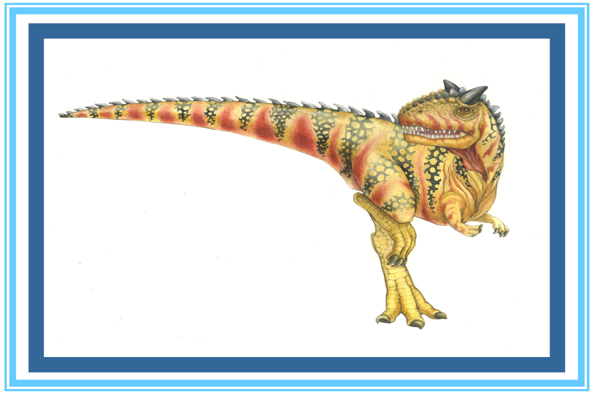
Chapter 10: File Systems

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

가천대학교 AI·소프트웨어학부
Gachon University

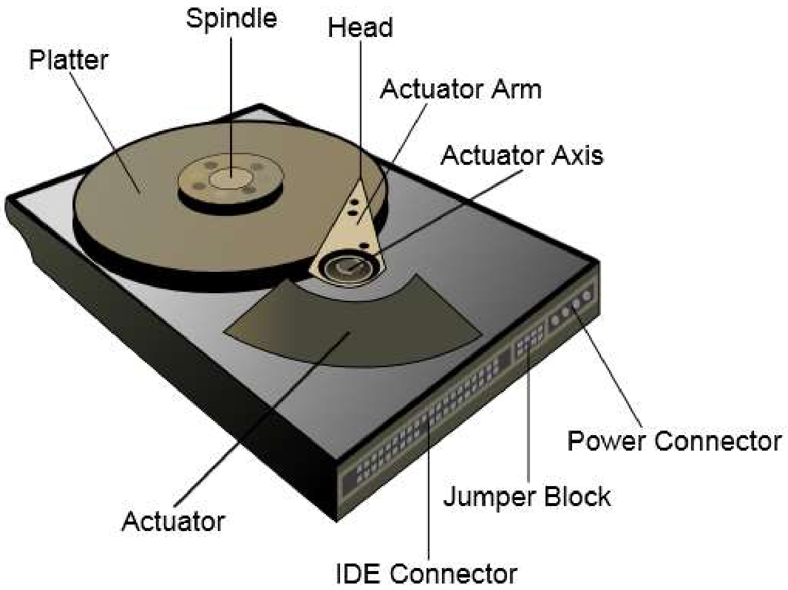
Objectives

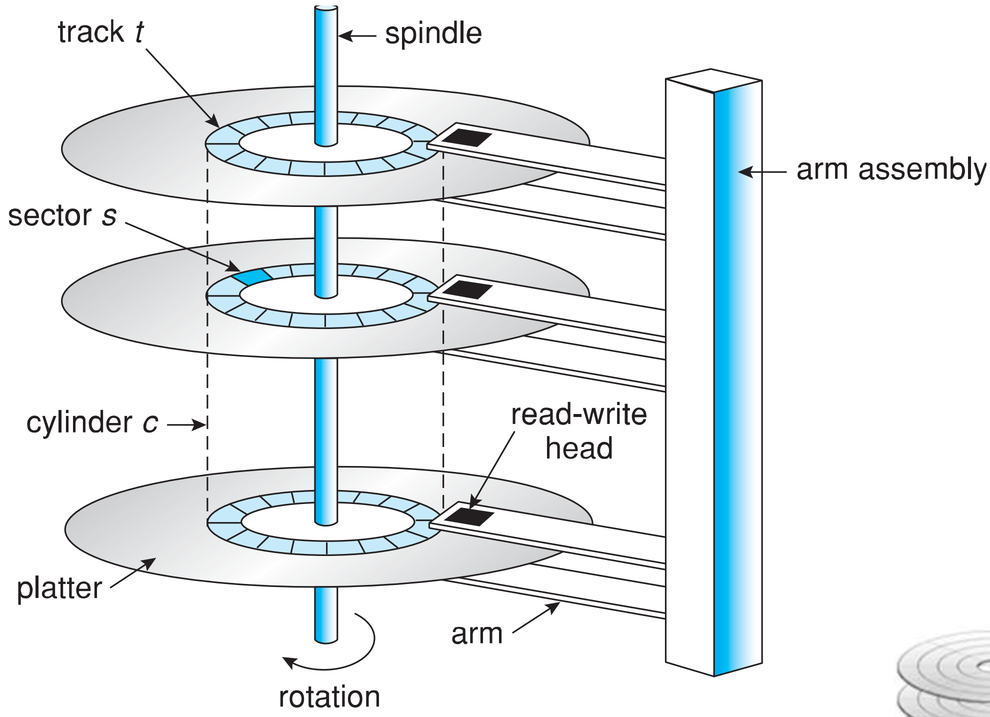
- To explain the concept of file systems
- To describe how file systems conduct file operations
- To discuss file allocation methods

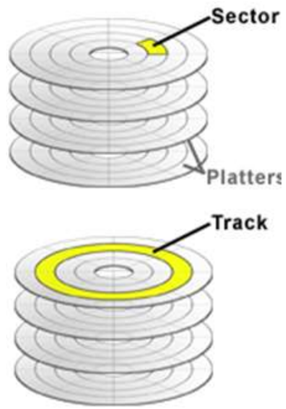
File Systems

- **File and File System Concept**
- File Operations
- File Access and Allocation Methods
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
- Free-Space Management

Physical Storage Unit: Hard Disks



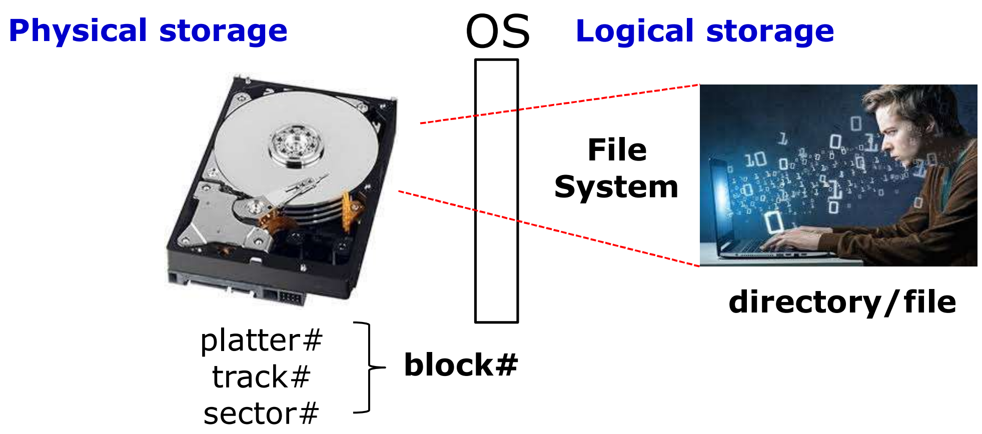




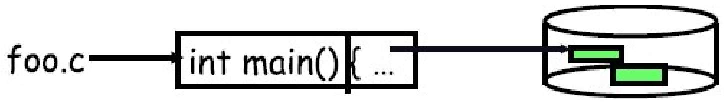
- Disk reads/writes in terms of **sectors** (not bytes)
    - e.g., read sector #143,212
- OS views disk as sequence of 1-dimensional **blocks**

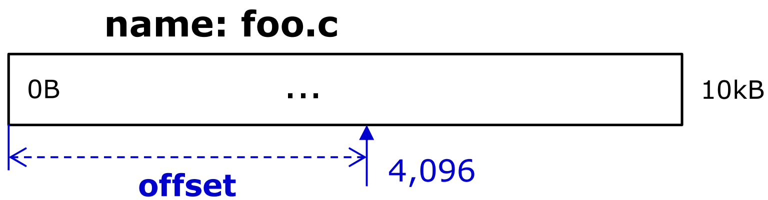
File Concept

- What is a **File** ?
  - Logical view of storage unit
  - **File system** (in OS) maps files to physical storage



- **File abstraction**:
  - **User's view**: **named sequence of bytes**



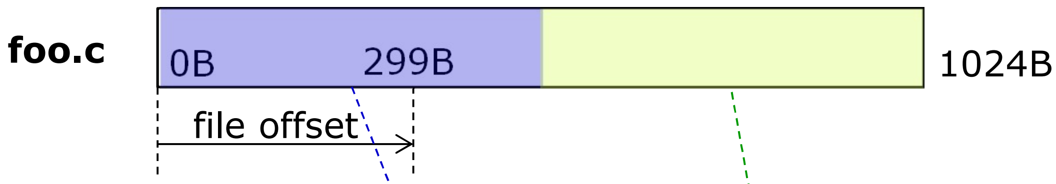


  - **File System's view**: collection of **disk blocks**
  - File system's job: translate name & offset to **disk blocks**:

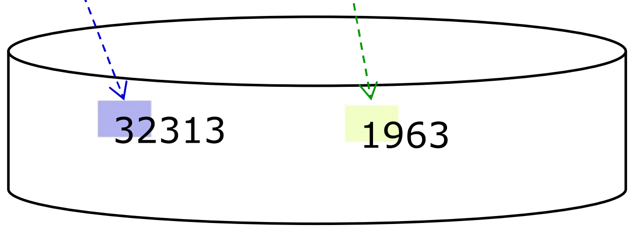
{file, offset} ⟶ FS ⟶ disk address

Internal File Structure

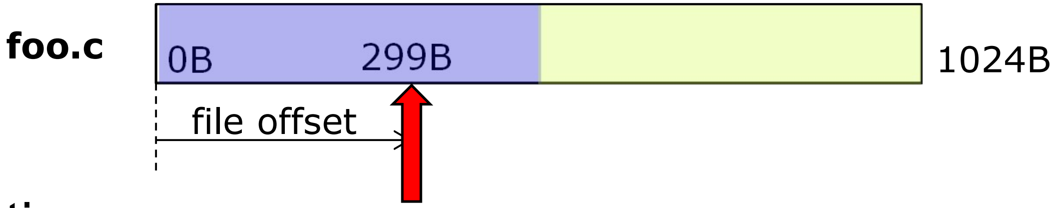
- **File**: Logical view
  - File is a **stream of bytes**
  - Each byte is addressable by its **offset** from beginning of file
    - e.g., Read 300th byte of file “foo.c” → read offset 299 of “foo.c”



- All disk I/O is performed in units of one **block** (=several disk sectors)
  - All basic I/O functions operate in terms of block
  - Covert logical records to physical blocks



File Information



- **File pointer**
  - When process read/writes a file, system tracks last read/write location
- Disk location of file
  - Actual file location on (hard) disk
    - e.g., **block number**
  - How to access?: Sequential/Direct access (next part)
  - How do allocate?: Disk allocation method (next part)

File Metadata (Ch. 10.1.1)

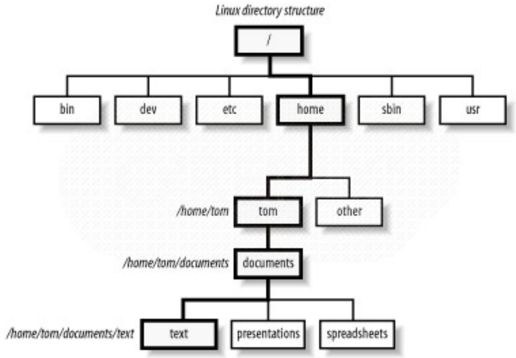
- **File Attribute** (= **Metadata**)
  - File Name
  - Identifier: File id. usually number.
  - Location: pointer to a device and location of the file
  - Size, time, date, ...

- **File control block** (**FCB**)
  - Contains information of file
    - e.g., **inode** in Linux

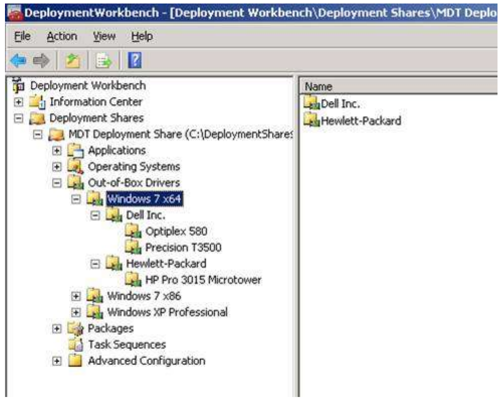
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

File Metadata: Directory

- **Directory**
  - A special file that contains list of file names
  - Typically forms a tree
  - For all file operations: <u>**first *search* directory**</u>!
    - **Root directory** is known to OS a priori

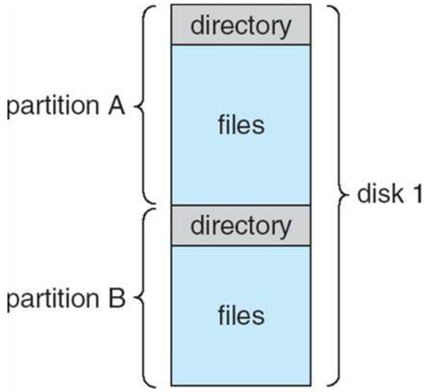


Linux



Windows

File System Structure



- **File System**: data structure on disk
- A physical disk can be **partitioned** into separate partitions
- Each partition can have a file system, swap space, and so on.
- Windows
  - Each volume is mounted in separate name space as “C:” or “D:”.

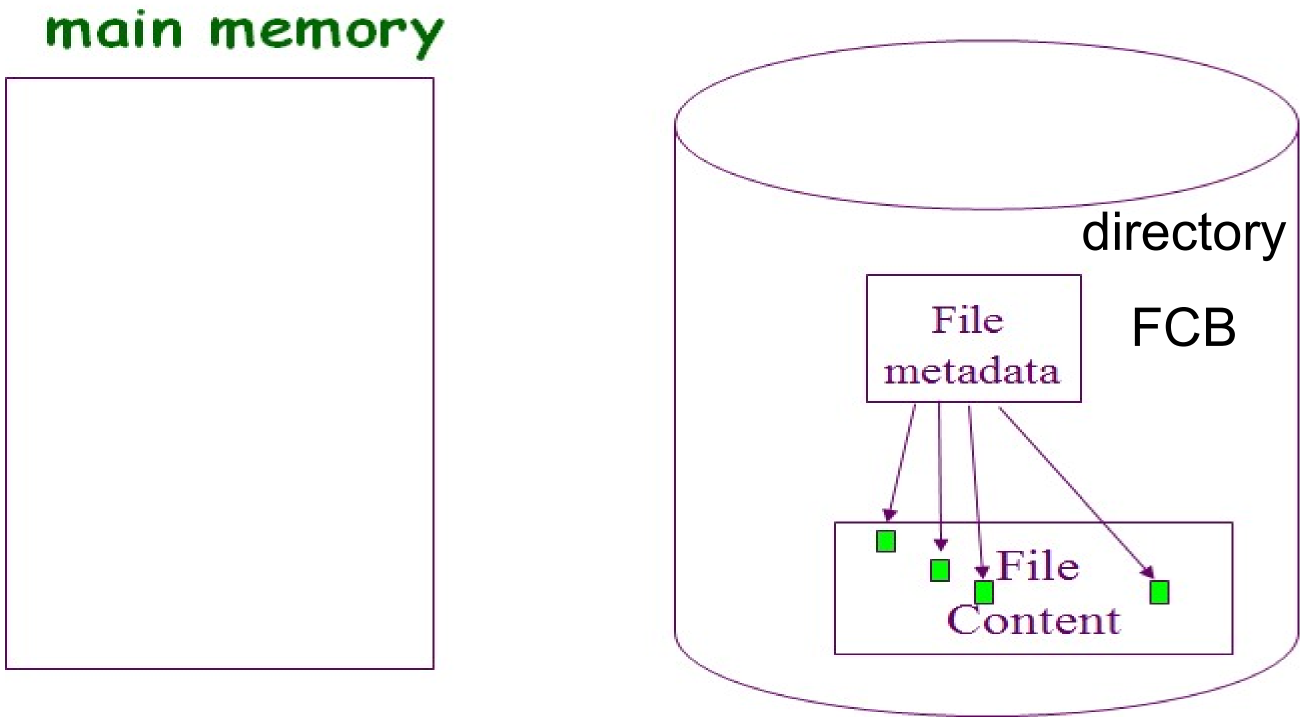
File Operations (Ch. 10.1.2)

- **Create/Delete, Open/Close**
  - Allocate disk space and update file system
  - e.g. fopen(), fclose()…
- **Read/Write**
  - e.g., fread(), fwrite(), …
- More details later

File Systems

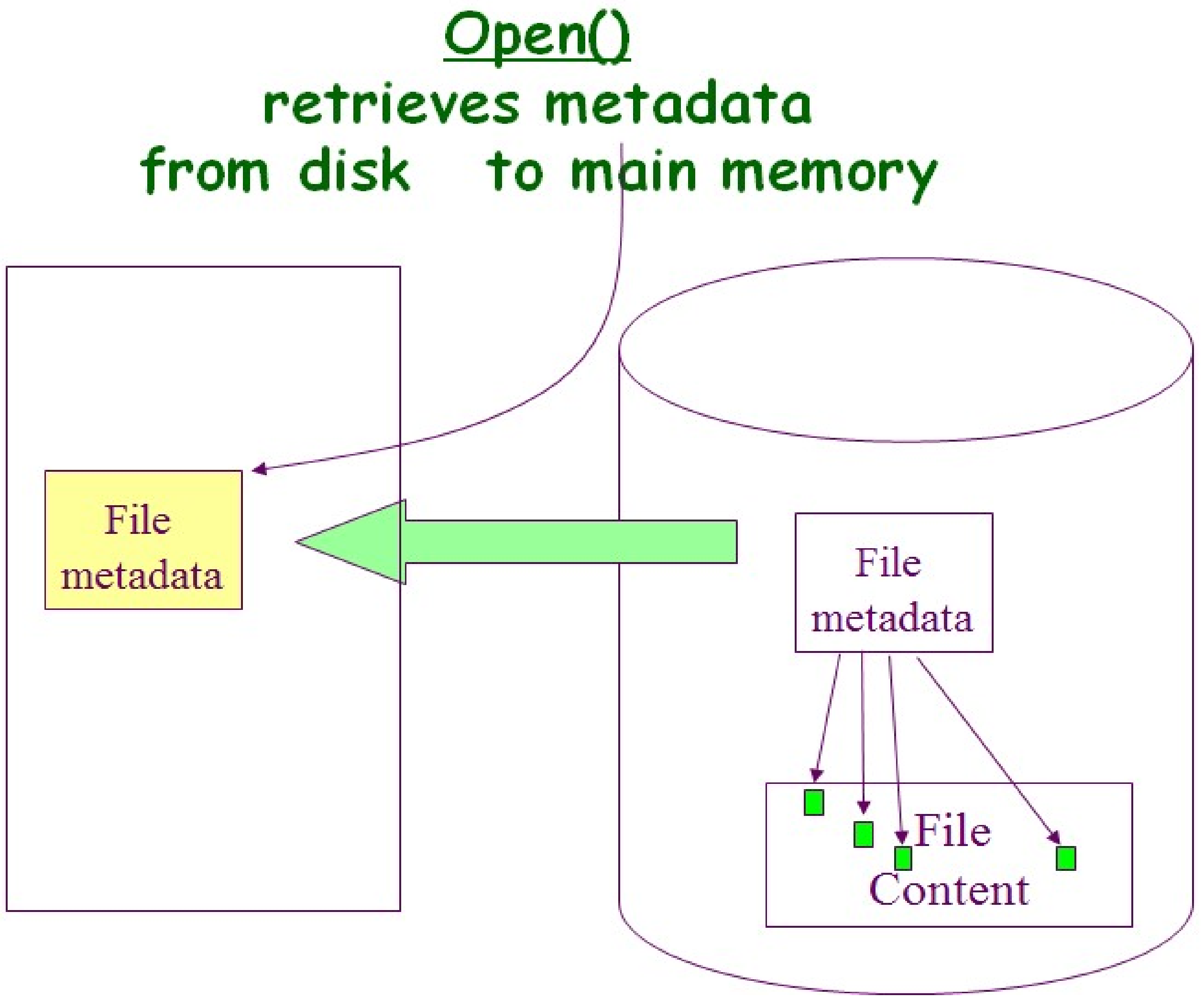
- File and File System Concept
- **File Operations**
- File Access and Allocation Methods
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
- Free-Space Management

