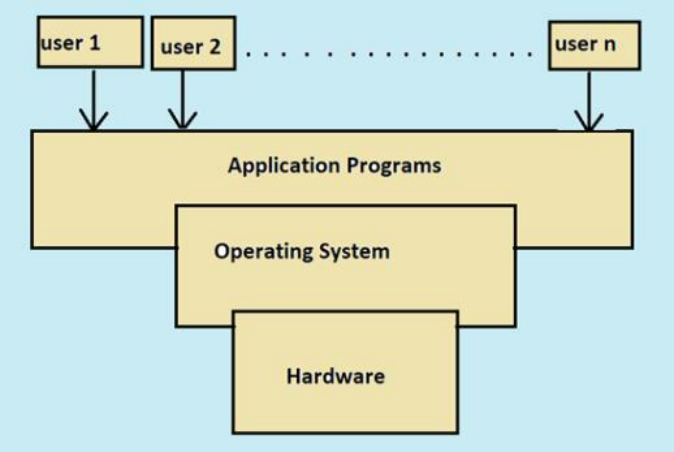
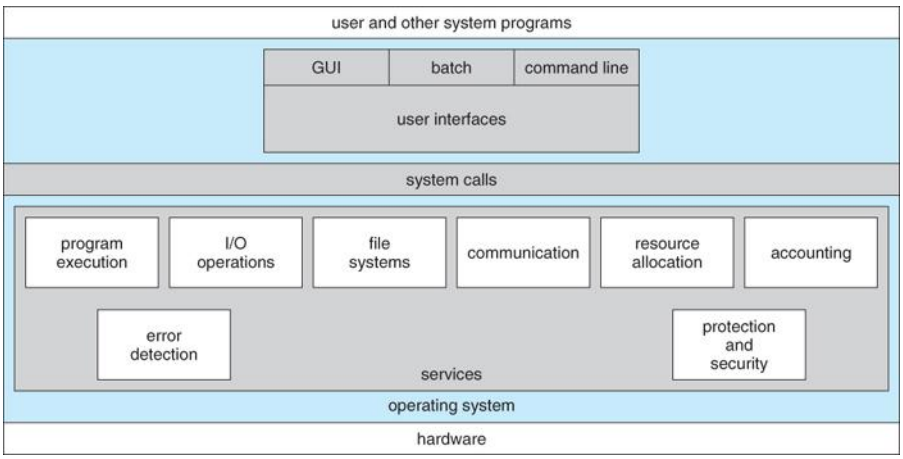
Operating Systems Cheat-Sheet

An OS is system software that manages computer hardware and software resources, providing essential services for application programs.



OS Services

1. **Process Management** – Handles process creation, execution, and termination.
2. **Memory Management** – Allocates and deallocates memory.
3. **File Management** – Controls file operations (read, write, delete).
4. **Device Management** – Manages I/O devices using drivers.
5. **Security & Protection** – Restricts unauthorized access.
6. **User Interface** – CLI (Command Line Interface) or GUI (Graphical User Interface).



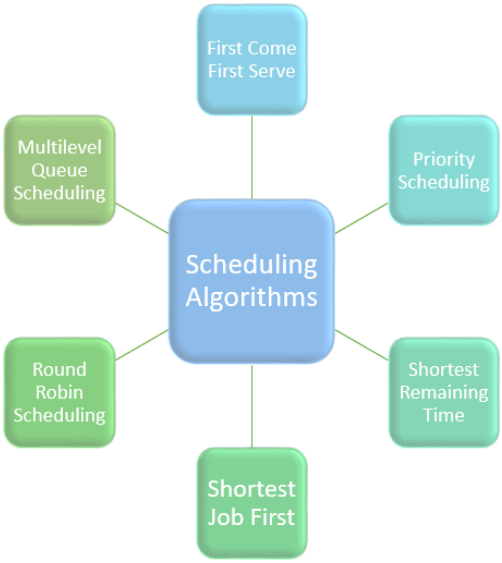
System Calls

System calls allow user programs to interact with the OS.

