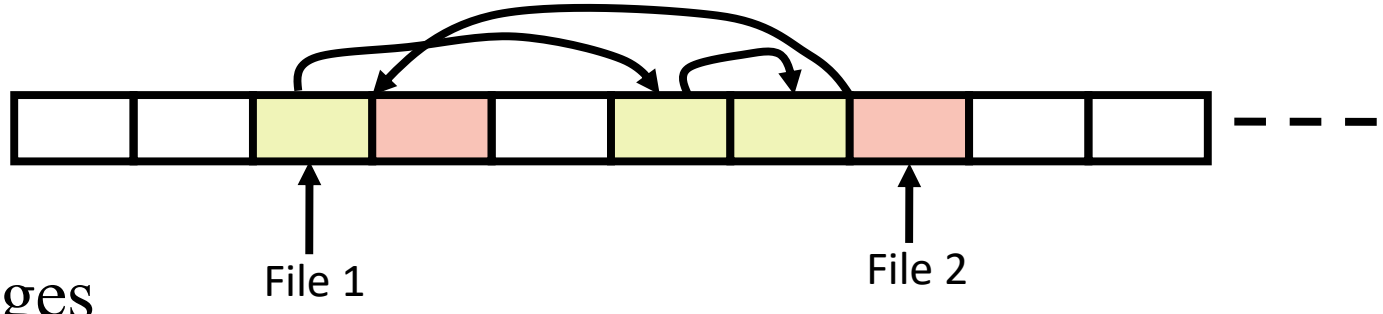



These may not be able to be used

Why not just link individual blocks (extent size fixed to 1)?

# *Linked Allocation (1)*

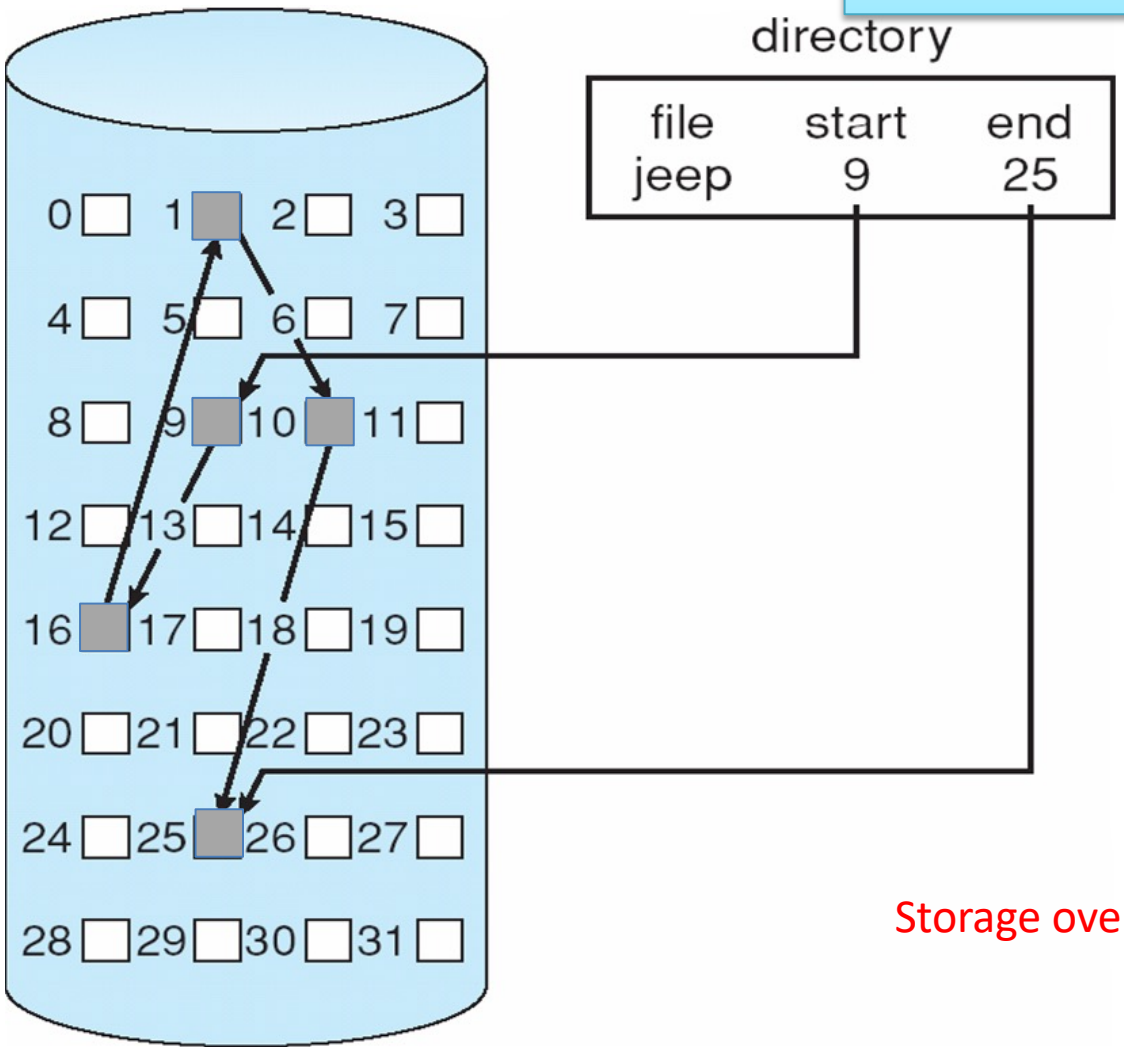
- Linked allocation allocates a linked list of blocks to a file
  - Keep a pointer to the first block of a file
  - Each block contains pointer to next block
  - File ends at NULL pointer
  - Free space management system called when new block needed