File Operation Overview (1)

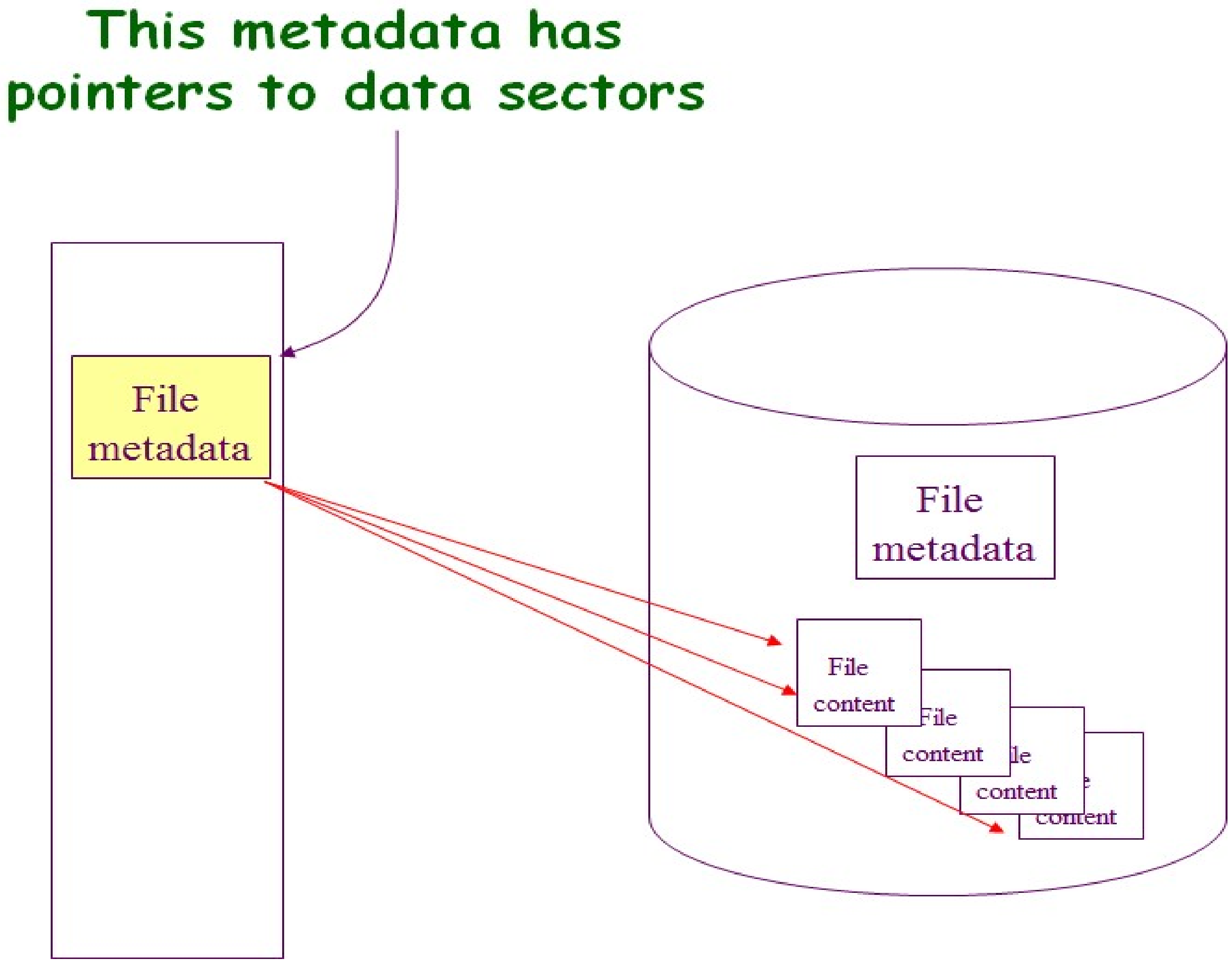


Example: Metadata : data explaining the data.
e.g.> Take Photo → **data**: photo image, **metadata**: resolution, file size, date/time, …

File Operation Overview (2)

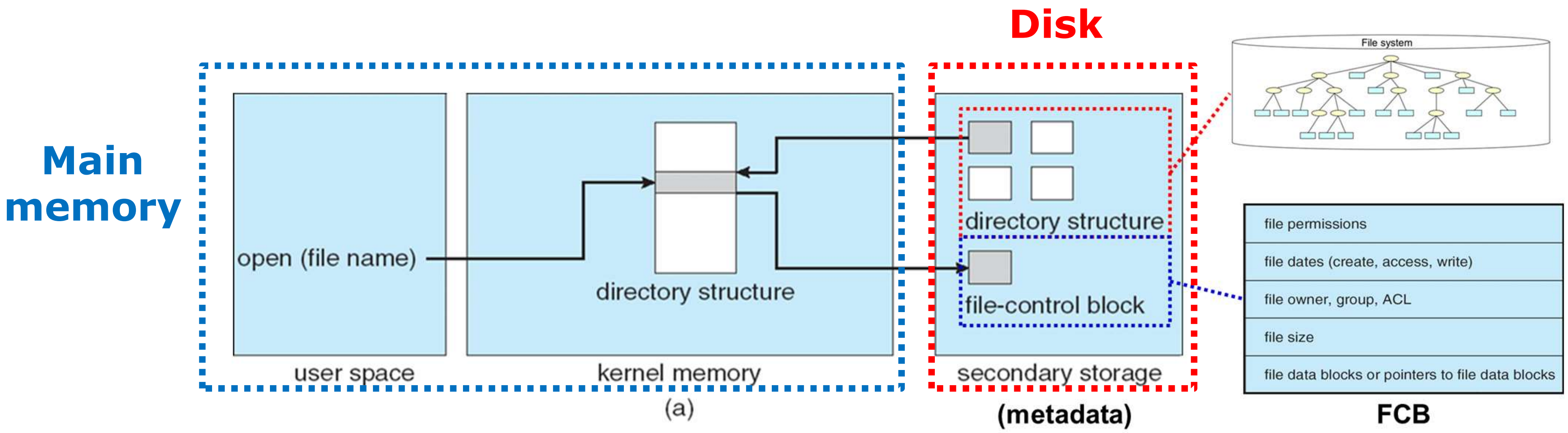


File Operation Overview (3)



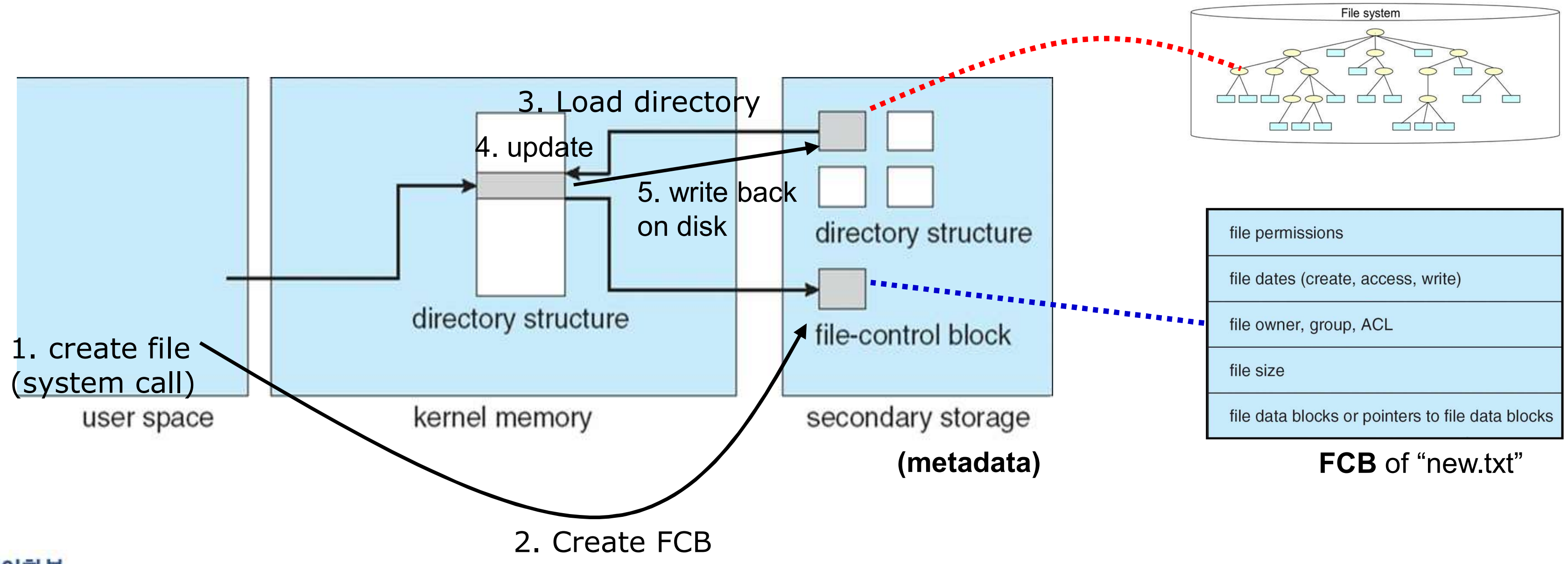
File System Implementation

- **In-memory structure**
  - For FS management and performance improvement via caching
  - In-memory **directory structure cache**
    - Directory information of *recently* accessed **directories**