Types of System Calls

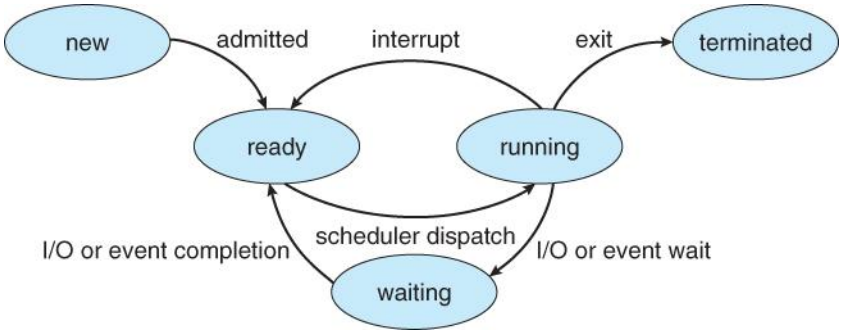
- **Process Control** – fork(), exit(), exec().
- **File Management** – open(), read(), write().
- **Device Management** – ioctl(), read(), write().
- **Information Maintenance** – getpid(), alarm().
- **Communication** – pipe(), send(), recv().

Process Scheduling Algorithms



1. **FCFS (First-Come, First-Served)** – Non-preemptive, executes processes in arrival order. Simple but causes the **convoy effect**.
2. **SJF (Shortest Job First)** – Picks the shortest process first. Efficient but can cause **starvation** of long processes.
3. **Round Robin (RR)** – Each process gets a fixed time slice. Ensures fairness but increases **context switching overhead** if the time slice is too small.
4. **Priority Scheduling** – Executes higher-priority processes first. Useful but can lead to **starvation** of low-priority processes.
5. **Multilevel Queue** – Divides processes into separate queues, each with its own scheduling algorithm. Good for handling different workloads.

Process States & Life Cycle

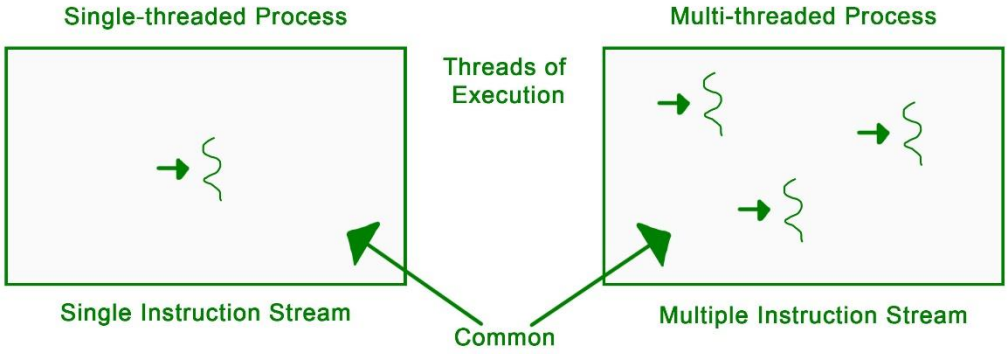


Process States:

1. **New** – Process is created.
2. **Ready** – Waiting for CPU.
3. **Running** – Executing on CPU.
4. **Waiting** – Waiting for an event (I/O completion).
5. **Terminated** – Process execution is complete.

Threads & Multithreading Models

A thread is the smallest unit of CPU execution.



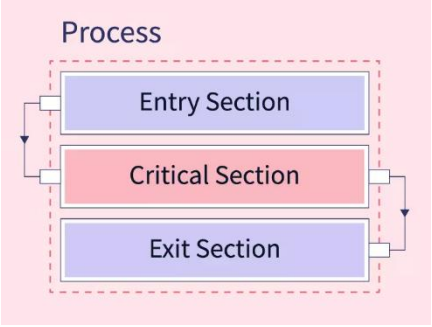
Single Thread and Multi Thread Process

Multithreading Models

1. **Many-to-One** – Multiple user threads mapped to a single kernel thread.
2. **One-to-One** – Each user thread has a kernel thread.
3. **Many-to-Many** – Multiple user threads mapped to multiple kernel threads.

Critical Section

The **critical section** is a part of a process where shared resources (such as variables, files, or memory) are accessed. To prevent **race conditions**, where multiple processes modify shared data inconsistently, only one process should execute the critical section at a time.



Critical Section Problem

The main goal is to ensure that multiple processes do not enter the critical section simultaneously while maintaining system efficiency.