- Advantages
  - No external fragmentation
  - No compaction
- Disadvantages
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks

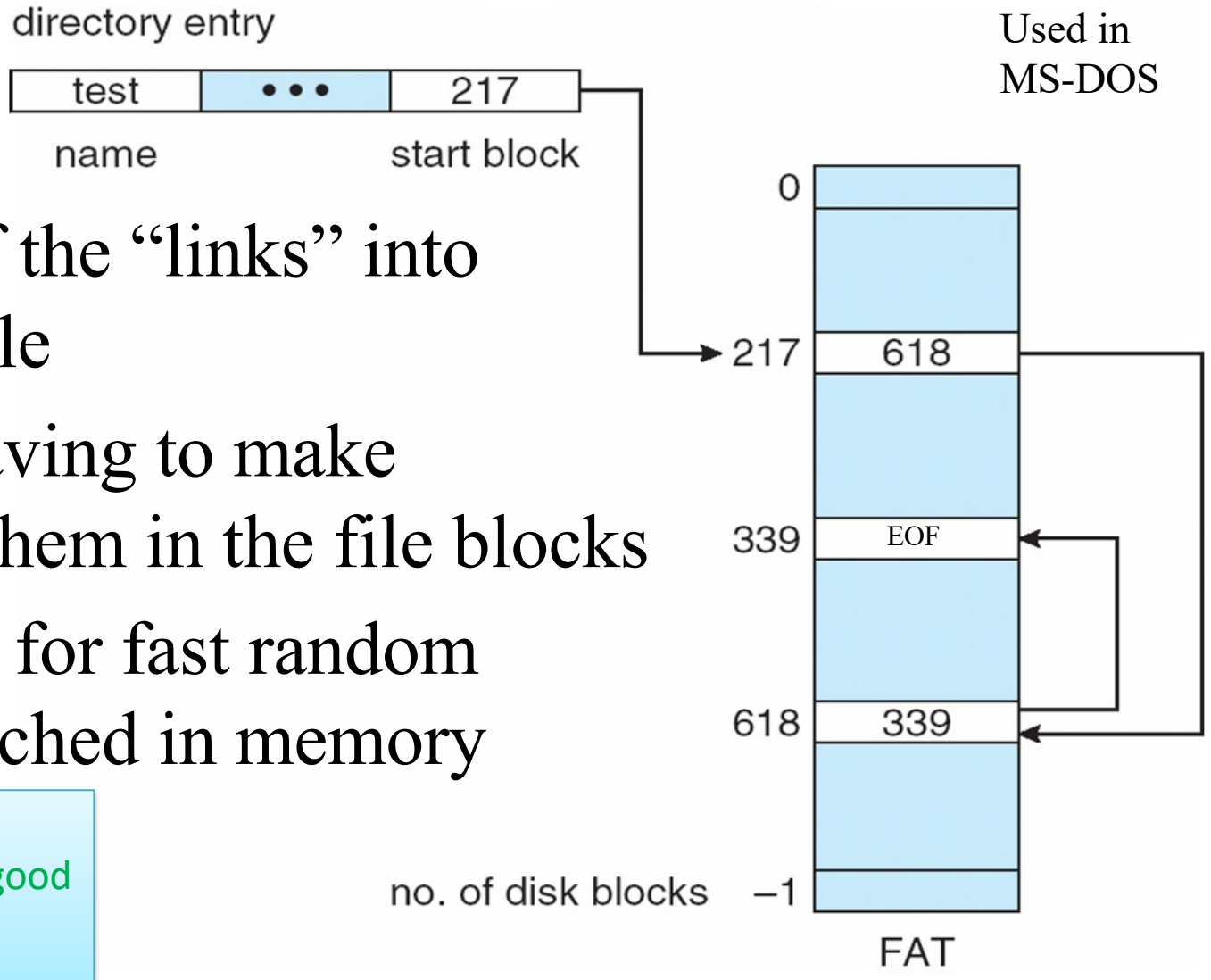
# *Linked Allocation (2)*

Sequential access: good
Direct (random) access: poor
Fragmentation: none
Storage overhead: high




Storage overhead?

# *Modified: Linked Allocation: File Allocation Table (FAT)*



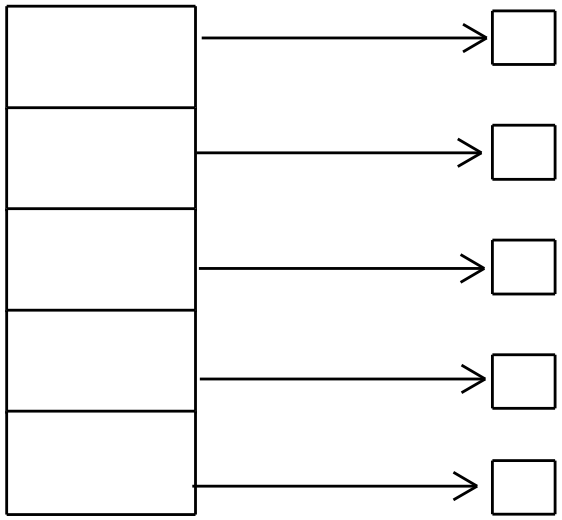

- Bring all of the "links" into a single table
- Relieves having to make spaces for them in the file blocks
- FAT allows for fast random access if cached in memory

Sequential access: good
Direct (random) access: good
Fragmentation: none
Storage overhead: high

# *Indexed Allocation*

- *Indexed allocation*
  - Each file has its own *index block(s)* of pointers to ALL its data blocks