File operations

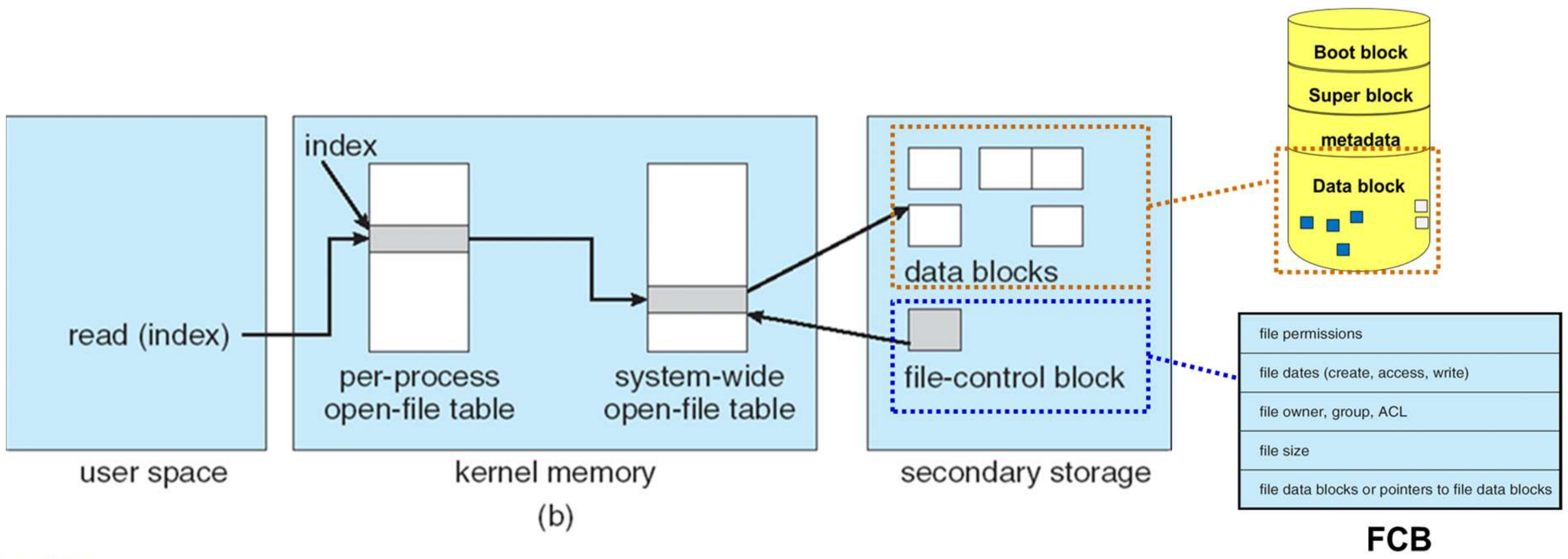
- **Create a file**
  1. Application calls file system (via system call) – e.g., fopen(“/usr/new.txt”)
  2. File system **creates a new FCB** – e.g., create a FCB (inode) for “/usr/new.txt”
  3. Search and Load appropriate **directory** into memory
     – e.g., add “/usr/” directory in in-memory directory structure cache
  4. **update directory** with new **file name** and **FCB information**
     – e.g., add new.txt and its FCB information (inode id) in “/usr/” directory
  5. write back on disk – e.g., write updated “/usr/” directory in directory structure (disk)



FCB of “new.txt”

File System Implementation

- **System-wide open-file table**
  - Information (FCBs) of all open files in <u>**system**</u>
  - **count**: number of processes that opened the file
- **Per-process open-file table**
  - Pointer to the appropriate entry in system-wide open-file table for <u>**each process**</u>



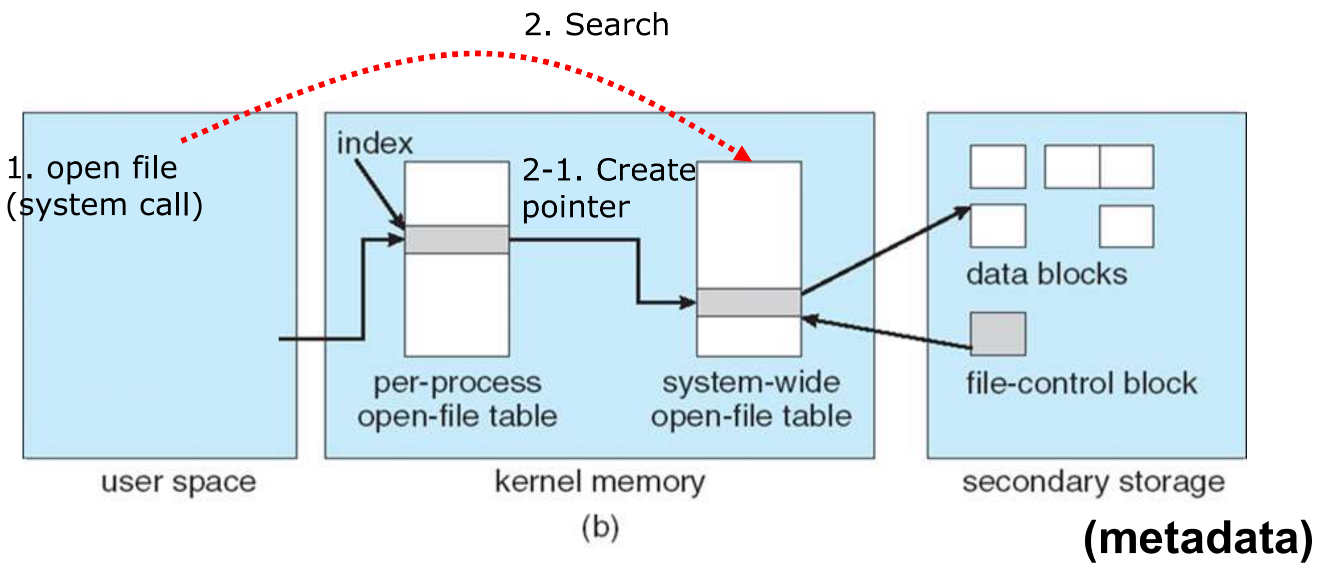
File operations

- **Open a file**

  1. **Open()** system call (e.g., **`fopen("file.txt")`** in C)

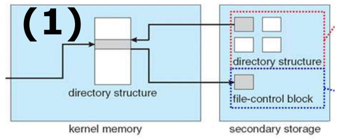
  2. First search **system-wide open-file table**

  2-1. If exists – some other process has already opened the file! Then, simply create a **per-process open-file table** entry that *points* to the *existing* **system-wide open-file table** entry

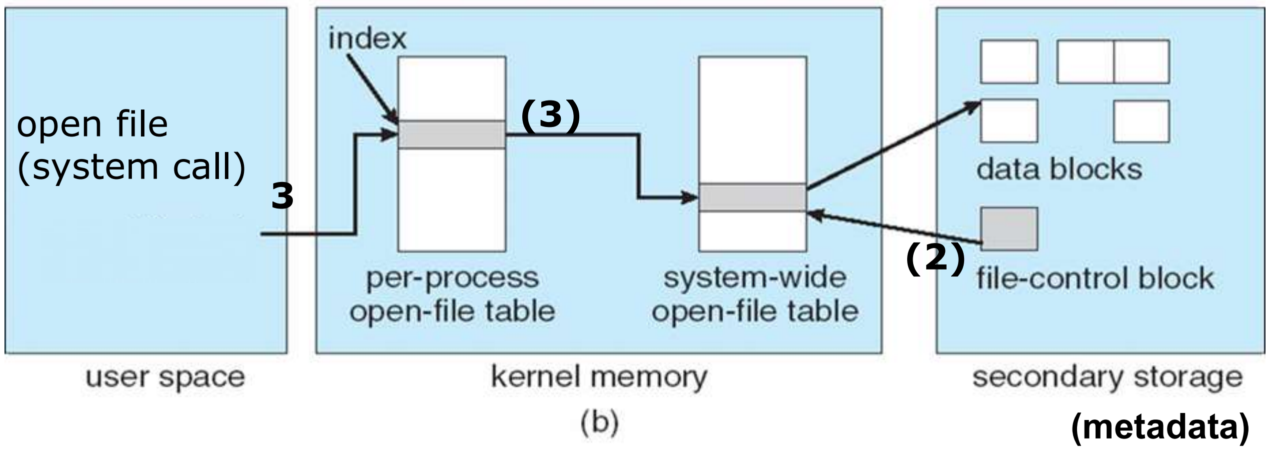


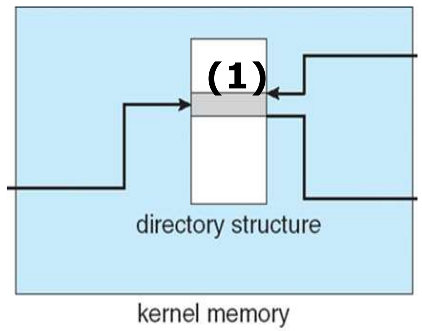
File operations

- Open a file (contd.)
  - 2-2. If not exists – search directory structure (in disk) for given file name
    - (1) Update in-memory **directory structure cache**
    - (2) Copy FCB in **system-wide open-file table** entry
    - (3) Create a **per-process open-file table** entry that <u>points</u> to the **system-wide open-file table** entry (same as 2-1)