Solution Requirements (Critical Section Design Goals)

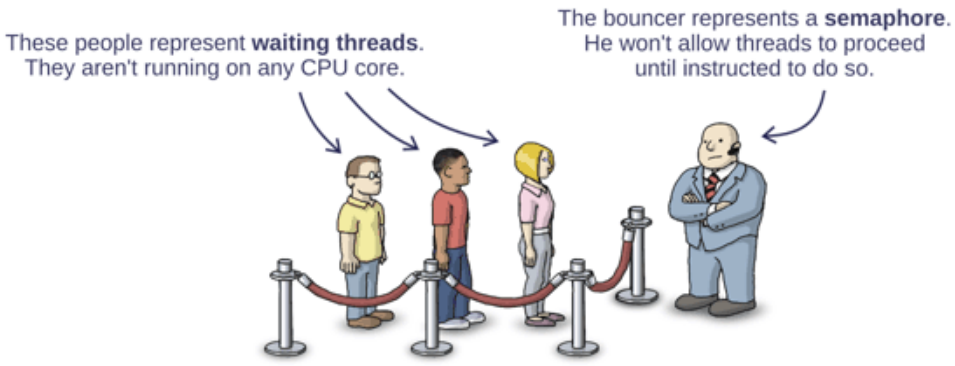
1. **Mutual Exclusion** – Only one process can enter the critical section at a time.
2. **Progress** – If no process is in the critical section, one of the waiting processes should be allowed to enter.
3. **Bounded Waiting** – A process should not wait indefinitely to enter the critical section.

Synchronization Mechanisms to Handle Critical Sections

- **Peterson's Solution** – A software-based approach using flags and turn variables.
- **Locks (Mutex)** – Ensures that only one process accesses a resource at a time.
- **Semaphores** – Signaling mechanisms to control process access.
- **Monitors** – High-level synchronization constructs that encapsulate shared resources.

Semaphores

A **semaphore** is a synchronization tool used to control access to shared resources in a multi-process or multi-threaded environment. It helps prevent **race conditions** and ensures **mutual exclusion**.



Types of Semaphores

1. **Binary Semaphore (Mutex)**
   - Can have only two values: 0 (locked) and 1 (unlocked).
   - Used for **mutual exclusion** (only one process can enter the critical section).
2. **Counting Semaphore**
   - Can have a value greater than 1.
   - Used to manage multiple resources (e.g., controlling access to a limited number of database connections).

Operations on Semaphores

Semaphores work with two atomic operations:

1. **wait(S) / P(S) (Proberen/Test)**
   - Decreases the semaphore value.
   - If S > 0, the process enters; otherwise, it waits.
   - **signal(S) / V(S) (Verhogen/Increment)** Increases the semaphore value.
   - Wakes up a waiting process if any.

Semaphore Implementation Example (Pseudocode)

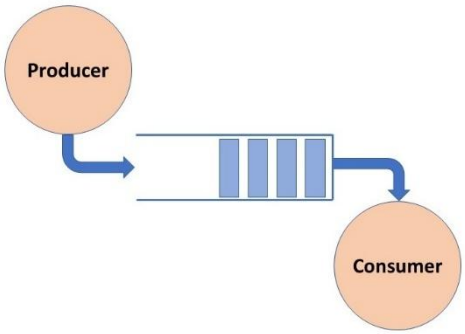
```
Semaphore S = 1; // Binary semaphore initialized

void process() {
    wait(S);  // Acquire lock
    // Critical Section
    signal(S); // Release lock
}
```

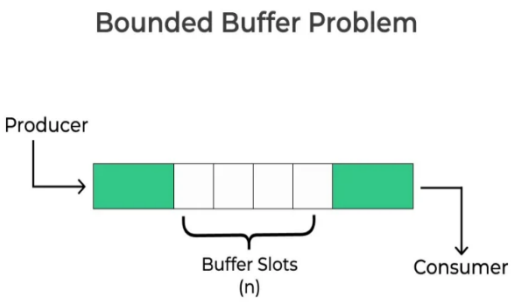
Use Cases of Semaphores

- **Mutual Exclusion (Mutex Locks)** – Prevent multiple processes from accessing a shared resource.
- **Producer-Consumer Problem** – Synchronizes producer and consumer threads.
- **Reader-Writer Problem** – Controls access to shared data for readers and writers.
- **Dining Philosophers Problem** – Prevents deadlocks when multiple processes compete for limited resources.