- Logical view



index table

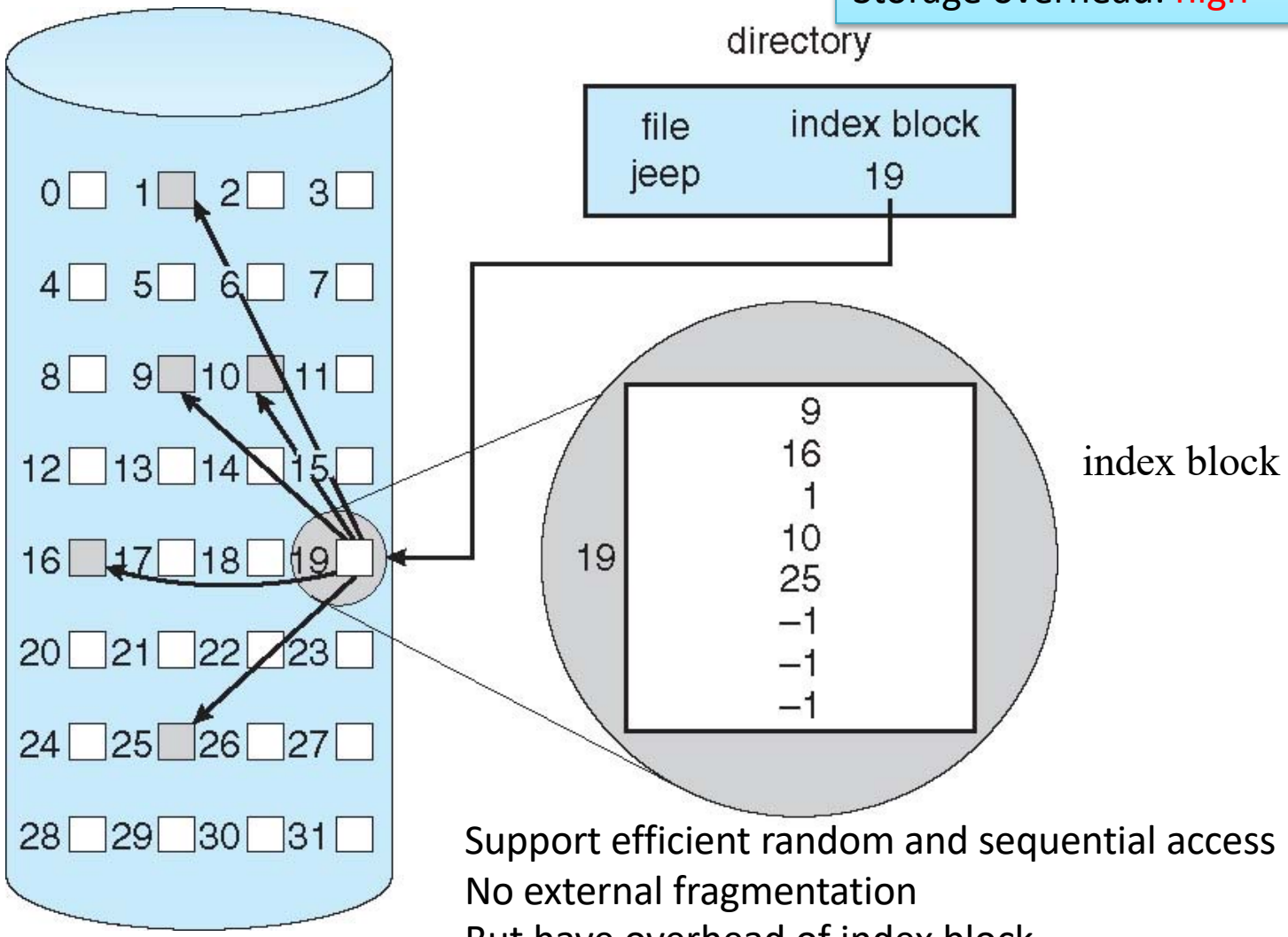
Looks similar to a page table

Where to store?

How to store?

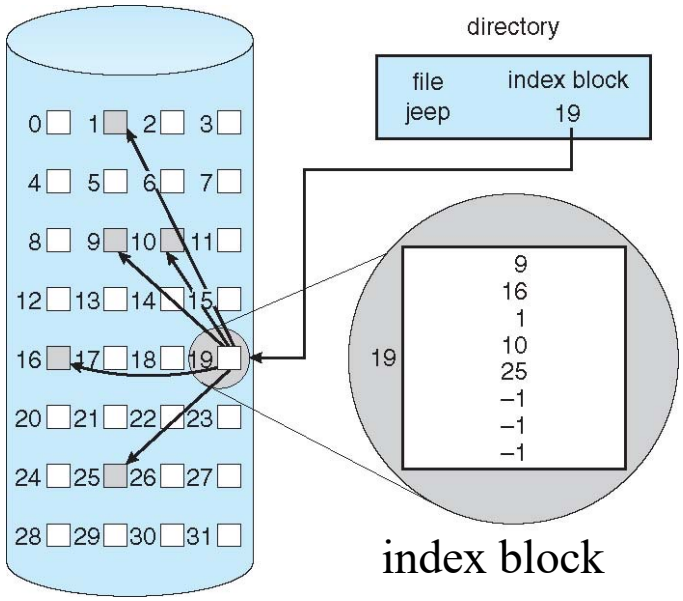
# *Example of Indexed Allocation*

Sequential access: good
Direct (random) access: good
Fragmentation: none
Storage overhead: high




Support efficient random and sequential access
No external fragmentation
But have overhead of index block

# *Indexed Allocation - Implementation*




If each block is 4KB, and each pointer is 32 bit. What the maximum file size we can index?