  - 3. Open returns a **<u>pointer</u>** to **per-process open-file table** entry
    - – e.g., return value of *fopen()





2. Search

File operations

- Close a file (**Close()** system call)
  - Per-process open-file table entry is removed
  - System-wide open-file entry's open count is decremented
    - removed if count=0

File Systems

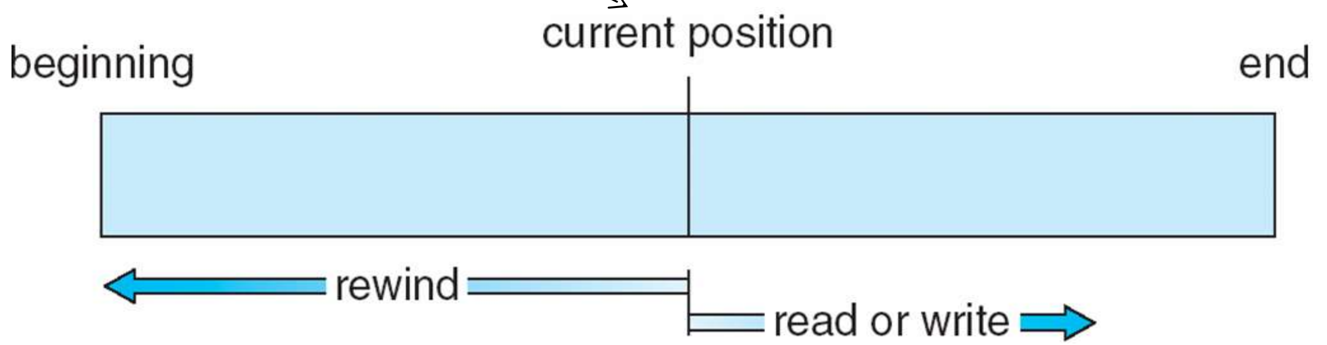
- File and File System Concept
- File Operations
- **File Access and Allocation Methods**
  - **Sequential and Direct Access**
  - **Contiguous, Linked, and Indexed Allocation**
- Free-Space Management

File Access Methods

- Sequential Access
  - Access file from start in sequence
- Direct Access
  - Directly access any part of file

Sequential access

- **Sequential access**
  - Information in the file is accessed/processed in order, one record (byte) after the other
  - File pointer
    - Tracks the current I/O location
  - **Most common**



Direct access

- **Direct access** (relative access, random access)
  - Allow programs to read/write records randomly, in no particular order
    - e.g., database systems
  - Method: Use **relative (logical) block number**
    - Relative block number: Index relative to the beginning of file
      - e.g., first relative block of file is 0, next is 1, …
    - read(n): n is the relative block number
      - read(3): read relative block number 3

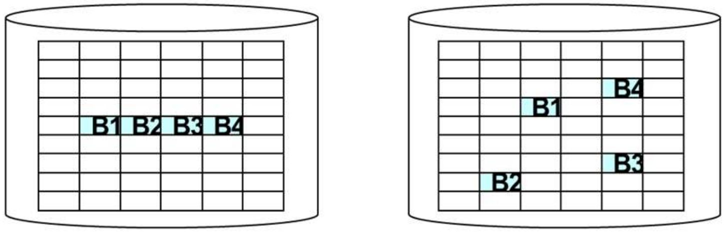
foo.c

block 0	block 1	block 2	block 3

relative block number

Files

- The data stored in a file must be persistent
  - Files are stored as disk blocks



(a) Contiguous (b) Non-contiguous

◦ Memory ⟷ Disk ;
block

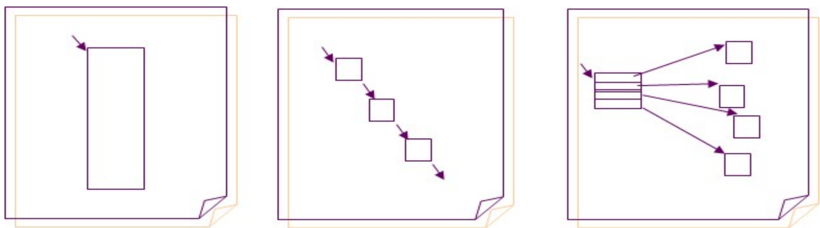
– Each block : 1 ~ many sectors (≈ 512bytes ~ 4kB, 8kB, ...)

- Allocation Method
  - How to allocate disk spaces (i.e., disk blocks) for files

Disk Allocation Methods

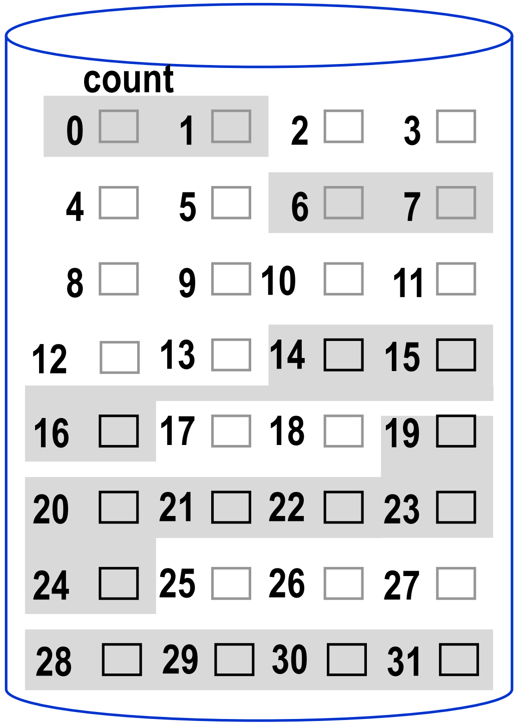
- Allocation Goal
  - To effectively utilize the disk space
  - To access files quickly

- 3 Major methods for allocating disk space
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation



Contiguous Allocation

- **Contiguous allocation**
  - Each file occupies a set of **contiguous blocks** on the disk
  - Simple : only (i) **starting location** (block #) and
    (ii) **length** (number of blocks) are required



directory

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

- Mapping from logical to physical
  - Block to be accessed R = Q + starting address

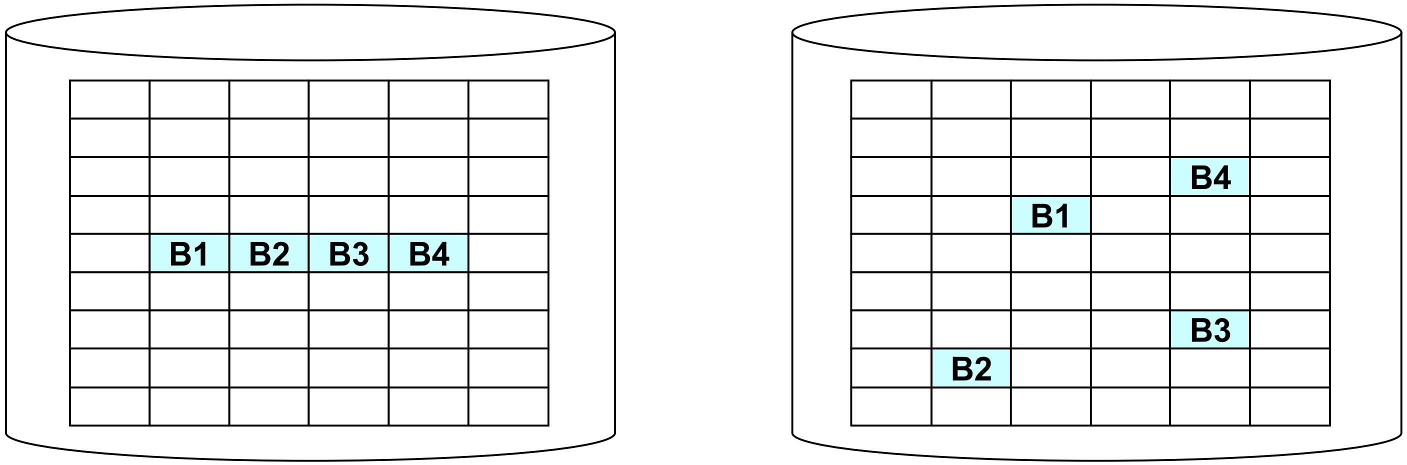
Contiguous allocation of disk space

Contiguous Allocation

- **Pros** : Easily supports **direct access** (random access)
  - Know any position (from its starting address)
- **Cons:**
  - Finding space for a new file
    - Done by free space management system
  - **External fragmentation** (Where did we see this before?)
    - The largest contiguous chunk is insufficient for a request
    - → need periodically **compaction** (compact all free space into one contiguous space)
      - memory vs. disk compaction?
  - Handling file size extension
    - Files may grow
    - Determining (predict) how much space is needed for a file