Classic Synchronization Problems

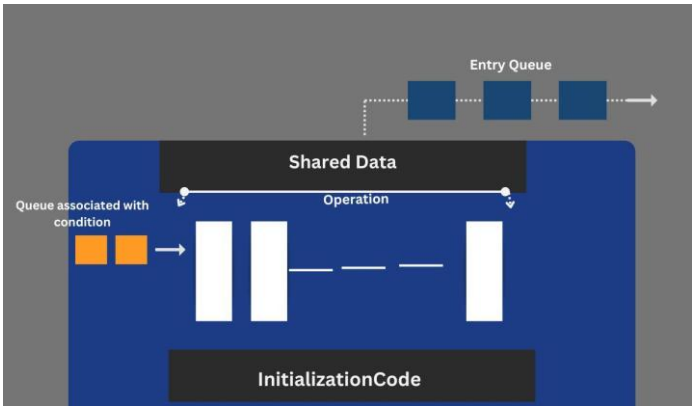
1. **Producer-Consumer** – Producers add data, consumers remove it.
2. **Dining Philosophers** – Preventing deadlock when philosophers need shared chopsticks.
3. **Readers-Writers** – Managing multiple readers and writers for a shared resource.





Monitors

A **monitor** is a high-level synchronization construct that ensures mutual exclusion by allowing only one process or thread to access shared resources at a time. It simplifies synchronization compared to semaphores by automatically managing locks and condition variables.



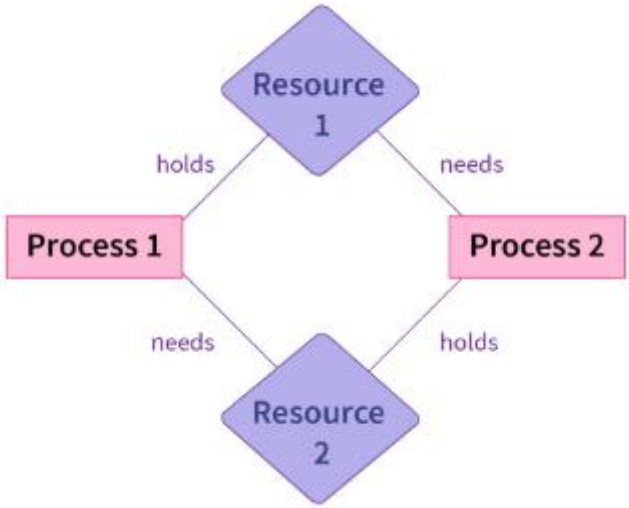
Structure of a Monitor

A monitor consists of:

1. **Shared Variables** – Variables that need controlled access.
2. **Procedures (Methods)** – Functions that define operations on shared variables.
3. **Condition Variables** – Special variables used to block or wake up processes when specific conditions are met.

Deadlocks

A **deadlock** is a situation where two or more processes are indefinitely waiting for each other to release resources, preventing further execution. It occurs when a circular chain of processes holds resources that others need.



Necessary Conditions for Deadlock (Coffman's Conditions)

A deadlock occurs if all four conditions are met:

1. **Mutual Exclusion** – Only one process can use a resource at a time.
2. **Hold and Wait** – A process holding at least one resource is waiting for more.
3. **No Preemption** – A resource cannot be forcibly taken from a process.
4. **Circular Wait** – A closed chain of processes exists, where each process holds a resource the next one needs.

Deadlock Prevention

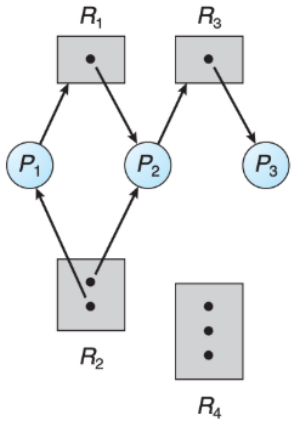
Deadlock prevention aims to break at least one of Coffman's conditions:

1. **Mutual Exclusion** – Eliminate the need for resource locking by allowing shared access (not always feasible).
2. **Hold and Wait** – Require a process to request all resources before execution starts, preventing partial resource holding.
3. **No Preemption** – If a process is waiting, take resources away from it and give them to another process (not always practical).
4. **Circular Wait** – Impose a **resource hierarchy** where processes request resources in a predefined order, avoiding circular dependency.

Deadlock Detection

If deadlock prevention is not feasible, the system must **detect** and resolve deadlocks.

