UNIT-II FILE SYSTEM STRUCTURE

Q-1: Elaborate Buffer cache with its Structure of buffer pool and describe the Scenarios for buffer retrieval with example

Buffer Cache:

The **buffer cache** is a portion of main memory that holds disk blocks temporarily while they are being read from or written to disk.
It improves system performance by reducing the number of slow disk I/O operations.
The buffer cache is managed by the **kernel** and is shared by all processes.

Buffer Header:

A **buffer header** is a metadata structure maintained by the operating system to manage a buffer (which contains a copy of a disk block in memory).

- The buffer header does **not** hold the actual file data.
- Instead, it stores information about the buffer such as:
    - Which disk block it corresponds to.
    - Its current status (free, busy, dirty, delayed write).
    - Links to hash queues and free lists.

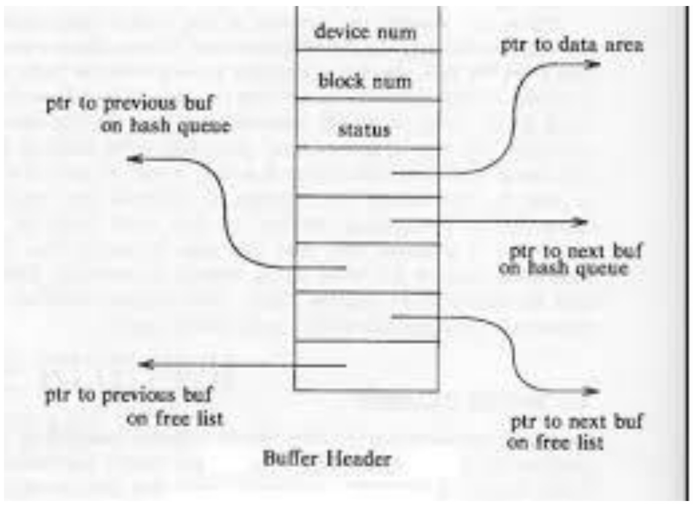
Example Scenario

Suppose a process requests **Block 28 from Disk D1**.

- The kernel searches the **hash queue** using the buffer header (device ID + block number).
- If the buffer header for **(D1, Block 28)** exists:
    - The buffer header points to the data block in memory.
    - The kernel returns it immediately (Cache Hit).

If it does not exist:

- A free buffer header is taken from the free list.
- Updated with:
    - Device = D1
    - Block = 28
    - Status = Busy (while being read)
- Then linked into the proper **hash queue** for future lookup.
- After disk read completes, process gets the block.



device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list
Buffer Header

Structure of a Buffer Pool:

The **Buffer Pool** in UNIX (or any OS using buffer cache) is the collection of **buffer headers + data blocks** in memory. It ensures efficient management of disk blocks by caching them in RAM.

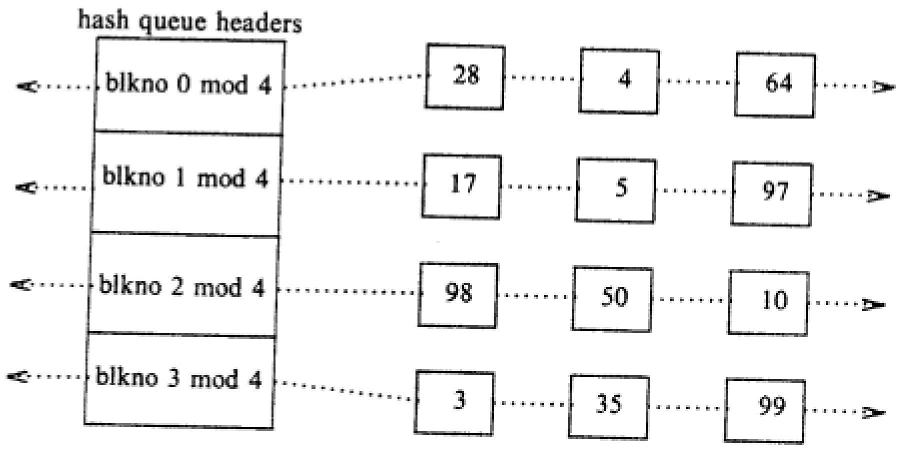
Main Components of Buffer Pool

1. **Buffer Headers**
   - Metadata about each buffer.
   - Contains block number, device ID, status (busy, free, dirty), and pointers.
   - Linked to **hash queues** and **free list**.
2. **Data Blocks (Buffer Contents)**
   - Actual space in memory where disk blocks are copied.
   - Each buffer header points to one data block.
3. **Hash Queues**
   - Buffers are hashed based on (device, block number).
   - Used for **fast lookup**: quickly check if a block is already in memory.
4. **Free List**
   - Buffers not currently in use are linked here.
   - If a new block needs to be loaded and isn't in cache, a buffer is taken from the free list.