Max number of indices 4KB/4B = 1K
Max file size = 1K * 4K = 4MB

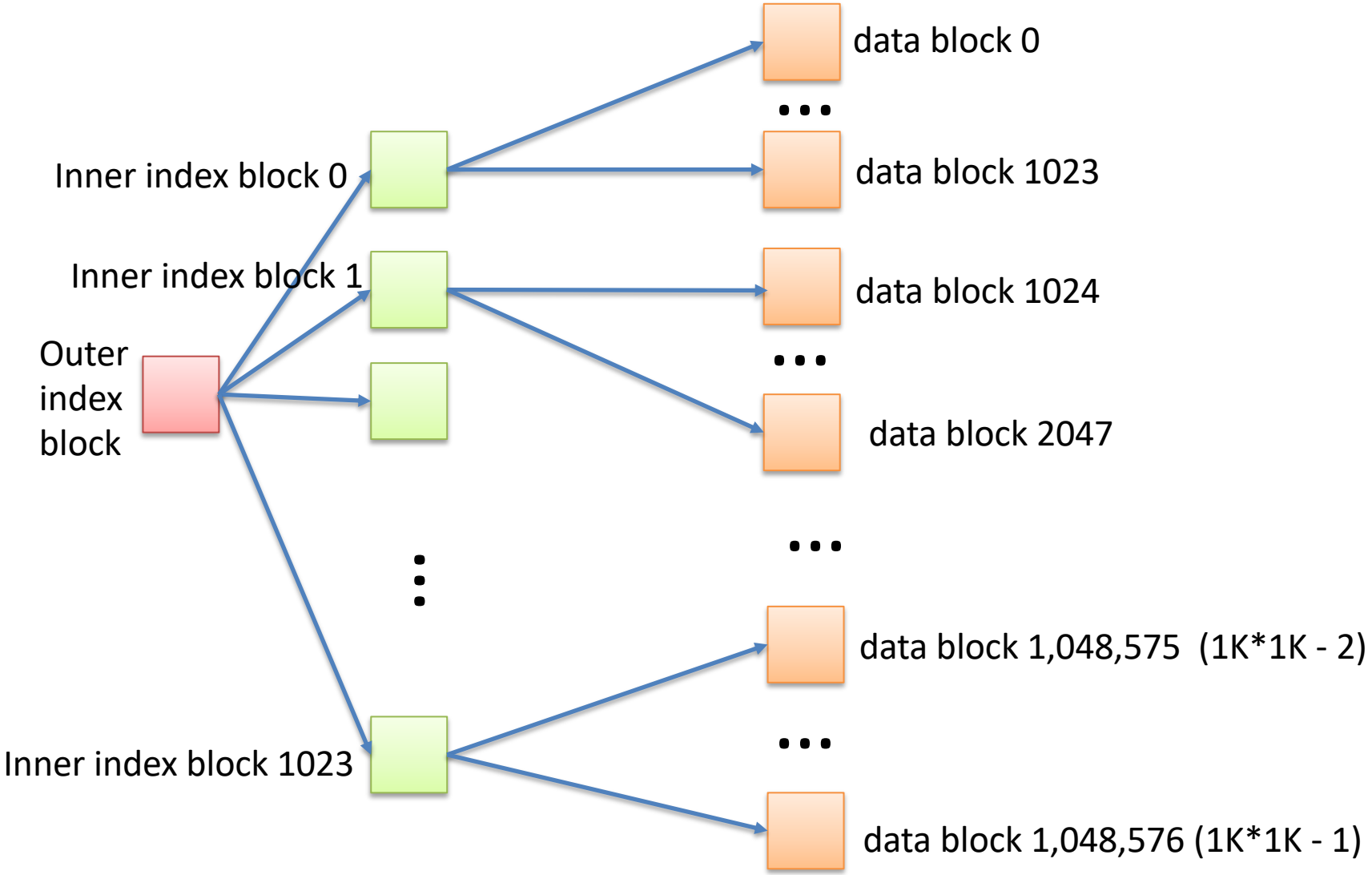
How to store a large file?

One index block is not enough; we need more index blocks
For a file of 4GB, we need 1K index blocks

But how to track all index blocks?

# *Two-level index allocation*

Each block = 4KB
Total 4 GB



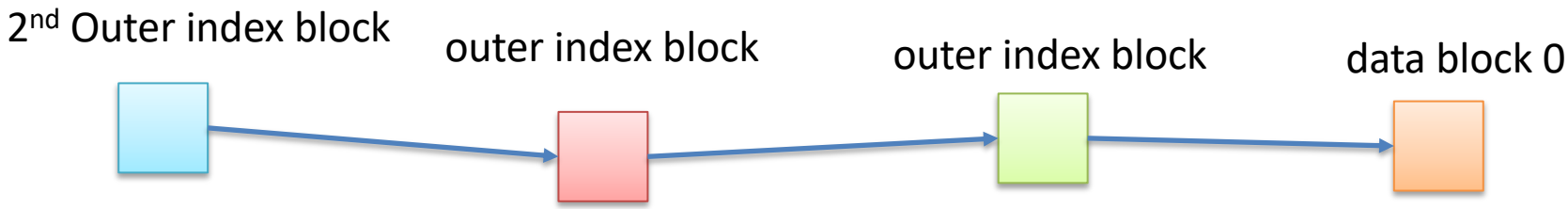
# *Two-level index allocation*

What if we want to store more than 4GB?

Multilevel index allocation

- Example 3-level: 4TB

What is the issue?




Need level + 1 accesses to fetch data.