- **Resource Allocation Graph (RAG)** – A directed graph where:
  - **Nodes** represent processes and resources.
  - **Edges** represent resource assignments and requests.
  - **Cycles in RAG indicate deadlocks.**
- **Detection Algorithm:**
  - Periodically check for cycles in RAG.
  - If a cycle is found, **terminate** or **preempt** processes to break the deadlock.



Deadlock Avoidance (Banker's Algorithm)

Banker's Algorithm prevents deadlocks by ensuring the system never enters an unsafe state.

Steps in Banker's Algorithm:

Initialize

- Each process declares its **maximum resource need**.
- The system tracks **available resources** and **allocated resources**.

Request Handling

- A process requests resources.
- The system checks if the request **exceeds the declared maximum**—if yes, it is denied.

Safety Check

- Temporarily allocate the requested resources.
- Check if there exists a **safe sequence** (an order in which all processes can finish without deadlock).
- If a safe sequence **exists**, approve the request; otherwise, deny it.

Resource Allocation

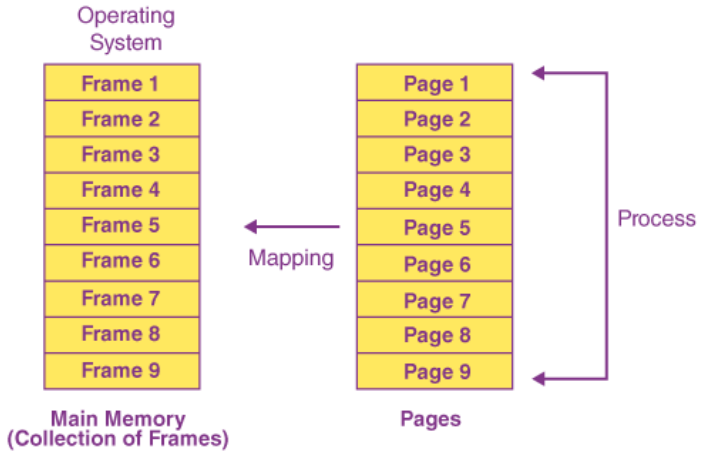
- If approved, resources are assigned.
- If denied, the process must wait.

Safe State vs. Unsafe State

- **Safe State** – The system can allocate resources in some order without leading to deadlock.
- **Unsafe State** – There is a risk of deadlock if resources are allocated.

Paging

Paging is a **memory management technique** that eliminates fragmentation by dividing memory into fixed-size blocks. It allows processes to be loaded into non-contiguous memory locations, improving **efficiency and memory utilization**.



How Paging Works

1. **Logical Memory Division**
    - The process's **address space** is divided into fixed-size **pages** (e.g., 4 KB each).
2. **Physical Memory Division**
    - RAM is divided into **frames** of the same size as pages.
3. **Mapping Pages to Frames**
    - When a process is loaded, its pages are placed into available frames.

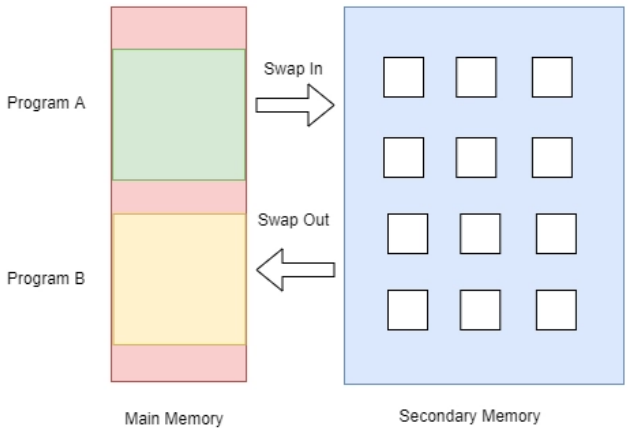
- A **page table** maintains the mapping of **logical page numbers** to **physical frame numbers**.

4. **Address Translation (Paging Hardware)**

   - The **logical address** (generated by the CPU) consists of:
     Page Number (p) – Indexes the **page table**.
     Offset (d) – Specifies the exact location within the page.
   - The page table maps **page numbers** to **frame numbers**, and the offset gives the final **physical address** in RAM.

Demand Paging

Demand paging is a **virtual memory management** technique where **pages are loaded into memory only when needed**. This reduces the initial memory load and improves system performance.