Example: Process requests Block 28 from Disk D1 [28%4=0→blkno 0]

1. **Hash Queue Lookup**
   - The kernel checks the hash queue using (D1, 28).
   - If found → buffer header points to data block → **cache hit** (fast).
2. **Cache Miss (Not Found in Hash Queue)**
   - Kernel looks for a free buffer in the free list.
   - Suppose buffer B is free → header updated with (Device = D1, Block = 28).

3. **Read from Disk**
    - Block 28 is read into buffer B's data block.
    - Buffer header added to hash queue for (D1, 28).
4. **Process Gets the Data**
    - Now the process can read from buffer pool (not directly from disk).
    - If another process asks for the same block → retrieved from hash queue instantly.



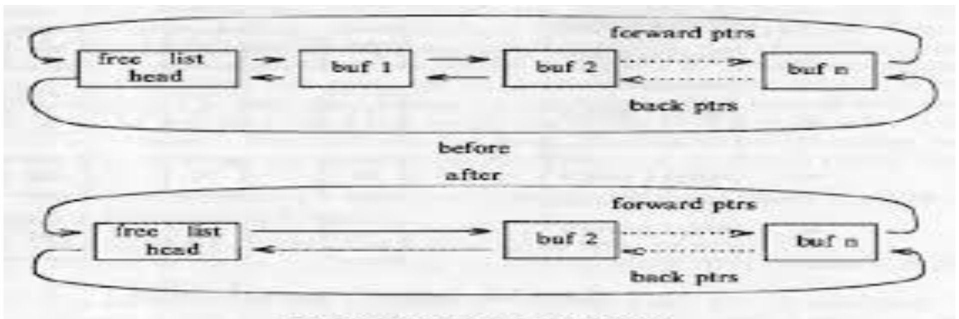
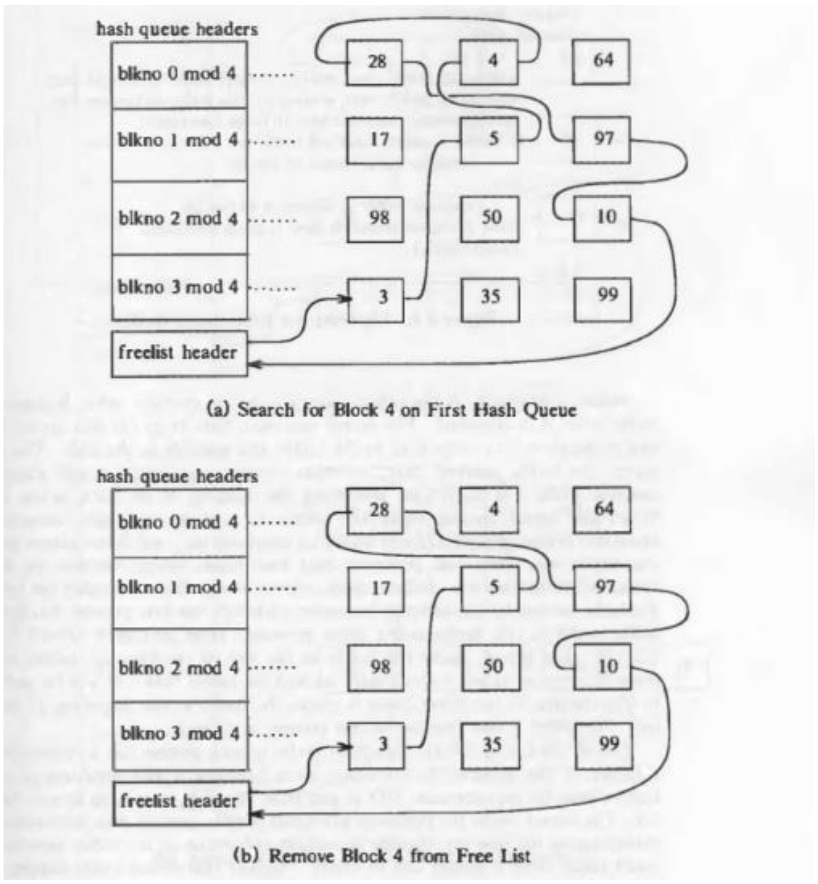


Figure 3.2. Free List of Buffers

Scenarios for Buffer Retrieval:

When a process requests a disk block, the kernel must retrieve it from the buffer cache.