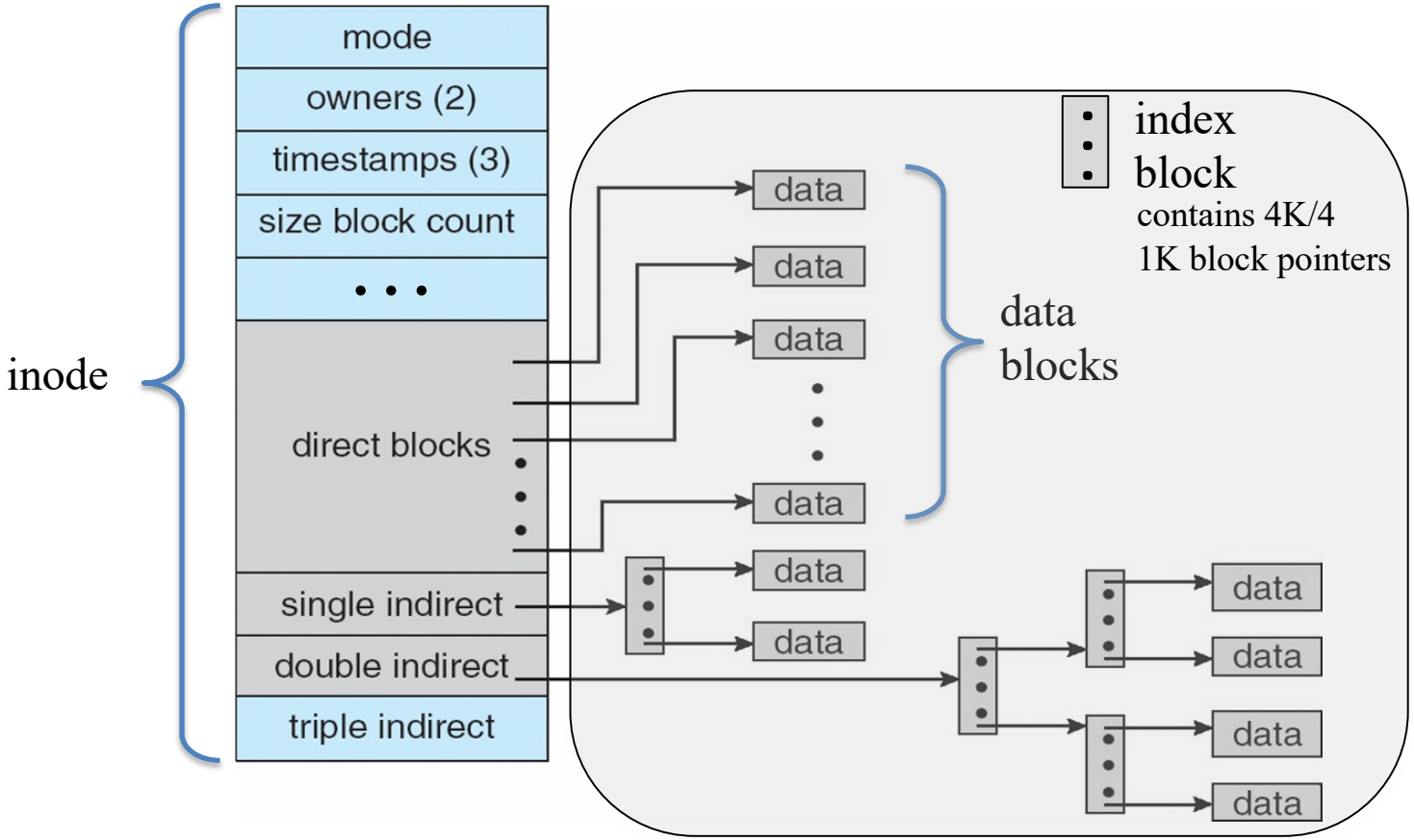
What if we want to store a small file?
This overhead is not necessary

How to be adaptive?

# *Combined Scheme – UNIX Inode*

4K bytes per block, 32-bit addresses (block pointer (ID))