How Demand Paging Works

1. **Lazy Loading (On-Demand Page Loading)**

   - Unlike **normal paging**, where all pages of a process are loaded at once, **demand paging loads pages only when a process requests them**.

2. **Page Table and Page Faults**

   - The OS maintains a **page table** that tracks which pages are in memory.
   - If a process requests a page **not in memory**, a **page fault** occurs.
   - The OS fetches the missing page from the **hard disk (swap space)** and loads it into RAM.

3. **Page Replacement (When Memory is Full)**

   - If memory is **full**, the OS replaces an existing page using a **Page Replacement Algorithm** (e.g., FIFO, LRU).

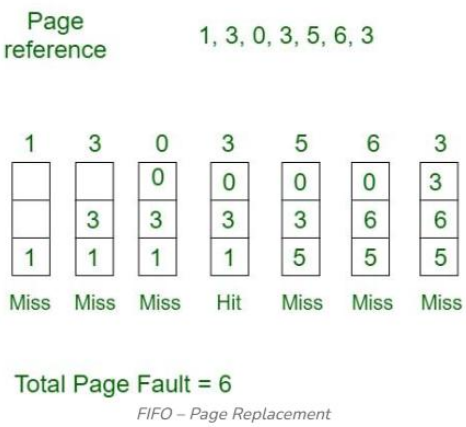
Demand Paging vs. Normal Paging

Feature	Normal Paging	Demand Paging
Page Loading	Preloaded into memory	Loaded only when required
Page Faults	Less frequent	More frequent (but manageable)
Performance	Faster execution	Can be slow if thrashing occurs
Memory Usage	Higher (more pages loaded)	Lower (only needed pages are loaded)

Page Replacement Algorithms

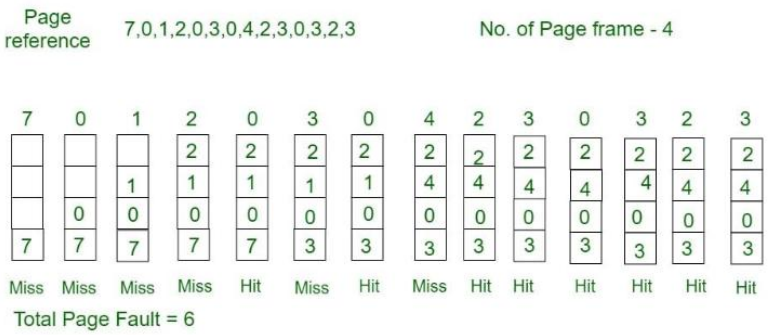
Page replacement algorithms are used in demand paging to decide which memory page to remove when a new page needs to be loaded, but RAM is full. The goal is to minimize page faults and improve system performance.

1. **FIFO (First In, First Out)** – Oldest page is replaced first.



FIFO – Page Replacement

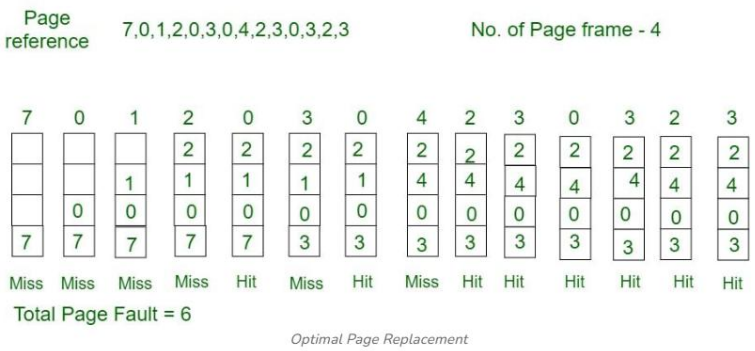
2. **LRU (Least Recently Used)** – Replaces the least recently used page.



Here LRU has same number of page fault as optimal but it may differ according to question.

Least Recently Used – Page Replacement

3. **Optimal Page Replacement** – Replaces the page that won't be needed for the longest time.



Optimal Page Replacement

File Concepts

A **file** is a collection of related data stored on a disk, identified by a unique name and organized in a file system. It serves as a unit of storage, allowing users and applications to read, write, and modify data.