Contiguous vs. Non-contiguous Allocation

Contiguous vs. non-contiguous allocation

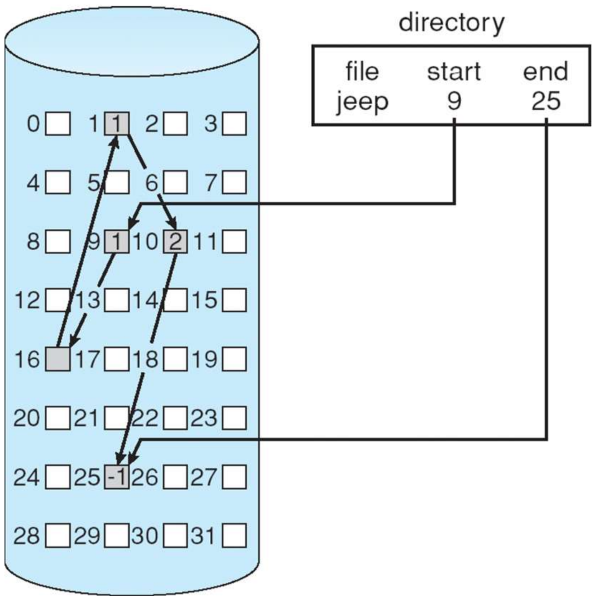


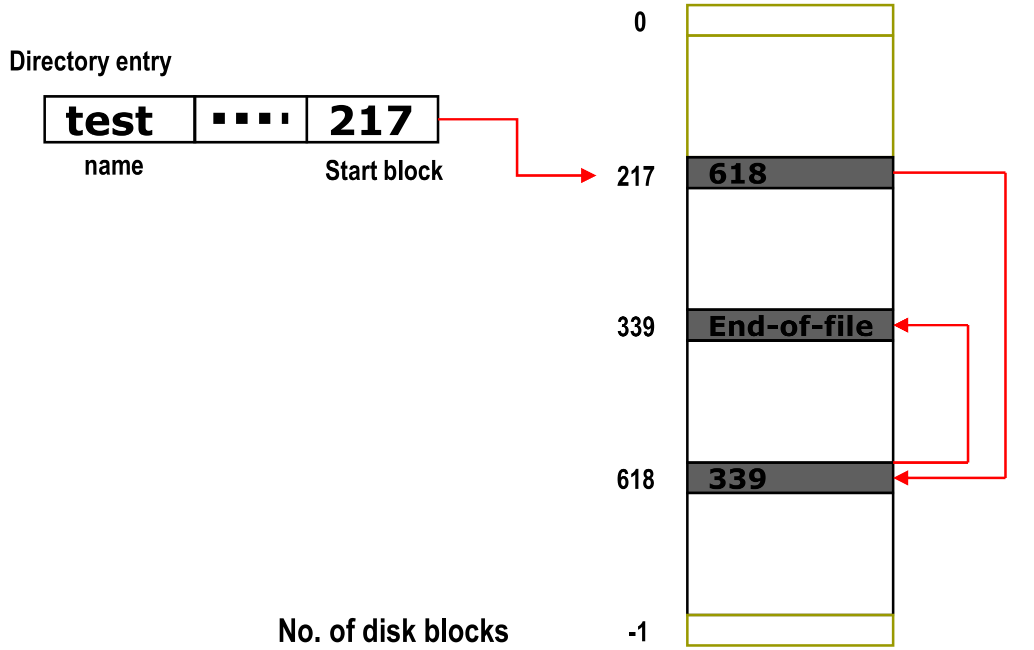
(a) Contiguous (b) Non-contiguous

File F consists of 4 blocks, B1, B2, B3, B4

Linked Allocation

- **Linked allocation** (non-contiguous)
  - Each file is a **linked list** of disk blocks
    - Blocks have the link, i.e., next block, information
    - Blocks may be **<u>scattered</u> anywhere** on the disk
  - The directory contains **pointers** to the first and/or last blocks of the file



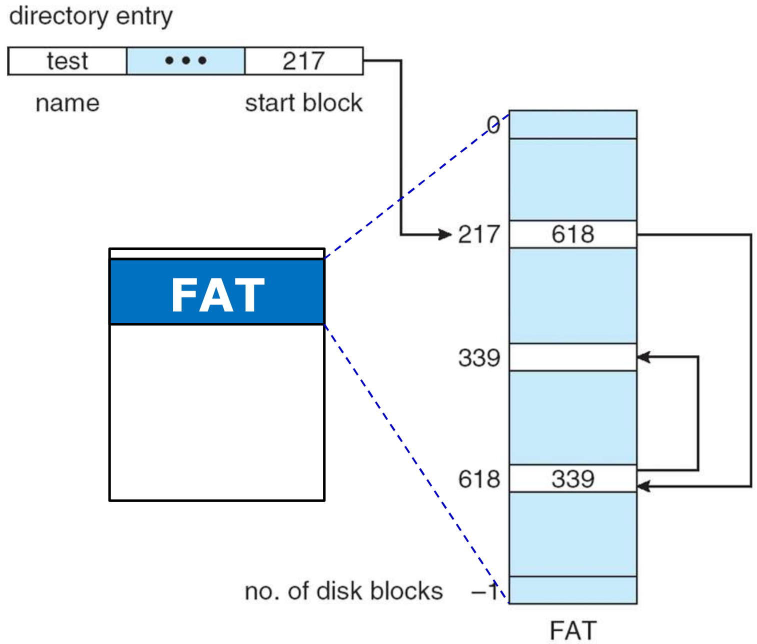


Linked Allocation

- **Pros**
  - No external fragmentation
  - Efficient for sequential access
- **Disadvantages**
  - **Inefficient** to support **direct-access** capability
    - Must always start at beginning of file and follow the links
    - A significant number of disk head seeks
  - **Extra space** required for the **pointers**
    - Solution: Increase block size! – increases **internal fragmentation**
  - Reliability
    - A link failure at beginning → whole file can not be used

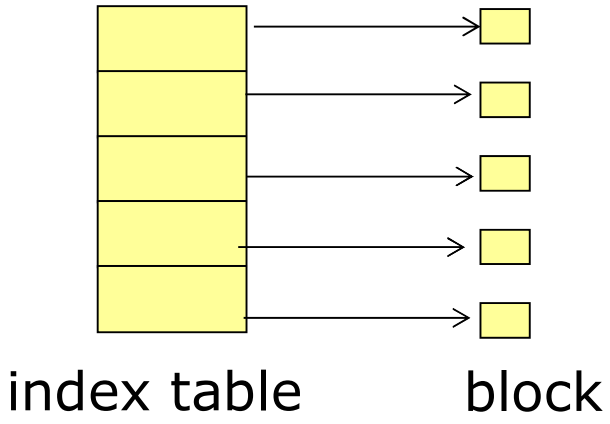
Linked Allocation - FAT

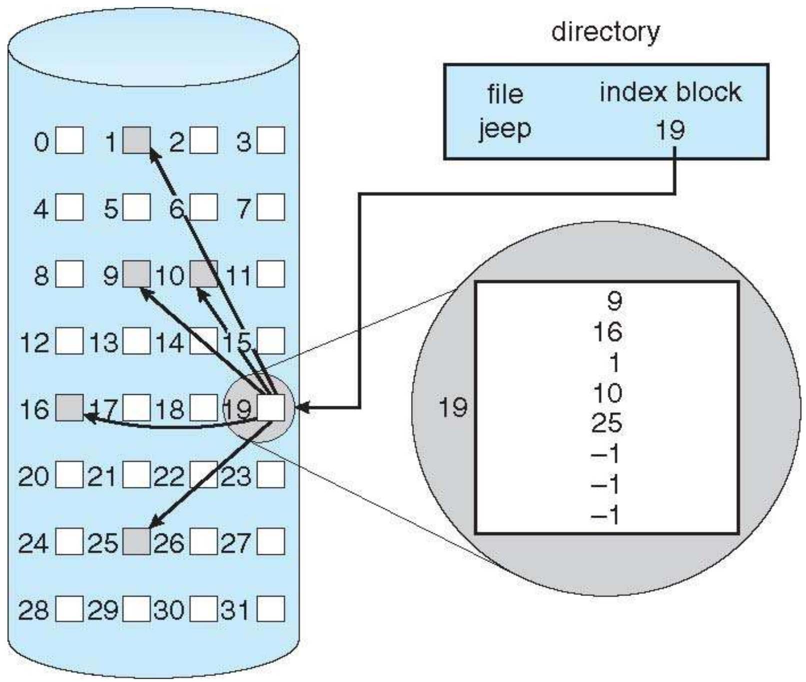
- Linked allocation: variation → FAT
- **File-Allocation Table** (**FAT**)
  - **A section of disk at the beginning of each partition** is set to contain the FAT
    - One entry for each disk block
    - Indexed by block number
  - FAT can be stored cached in memory
    - Random access time is improved!
  - e.g., MS DOS, Windows 95/98



Indexed Allocation - inode

- Indexed allocation
  - Bring **all the pointers** together into one location, the **index block**
    - Each file has its own **index block**
  - **Index block**
    - Array of disk block addresses





  - Unix, Linux File System

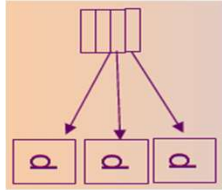
Indexed Allocation

- Characteristics
  - **Pros.**
    - No external fragmentation
    - Supports direct access (random access)
  - **Disadvantages**
    - Need Index table → Suffer from wasted space
    - Determining the size of **index block**
      - Optimal size varies depending on the file size

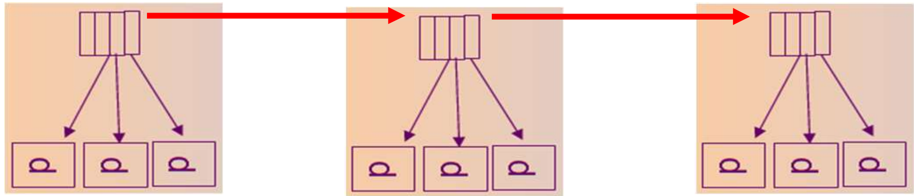
- Mechanism to deal with **index block size**
  - Linked scheme
  - Multilevel index
  - Combined scheme

Indexed Allocation – Mapping (Cont.)