Case 1 – Block Found in Hash Queue and Free

(Cache Hit & Free → Immediate Allocation)

- Block is already present in cache (hash queue) **and** in free list (not busy).
- Kernel removes it from free list and allocates it immediately.
- Fastest case → **Cache Hit**.

Example:

- Process requests **Block 4**.
- Hash Queue 0 contains: 28 → 4 → 64.
- Block **4 is found** and also free.
- Kernel removes it from free list and returns to process **without disk I/O**.

Case 2 – Block Found in Hash Queue but Busy

(Cache Hit but Busy → Wait)

- Block is already present in cache, but some other process is **using it (busy)**.
- The requesting process must wait (sleep) until the buffer becomes free.

Example:

- Suppose Process A is writing to **Block 64**.
- Now Process B also requests Block 64.
- Since Block 64 is in hash queue but **busy**, Process B must **wait** until Process A releases it.

Case 3 – Block Not in Hash Queue, Free Buffer Available
(Cache Miss, Free Buffer → Read from Disk)

- Requested block is not in hash queue (not cached). But there is a **free buffer** available.
- Kernel takes a buffer from the free list, assigns it to the requested block, reads block from disk, and inserts it into correct hash queue.

Example:

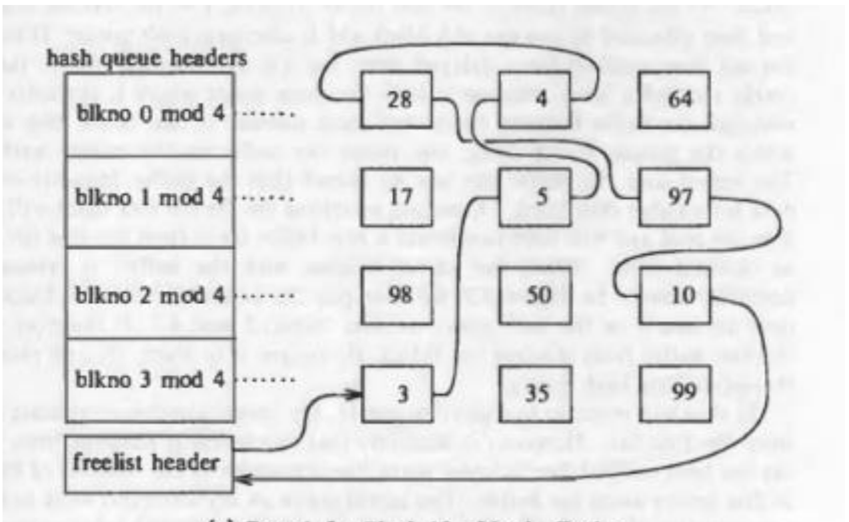
- Process requests **Block 45** (not in any hash queue).
- Free list has blocks available (e.g., block 35).
- Kernel removes block 35 from free list → reads **Block 45 from disk** → places it into hash queue → returns it.

Case 4 – Block Not in Hash Queue, No Free Buffer Available
(Cache Miss, No Free Buffer → Wait for Free Buffer + Read)

- Requested block is not in cache AND free list is empty (all buffers busy).
- Kernel must wait until some process releases a buffer into free list. Then, it will allocate that buffer for requested block.

Example:

- Process requests **Block 50**.
- Suppose all buffers are busy (free list empty).
- Kernel makes the process wait → when a buffer (say Block 97) is released → Kernel reuses it → reads Block 50 from disk → returns it.



(a) Search for Block 18 - Not in Cache

Case 1 – Block Found in Hash Queue and Free

- The requested block is already present in the hash queue.
- It is **not marked busy**, so the kernel can allocate it immediately.
- **Fastest case** → Cache Hit, no disk I/O.

Example: Process requests block **5**.

- Block 5 is in hash queue (5 mod 4 = 1) and is free.
- Kernel returns it directly from cache.

Case 2 – Block Found in Hash Queue but Busy

- Block is in cache but marked **busy** (another process is using it).
- The requesting process must **wait (sleep)** until the block is released.

Example: Process A is using block **50**.