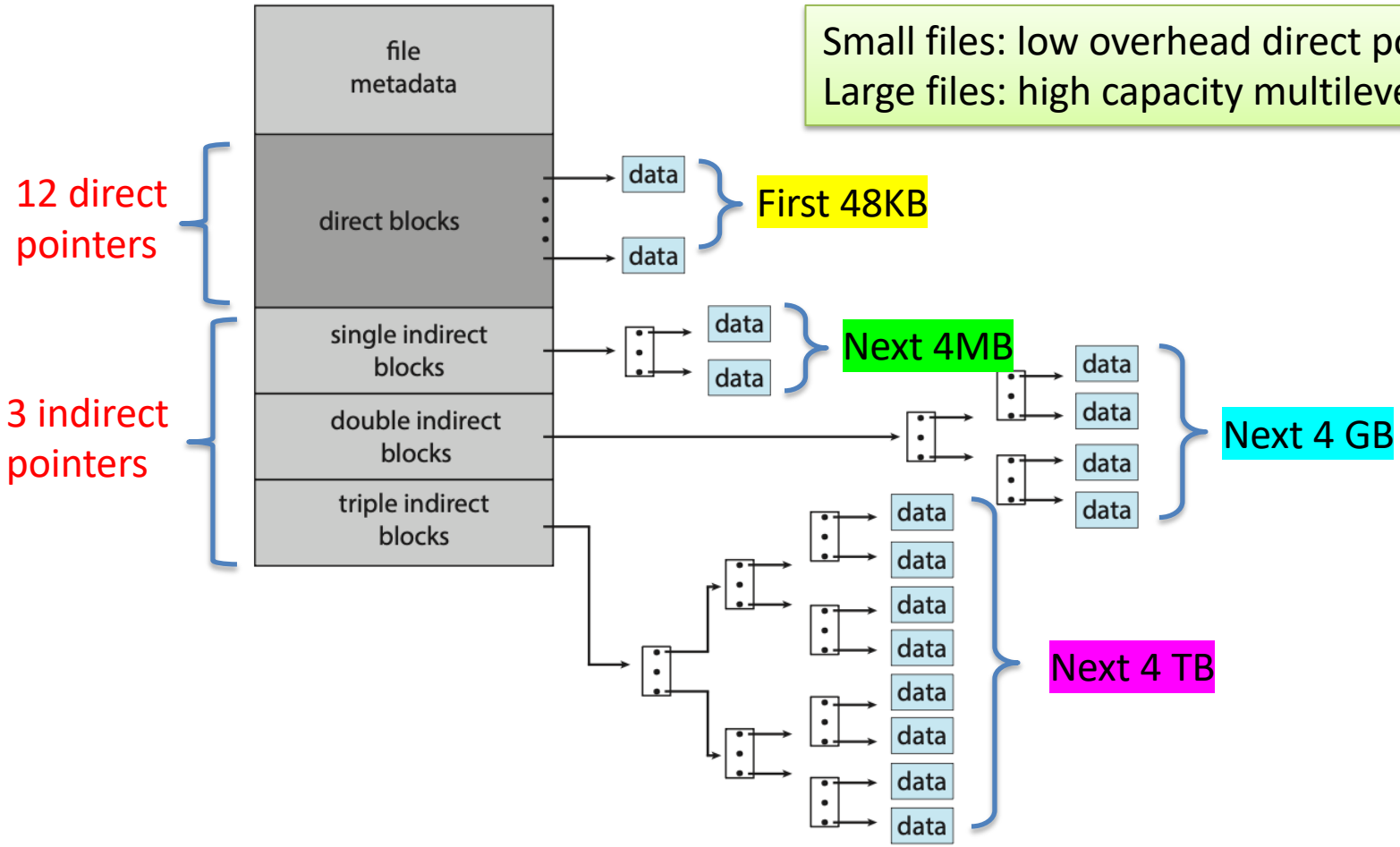
UFS: Unix File System



# *Combined Scheme – UNIX Inode*

4K bytes per block, 32-bit addresses (block pointer (ID))

UFS: Unix File System

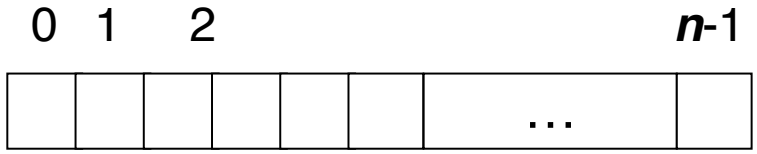


# *Storage Allocation - Summary*

- Best method depends on file access type
  - Contiguous allocation great for sequential and random
- Linked good for sequential, not random
  - Must start from 1st block and follow links to any location
- Declare access type at file creation
  - Select either contiguous or linked
- Indexed scheme is more complex
  - Single block access could require L times index block reads then data block read
  - L = number of levels

# *Free-Space Management (1)*

- File system maintains *free-space list* to track available blocks/clusters
  - Using term "block" for simplicity
- Bit vector or bit map (*n* blocks)




bit[*i*] = { 1 ⇒ block[*i*] free; 0 ⇒ block[*i*] occupied }

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word of first "1" bit

# *Free-Space Management (2)*

- Bit map requires extra space

  block size = 4KB = 212 bytes

  disk size = 240 bytes (1 terabyte)

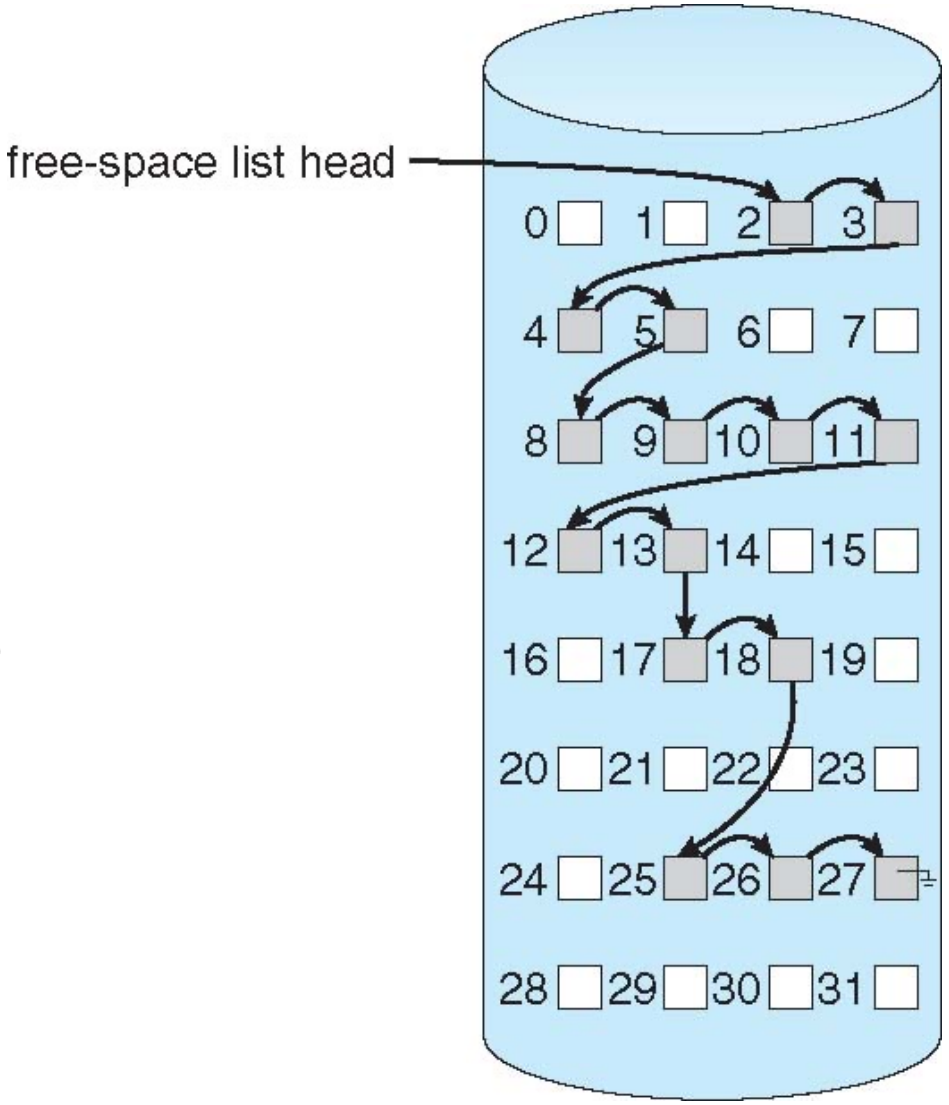
  n = 240/212 = 228 bits (or 32MB)