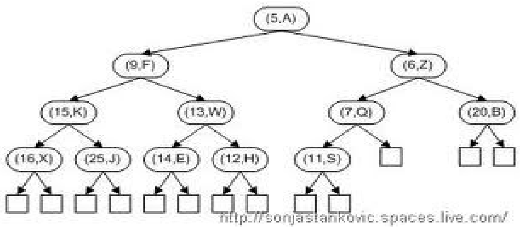
- What if file is <u>too large?</u> → one disk block is too small for index table

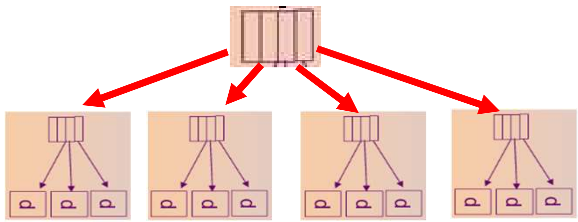


1. **Linked scheme** – Link blocks of index table (no limit on size).



2. **Two-level index**

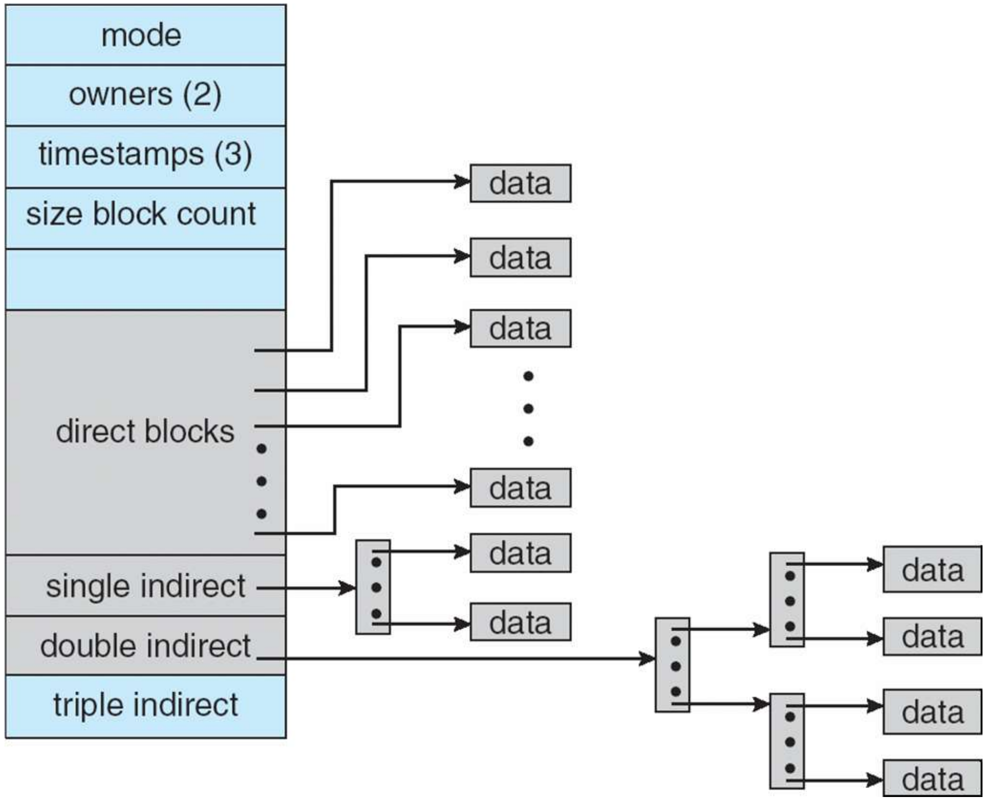




3. **Combined scheme** (e.g., UNIX/Linux inode)
   - **Direct blocks** (for small files) and **Indirect blocks** (for large files)

Indexed Allocation – UNIX inode

- **Combined Scheme**: UNIX (4K bytes per block)



Access Performance for Allocation Methods

- Performance Criteria
  - Data-block access time
  - Storage efficiency
- Performance
  - Contiguous allocation
    - Requires only one access to get a disk block
      - Initial address of file in memory, can calculate immediately the disk address of i^{th} block
    - Good for both **sequential/direct access**
    - Storage efficiency not good – external fragmentation, file size prediction, ...
  - Linked allocation
    - Fine for sequential access but not for **direct access**
      - May need i disk reads to read i^{th} block

Access Performance for Allocation Methods

- Indexed allocation
  - If index is in memory, supports both sequential access and direct access
  - If index is not in memory, single-level index needs two disk reads: read index block then data block, Two-level needs three, …
- Use contiguous allocation for small files
  - If the file grows → switch to indexed allocation

File Systems

- File and File System Concept
- File Operations
- File Access and Allocation Methods
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
- **Free-Space Management**

Free-Space Management

- To keep track of free disk space, the system maintains a **free-space list**
  - Records all *free disk* blocks

- **Bit vector** (bit map)
  - Easy to get contiguous files
- **Linked list** (free list)
  - Cannot get contiguous space easily
  - No waste of space

Bit Vector (Bit Map)

- **Bit vector** (*n* blocks)
  - Free space list is implemented as a bit map or bit vector

block#	0	1	2				n-1
	0	**0**	**1**	**1**	**1**	**1**	…

$$\text{bit}[i] = \begin{cases} \mathbf{0} \Rightarrow \text{block}[i] \text{ free} \\ \mathbf{1} \Rightarrow \text{block}[i] \text{ occupied (allocated)} \end{cases}$$

- Free space list
  - ✓ Free space bit map

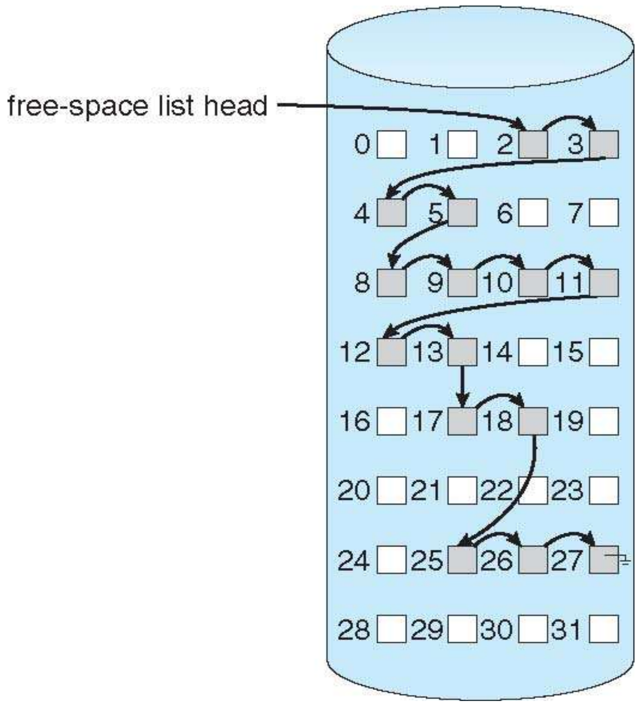
    Ex) 0011110011111100011000000011100000……

Bit Vector (Bit Map)

- Bit map requires extra space
  - Bit map : Must be kept on disk for large disks
    - 1-TB disk with 4-KB blocks require 256MB to store its bitmap
- Advantage:
  - Simple and efficient to find contiguous free blocks

Linked List

- **Linked List**
  - Linked Free-space List
    - The 1st block contains a pointer to the next free disk block
  - Traversing this list is slow (disk I/O)
    - Need to follow the links
    - But traversing is not needed frequently
  - No waste of space
    - The link is stored in the free blocks anyway...
  - e.g., FAT
  - Hard to find contiguous free blocks
    - bad for disk access time



- Appendix
  - Efficiency and Performance
  - File Protection

Efficiency and Performance

- Disks tend to represent major bottleneck in system performance – slowest main computer component
- Need to improve the efficiency and performance

- **Improving Performance**
  - **Buffer cache / Page cache** – separate section of main memory for frequently used blocks
  - **Synchronous writes** and **Asynchronous writes** – Stable or faster writes
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access

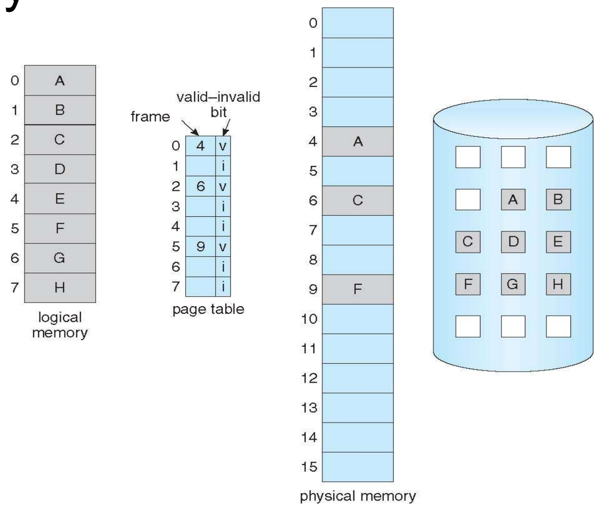
Buffer Cache and Page Cache

- **Buffer cache**
  - A separate section of main memory
  - Recently accessed disk blocks are kept