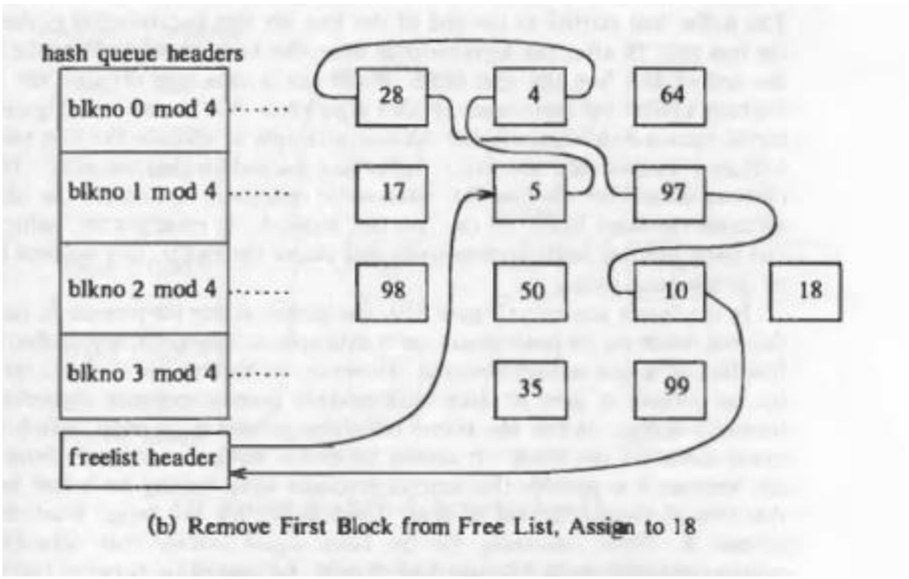
- Process B also requests block 50.
- Since block 50 is busy, Process B waits until Process A releases it.

Case 3 – Block Not in Hash Queue, Free Buffer Available

- Block is **not in cache** (not present in hash queue).
- Kernel checks the **free list** for an available buffer.
- If a free buffer exists, it is removed from the free list, assigned to block 18, and data is read from disk.
- After reading, block 18 is inserted into its corresponding hash queue (18 mod 4 = 2).

Example (Diagram): Process requests **block 18**.

- Block 18 not in hash queue → MISS.
- Kernel takes a free buffer (say block 3 from free list).
- Reads block 18 from disk into that buffer.
- Inserts block 18 into hash queue (18 mod 4 = 2).
- Remove the first free block, means least recently used (block 3) from free list.



(b) Remove First Block from Free List, Assign to 18

Q-2: Elaborate Allocation of disk blocks and classify Reading and writing disk blocks with example

Allocation of Disk Blocks:

- Disk space is divided into **blocks** (fixed size, e.g., 512 bytes or 4 KB).
- When a file is created or extended, the operating system must **allocate disk blocks** to store its contents.
- Efficient allocation is important for **performance, fast access, and minimal fragmentation**.

Techniques for Allocation of Disk Blocks

1. **Contiguous Allocation**
   - Each file occupies a set of **consecutive blocks**.
   - Directory stores the **starting block number** and **length**.
   - **Advantages:** Fast sequential and direct access.
   - **Disadvantages:** External fragmentation, difficult to extend file.
   - **Example:** File A needs 4 blocks → allocated blocks 100–103 consecutively.

2. **Linked Allocation**
    - Each file is a **linked list of blocks** scattered on disk.
    - Each block contains **data + pointer to next block**.
    - Directory stores the **starting block**.
    - **Advantages:** No external fragmentation, easy to extend.
    - **Disadvantages:** Slow direct access, pointer overhead.
    - **Example:** File B uses blocks 10 → 55 → 37 → 92 (linked by pointers).

3. **Indexed Allocation**
    - Each file has an **index block** that contains all pointers to its data blocks.
    - Directory entry stores the **index block number**.
    - **Advantages:** Supports direct access, no external fragmentation.
    - **Disadvantages:** Extra space needed for index block.
    - **Example:** File C's index block contains [12, 25, 40, 68] → data is stored in these blocks.

(add Allocation of Disk Blocks Algorithm here)

Reading and Writing Disk Blocks:

Disk block operations are classified into **three main categories**:

1. Reading a Disk Block

When a process requests a block:

1. **Check Buffer Cache**
    - If block already present (**cache hit**) → copy from buffer to process.
    - If not present (**cache miss**) → allocate buffer → read block from disk → store in cache → give to process.

   Example:

    - Process requests block 50.
    - Not in cache → OS allocates free buffer → reads block 50 from disk → stores in cache →Block 50 added to hash queue (50 mod 4 = 2) → delivers data. [Next time **block 50** is requested, it will be found directly in cache (fast, no disk read).]

(add Reading a Disk Block Algorithm here)

2. Writing a Disk Block

When a process wants to update (write) a disk block, the kernel manages it using **buffer cache**. There are **two approaches**:

(a) Synchronous Write

1. Process modifies data in buffer.
2. Kernel **immediately writes** buffer contents to disk.
3. Once disk I/O completes, the process continues.

Advantages:

- Guaranteed **data consistency** (safe even if system crashes right after write).
- Used for **critical data** like metadata, superblocks, inodes.

Disadvantages:

- **Slower** because every write must wait for disk I/O.

Example:

- Updating the **inode table** when a file is created.
- Kernel writes both:
    - to buffer cache
    - and **instantly flushes** it to disk.