  if clusters of 4 blocks -> 8MB of memory

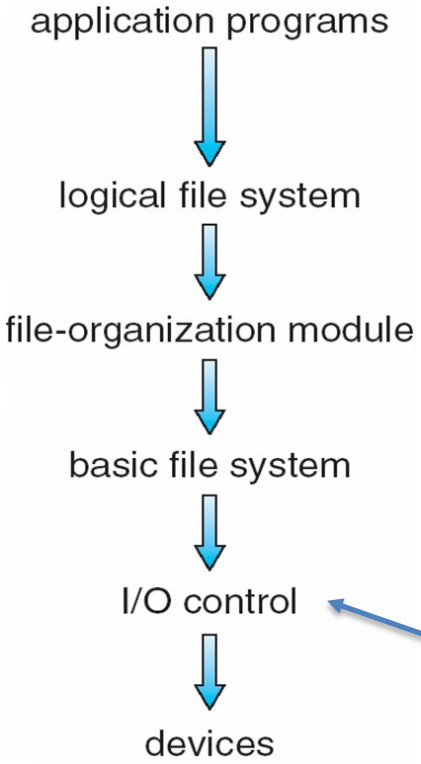
- Easy to get contiguous files

# *Linked Free Space List on Disk*

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list
    - if # free blocks recorded)



# *File System Layers*

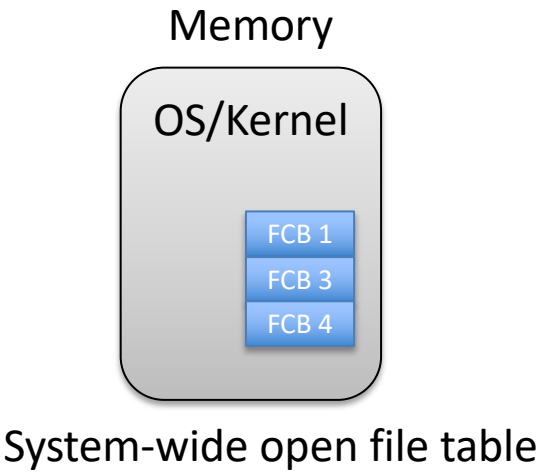



Manages memory buffers and caches during operation

- Buffers hold data in transit from I/O device to file layer in OS
- Caches in I/O devices can hold recently requested data

# *Inode Caching*

- Imagine searching through the inodes each time you do a read() or write() on a file
  - Too much overhead!
- Ideally: cache the inode in memory for a file in memory and keep re-using it
- Open file descriptor table kept in memory does this for us

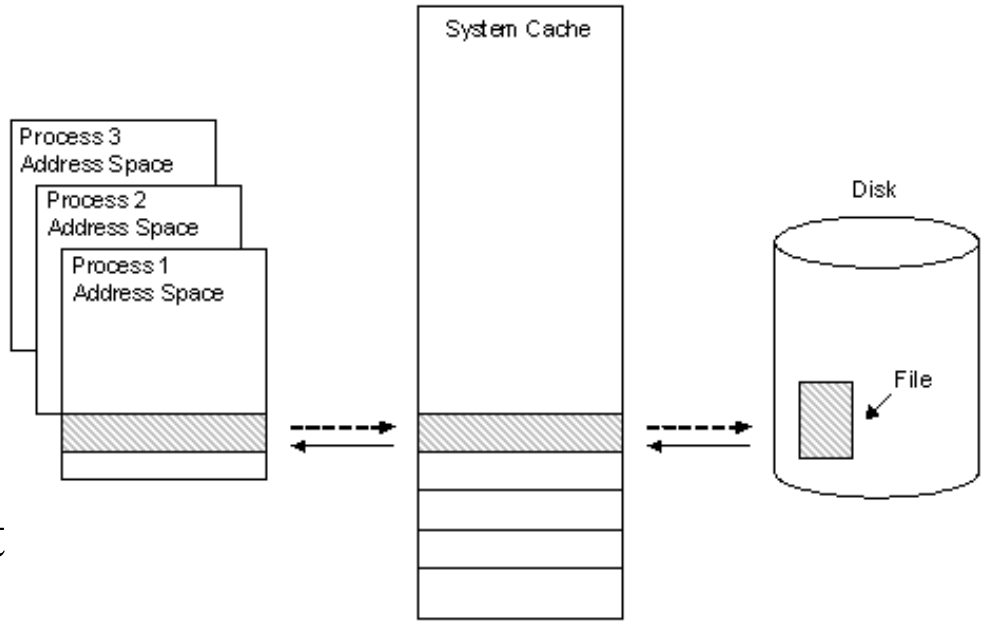


# *File Caching*

- Even if after all this (i.e. bringing the pointers to blocks of a file into memory), may not suffice since we still need to go to disk to get the blocks themselves.
- How do we address this problem?
  - Cache disk (data) blocks in main memory
  - Called file caching