- **Page cache**
  - A separate section of main memory
  - caches pages rather than disk blocks using virtual memory techniques and addresses
  - Memory-mapped files and I/O uses a page cache

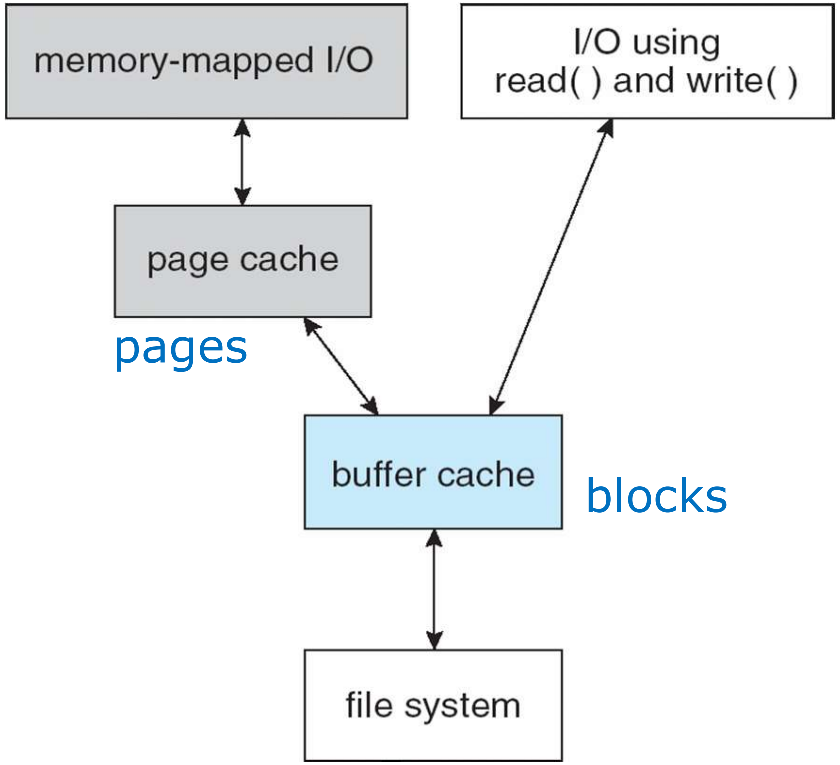
Memory-Mapped I/O (Ch. 9.7)

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using ***demand paging***
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()`, which require expensive system calls
- In most operating systems, the memory region mapped actually is the kernel's page cache, meaning that no copies need to be created in user space.



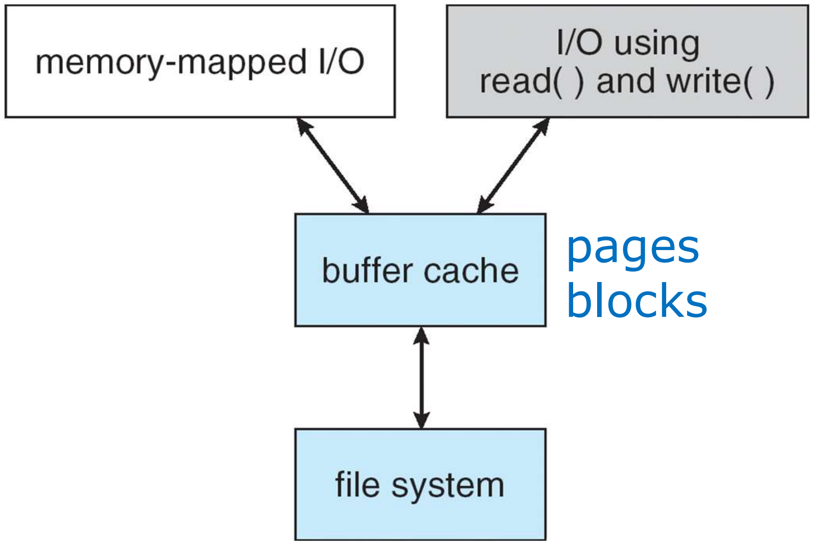
I/O Without a Unified Buffer Cache

- I/O Without a Unified Buffer Cache
  - Routine I/O through the file system uses the buffer cache
- Problem: **Double caching**
  - Contents of buffer cache must be copied to page cache
  - Double caching wastes memory, CPU and I/O cycles due to extra movement



I/O Using Unified Buffer Cache

- I/O Using 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 double caching
  - Virtual memory system manages caching of file-system data
  - Used in versions of UNIX/Linux, Solaris, and Windows



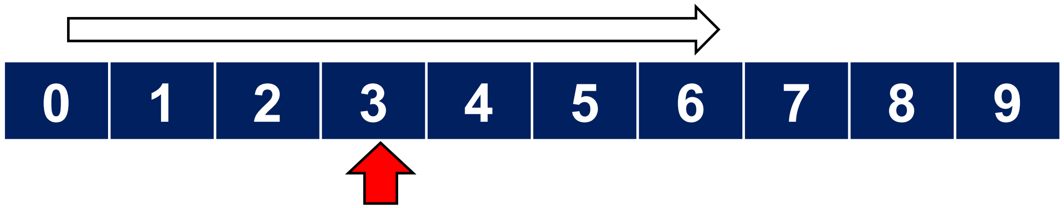
Synchronous / Asynchronous Writes

- **Synchronous writes**
  - Writes are not buffered/cached – writes must reach disk before process can proceed
  - Sometimes requested by apps or needed by OS
  - e.g., database systems for atomic transactions – assure that data reach stable storage in the required order

- **Asynchronous writes**
  - Data is stored in cache, control returns to caller
  - Most common, buffer-able, faster

Optimizing Page Cache for Sequential Access

- **Page Replacement Algorithms** in **sequential access** of files
  - A file being read sequentially should not have pages replaced using **LRU**
  - Usually most recently used page will be used last, or perhaps never again



- **Free-behind**
  - Remove a page from the buffer as soon as the next page is requested - previous pages are not likely to be used again

- **Read-ahead**
  - Requested page and several subsequent pages are read and cached - these pages are likely to be requested after the current page

More Topics on File Systems

- Virtual file system
  - Support for multiple types of file systems

- Network file system
  - e.g., SUN NFS

- Backup and Recovery
  - Consistency Checking
  - Log-structured file system

File Protection

- **File protection**
  - Protection mechanisms provide controlled access by limiting the types of file access that can be made
  - Protection mechanism for multi-user system is different from that of single-user system

Controlled Access

- **Controlled access** - File owner/creator should be able to control:
  - **what** can be done
  - by **whom**

- Types of operations that should be controlled
  - **Read** - read from the file
  - **Write** - write or rewrite the file
  - **Execute** - load the file into memory and execute it
  - **Append** – write new information at the end of the file
  - **Delete** - delete the file and free its space for possible reuse
  - **List** – list the name and attributes of the file

Access Control

- Make access dependent on the identity of the user
  - Example
    - Sara should be able to invoke **all** operations on the file
    - Jim, Dawn, and Kill should be able only to **read** and **write** the file; they should n ot be allowed to delete the file
    - All others should be able to **read**, but not **write** the file.
- **Mode of access**: **read**, **write**, **execute** (**rwx**)
- Three classes of users
  - **Owner** : the user who created the file
  - **Group**: a set of users who are sharing the file and need similar access
  - **Universe** (public) : All other users in the system

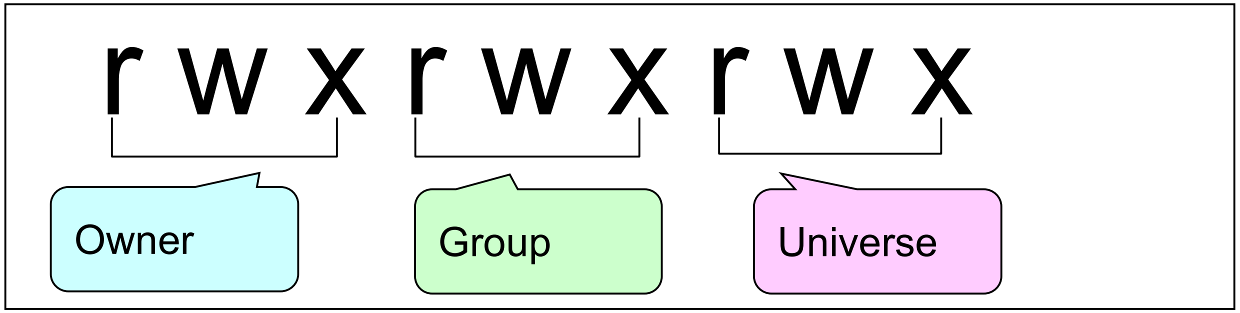
UNIX/Linux File Protection

- **Access list**
  - Specify user names and the types of access allowed for each user
  - Implementation
    - Implement each column in access matrix as an access list
    - Access list consists of ordered pairs
      - <domain, rights-set>

Access list for object F_k

$A_{list}(F_k) = \{ <D_1, R_1>, <D_2, R_2>, \ldots, <D_m, R_m> \}$

✓ Used in Unix file access permission management



A Sample UNIX Directory Listing

- Permission in a Unix/Linux system

-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/

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
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
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.

owner **group** **public**

chmod 761 game

Attach a group to a file

```
chgrp    G    game
```

Windows 7 Access-Control List Management