(b) Asynchronous (Delayed) Write

1. Process modifies data in buffer cache.
2. Kernel **marks buffer as "dirty"** but does not immediately write to disk.
3. Disk write happens later, when:
    - Buffer is reused, or
    - A periodic **sync daemon** flushes dirty buffers.

Advantages:

- **Faster** since process does not wait for disk I/O.
- Multiple writes can be **batched together**, reducing disk head movement.

Disadvantages:

- Risk of **data loss** if system crashes before buffers are written to disk.

Example:

- Writing to a text file:
    - Data goes into buffer cache first.
    - Actual disk update happens later during sync.

(add Writing to a Disk Block Algorithm here)

Advantages of Buffer Cache

1. **Speed (Reduced Disk I/O)**
   - Accessing data from main memory (RAM) is much faster than from disk. When a process requests a block, the OS first checks the buffer cache. If present, it avoids a slow disk read.
   - Example: Reading the same file multiple times—only the first access hits the disk, later accesses are served from cache.
2. **Caching Frequently Used Blocks**
   - The buffer cache holds commonly accessed disk blocks. This increases the probability of a cache hit, reducing redundant disk operations.
   - Example: System configuration files or directory blocks are repeatedly accessed and thus remain cached.
3. **Read-Ahead (Prefetching)**
   - The OS anticipates sequential file access. It preloads the next few blocks into the buffer cache before the process asks. This reduces waiting time and improves throughput in sequential read operations.
   - Example: While reading a large video file, the OS pre-fetches the next data blocks.
4. **Write Optimization (Delayed Write / Write-Back)**
   - Instead of writing immediately to disk, modifications are first stored in buffer cache. Writes are grouped together and written later, reducing the number of disk I/O operations.
   - Example: Multiple updates to the same file block are written once instead of several times.
5. **Synchronization and Uniform Interface**
   - Buffer cache ensures a uniform way for all processes to access files. Synchronization mechanisms prevent conflicts when multiple processes access the same file.
   - Example: Two processes opening the same log file both get consistent data through buffer cache management.

Disadvantages of Buffer Cache

1. **High Memory Usage**
   - Buffer cache occupies a large portion of RAM. This reduces memory available for running applications and other processes.
   - Example: On small systems, a large buffer cache can starve user programs of memory.
2. **Stale Data (Cache-Disk Mismatch)**
   - If cache contents are not properly written back, data in memory may be outdated compared to disk. This can cause inconsistency issues.
   - Example: If one system writes directly to disk (e.g., via raw disk access), buffer cache may still hold old data.
3. **Crash Risk (Data Loss in Delayed Writes)**

    - With delayed writes, data is kept in RAM temporarily. If the system crashes before flushing, unsaved data is lost.
    - Example: A sudden power failure may cause recent file changes to vanish.
4. **Overhead of Management**
    - Maintaining **hash queues**, **free lists**, and synchronization increases system complexity. CPU time is also spent in managing cache structures rather than actual processing.
    - Example: Searching and updating buffer headers in hash queues adds overhead.

Q-3: Demonstrate Kernel architecture along with Kernel data structure give me example

Kernel Architecture:

- The **Kernel** is the core part of an operating system.
- It provides **low-level services** such as process management, memory management, file system, device handling, and inter-process communication.
- The kernel acts as a **bridge between hardware and user applications**.

1. User Space (Applications Layer)

- This is where **users and applications** run.
- Applications don’t directly access hardware → they make **system calls** to interact with kernel.

Example:

- A text editor like **Notepad** or **vi** runs in user space.
- When you save a file, it doesn’t write directly to the hard disk—it requests the **kernel** to do it.

2. System Calls (Interface between User & Kernel)

- System calls act as a **bridge** between user programs and kernel services.
- They provide a **controlled entry point** into the kernel.

Example:

- open(), read(), write(), fork(), exec() are common system calls.
- If a C program calls printf("Hi"), internally it uses write() system call to send data to output device.

3. Kernel (Core of OS)

The kernel runs in **privileged mode** and directly controls the hardware.
It is divided into major components:

(a) Process Management

- Creates, schedules, and terminates processes.
- Manages CPU time using scheduling algorithms.

Example:

- Two programs running (e.g., **MS Word + Browser**) → kernel uses **Round Robin scheduling** to share CPU.

(b) Memory Management

- Allocates and frees RAM for processes.
- Uses **paging / segmentation** for efficient usage.

Example:

- If a program needs 1GB RAM but only 512MB is free → kernel uses **virtual memory (swap space)**.

(c) File System Management

- Provides an organized way to store and retrieve files.
- Maintains **inodes, directories, permissions**.