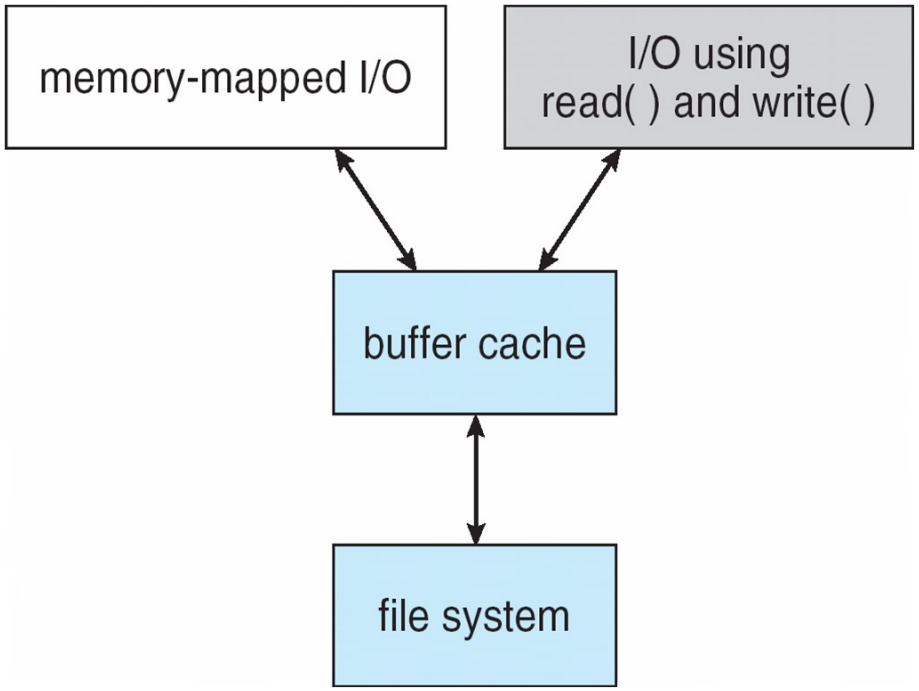
# *File Caching / Buffering (1)*

- Cache disk blocks that are in need in physical memory
- On a *read()* system call, first look up this cache to check if block is present
  - This is done in software
  - Look up based on block id.
  - Typically perform some kind of hashing
- If present, copy this from cache to user space
- Else, read block from disk, put data structure



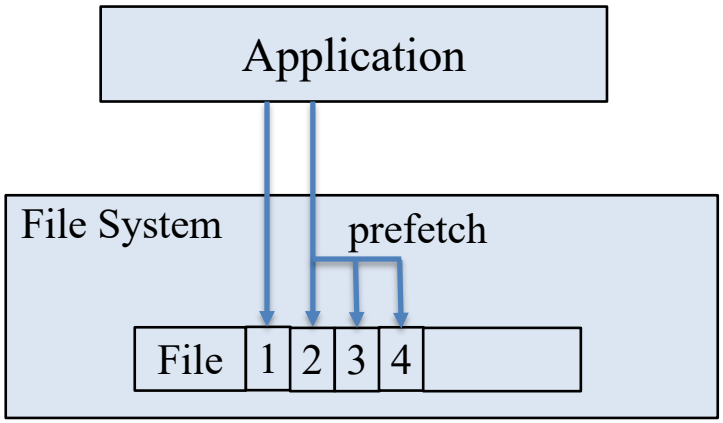
# *Unified Buffer Cache*

- A *unified buffer cache* uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid *complications*

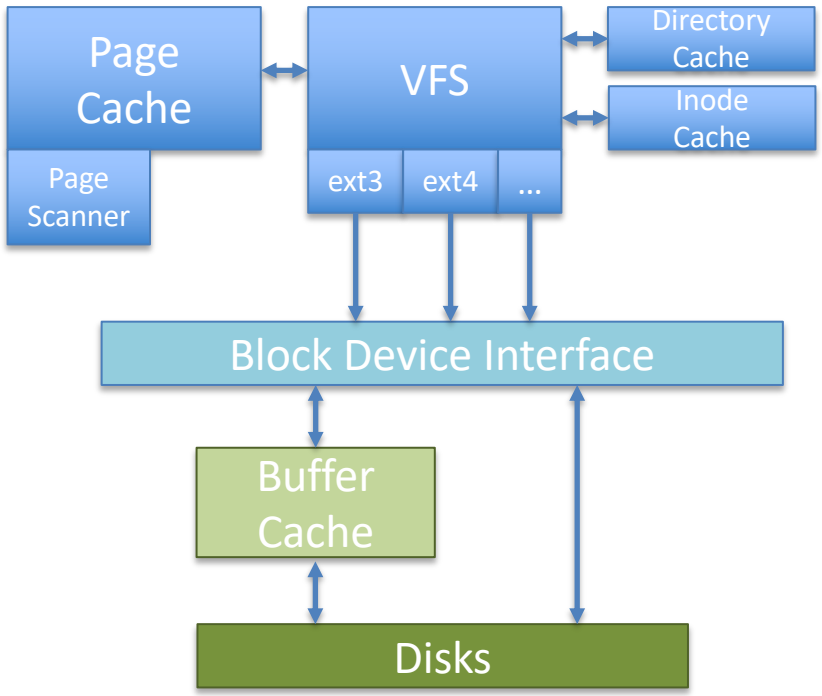


# *File System Prefetch*

- Often we are reading data sequentially (e.g., file system backup)
- If FS detects sequential workload based on current/previous I/O offsets, it can predict that the next reads will likely be sequential as well
- Populate the filesystem cache with these pre-fetches



# *Linux File System Caches*




*Page cache:* cache of virtual memory pages including memory-mapped files

*Buffer cache:* stores blocks from disk, now unified with the page cache (why?)

*Page scanner:* schedules dirty pages to be written to disk

*Dentry cache:* remembers mappings from directory entry to inode (for what purpose?)

*Inode cache:* stores inodes in hash table for fast lookup