Example:

- When user opens /home/user/file.txt, the kernel looks up the **inode number** and retrieves data blocks.

(d) Device Drivers

- Special programs inside kernel that control hardware.
- Convert generic I/O requests into device-specific commands.

Example:

- When you press "Print" → the kernel's **printer driver** converts it into printer commands.

(e) Inter-Process Communication (IPC)

- Mechanisms for processes to exchange data.
- Includes **pipes, message queues, shared memory, semaphores**.

Example:

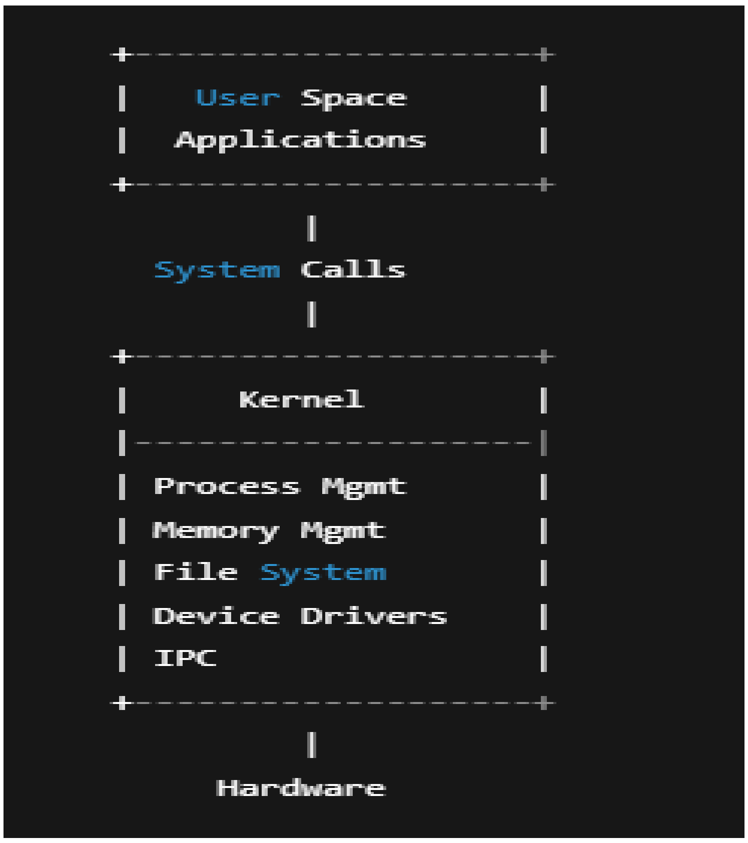
- A **media player** (decoding process) passes frames to a **renderer process** using shared memory.

4. Hardware (Physical Layer)

- The actual physical components: **CPU, RAM, Disk, Keyboard, Network card**.
- Kernel translates software requests into hardware operations.

Example:

- User types "A" → keyboard hardware sends interrupt → kernel handles it → character appears on screen.



```
+--------------------+
|    User Space      |
|   Applications     |
+--------------------+
         |
   System Calls
         |
+--------------------+
|       Kernel       |
|--------------------|
|  Process Mgmt      |
|  Memory Mgmt       |
|  File System       |
|  Device Drivers    |
|  IPC               |
+--------------------+
         |
      Hardware
```

Kernel Data Structures:

The kernel maintains **special data structures** in memory to manage processes, files, and devices. Some important kernel data structures are:

1. Process Table (Task Control Block)

- Stores information about each **process**.
- Fields: Process ID (PID), state (Running, Ready, Waiting), priority, program counter, CPU registers, open files.
- **Example:** When you run `ps` in Linux, you see data from the process table.

2. File Table (Inode Table / File Descriptor Table)

- Stores information about **open files**.
- Fields: File name, inode number, file pointer, access permissions.
- **Example:** When you open `file.txt`, its inode entry is added to the file table.

3. Memory Management Structures

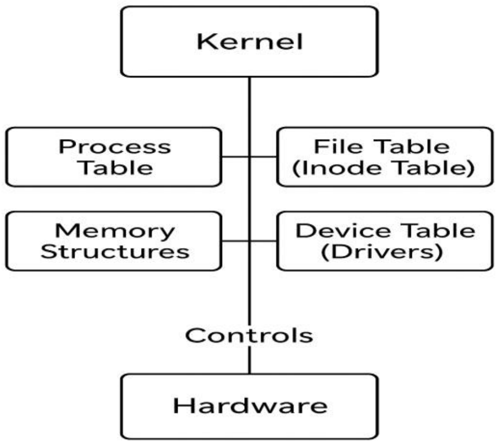
- Keeps track of memory allocation using:
  - **Page tables** (for virtual memory mapping)
  - **Free lists** (for free/allocated memory blocks)
- **Example:** When a process loads, the kernel updates its page table mapping virtual to physical addresses.

4. Buffer Cache / I/O Queues

- Kernel maintains buffer headers and free lists for disk I/O.
- Keeps frequently accessed blocks in memory to avoid repeated disk access.

5. Device Table

- Contains information about **I/O devices**.
- Each entry has device ID, status, buffer pointer, driver information.
- **Example:** When you plug in a USB, the kernel updates device table with its entry.



Q-4: Outline Inode and Structure of regular file. Also describe Conversion of a pathname to an inode alongside with inode assignment to a new file with example.

Inode (Index Node):

- An **inode** is a data structure used by UNIX-like file systems to store information about files.
- Every file (regular file, directory, device file) is represented by an inode.
- **Important:** Inode does **not store filename**, only file metadata. Filenames are stored in directories as mappings: **(filename → inode number)**.

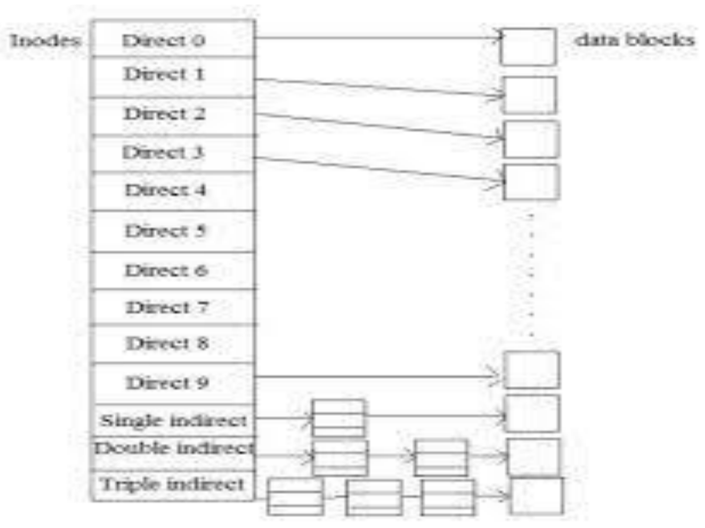
Structure of Inode:

Each inode contains:

1. **File Metadata**
   - File type (regular file, directory, device, etc.)
   - File size
   - Permissions (rwx for owner, group, others)
   - Owner UID and Group GID
   - Timestamps (created, modified, accessed)
   - Link count (number of directory entries pointing to inode)

2. **Disk Block Pointers**
   - Direct pointers (point to actual data blocks)
   - Single indirect pointer (points to block containing addresses of data blocks)
   - Double indirect pointer (block → contains block addresses → contains data addresses)

- Triple indirect pointer (3 levels of indirection)



Accessing Inodes:

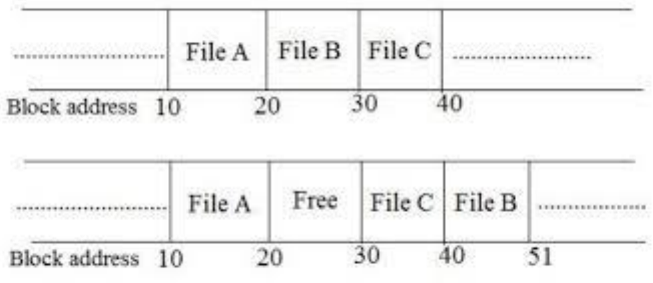
(add allocation of In-core Inode algorithm here)

Releasing Inodes:

(add releasing an Inode algorithm here)

Structure of a Regular File:

- A **regular file** is stored as **data blocks** on disk. Inode points to those blocks.
- If file is small → stored directly with **direct pointers**.
- If file is large → inode uses **indirect blocks** to extend storage.



- Each file is allocated sequentially in fixed blocks.
  - File A → Block 10
  - File B → Block 20
  - File C → Block 30
- Suppose **File B is deleted** → Block 20 becomes **Free**.
- Later, when File B is stored again, instead of going back to Block 20, the system puts it in the **next available free block (Block 51)**.

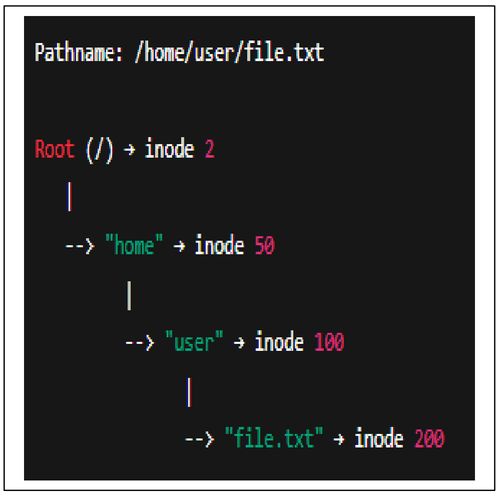
Conversion of a Pathname to an Inode:

In **UNIX / Linux file systems**, every file is identified by an **inode number**. When a process provides a **pathname** (like /home/user/file.txt), the kernel must **translate this pathname into the corresponding inode** to access the file.

- **Start at the Root Inode**
  - Root directory / always has a **fixed inode number (usually 2 in UNIX)**.
  - Kernel loads this inode into memory.
- **Parse Path Components**
  - Split pathname into its components (example: home, user, file.txt).
- **Search Each Directory in Sequence**
  - For each component:
    - Look up the directory's **data blocks** (directory entries).
    - Each directory entry maps **filename → inode number**.
- **Fetch Next Inode**
  - Once inode for the next component is found, load it into memory.
  - Continue until last component is resolved.
- **Return Final Inode**
  - The final inode number corresponds to the requested file.
  - With this inode, OS can access file metadata and data blocks.

Example: Pathname → /home/user/file.txt

1. Start at **Root /**
   - **Inode 2** (fixed root inode).
2. Look for "home" inside / directory.
   - Directory entry: "home" → inode 50.
3. Now go to **inode 50 (home directory)**.
   - Search for "user".
   - Entry: "user" → inode 100.
4. Now go to **inode 100 (user directory)**.
   - Search for "file.txt".
   - Entry: "file.txt" → inode 200.
5. Finally, **inode 200** is the inode of /home/user/file.txt.



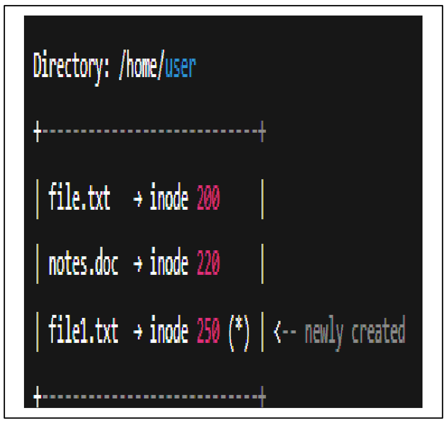
```
Pathname: /home/user/file.txt


Root (/) → inode 2
   |
   --> "home" → inode 50
          |
          --> "user" → inode 100
                 |
                 --> "file.txt" → inode 200
```

Inode Assignment to a New File:

Every file in UNIX/Linux is uniquely identified by an **inode number**. When you create a new file, the **file system must assign a free inode** from its inode table.

1. **File Creation Request:** A process issues a system call like:
    - User creates new file (create file1.txt).
    - **$touch file1.txt**
    - Using stat to See Inode: **$stat file1.txt**

```
  File: file1.txt
  Size: 0         Blocks: 0         IO Block: 4096 regular empty file
Device: 802h/2050d Inode: 523456      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/ abc)   Gid: ( 1000/ abc)
```



```
Directory: /home/user

+---------------------------+

| file.txt   → inode 200     |

| notes.doc  → inode 220     |

| file1.txt  → inode 250 (*) | <-- newly created

+---------------------------+
```

    - Kernel searches for a **free inode** in inode table.
    - Assigns inode number (say inode 250). file1.txt → inode 250.
    - Initializes inode fields: owner UID, permissions, size = 0, timestamps.
    - Updates parent directory (/home/user) with new entry.
    - Add execute permission for user: **$ chmod u+x file1.txt**
    - Set exact permissions (numeric mode): **$ chmod 644 file1.txt**
      (644→-rw-r--r--)

2. **Search for Free Inode**
    - The file system maintains an **inode table**.
    - The kernel searches for a **free inode** in this table.
    - Once found, it marks it as **allocated**.
3. **Fill Inode Metadata**
    - Inode structure stores **metadata** (not the name!):
        - File type (regular, directory, etc.)
        - Owner (UID, GID)
        - File permissions (e.g., -rw-r--r--)
        - Timestamps (created, modified, accessed)
        - Pointers to disk blocks (where data will be stored)
    - These details are filled in the newly assigned inode.
4. **Directory Entry Update**
    - The **directory (like /home/user/)** where the file is created is updated.
    - It creates a **mapping**: "notes.txt" → inode number (say 250)
    - This way, filenames are just labels pointing to inode numbers.
5. **Return File Handle to Process**
    - The process gets a **file descriptor**, which refers to the inode internally.
    - Now the process can read/write data.