File Attributes

Each file has **metadata (attributes)** stored by the operating system:

1. **Name** – The human-readable name of the file.
2. **Type** – Defines the format (e.g., .txt, .exe, .jpg).
3. **Location** – Physical address on storage (disk location).
4. **Size** – The total file size in bytes.
5. **Protection** – Permissions defining who can read, write, or execute.
6. **Creation Time & Modification Time** – Timestamp of when the file was created or last modified.
7. **Owner** – Identifies the user who owns the file.

File Operations

Files support various operations:

1. **Create** – A new file is created in the system.
2. **Open** – A file is accessed for reading or writing.
3. **Read** – Data is retrieved from the file.
4. **Write** – Data is modified or added to the file.

5. **Append** – Adds data at the end of a file.
6. **Delete** – Removes the file from the system.
7. **Truncate** – Clears the content while keeping the file.
8. **Close** – Ends access to the file, freeing system resources.

File Types

File Type	Standard Extension
executable	bin, jar, none
object	obj, o
source code	c, h, py, java, wsdl, cpp, hpp
batch	sh, csh
text	doc, txt, pdf, ps
word processor	doc, tex, wp, rrf
library	a, so
archive	tar, rpm, deb

File Access Methods

Define how data is read and written in a file, optimizing retrieval and modification.

1. Sequential Access

- Data is read **in order**, like reading a book page by page.
- **Operations:** read next, write next
- **Example:** Text files, logs
- **Pros:** Simple, efficient for large files.
- **Cons:** Slow for random access.

2. Direct (Random) Access

- Data is accessed **directly** using an index.
- **Operations:** read(position), write(position)
- **Example:** Databases, executable files
- **Pros:** Fast retrieval for large structured data.
- **Cons:** Needs additional indexing.

3. Indexed Access

- Uses an **index table** to locate data blocks.

- **Operations:** find(index), read(index)
- **Example:** Databases, library catalogs
- **Pros:** Faster than sequential for large data.
- **Cons:** Requires extra storage for indexing.

4. Hashed Access

- Uses a **hash function** to map file locations.
- **Operations:** hash(key), retrieve(key)
- **Example:** Database indexing, caching
- **Pros:** Very fast for large datasets.
- **Cons:** May cause **collisions**, needing resolution.

Access Matrix

An **Access Matrix** is a **security model** that defines the **permissions** (read, write, execute, etc.) that users or processes have on system resources (files, devices, memory, etc.). It helps in implementing **access control** in an operating system.

Structure of Access Matrix

- **Rows** → Represent **Subjects** (Users, Processes).
- **Columns** → Represent **Objects** (Files, Devices, Resources).
- **Cells** → Define **Permissions** (Read, Write, Execute, etc.).

Example of an Access Matrix:

Object/Subject	User A	User B	Process P
File 1	R/W	R	X
File 2	R	R/W	-
Printer	X	-	W

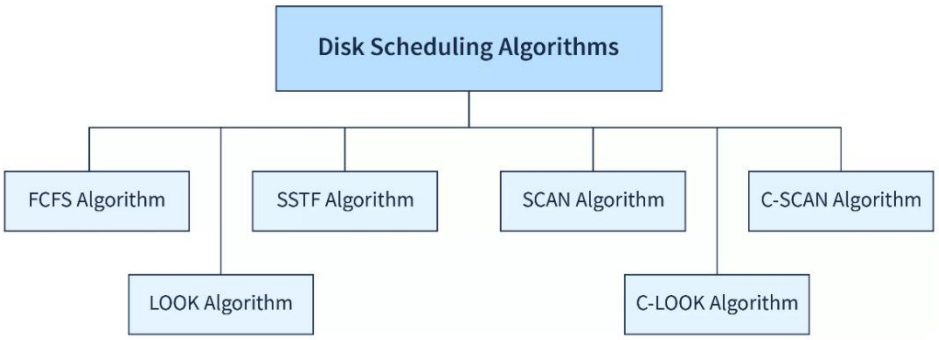
- **R (Read)** – Subject can view the object.
- **W (Write)** – Subject can modify the object.
- **X (Execute)** – Subject can run the object.
- **- (No Access)** – Subject cannot access the object.

Access Control Using Access Matrix

1. **Each subject has a set of rights** for different objects.
2. **Enforces security policies** to restrict unauthorized access.
3. **Can be implemented as:**
    - **Access Control Lists (ACLs)** – Each object has a list of subjects with permissions.
    - **Capability Lists** – Each subject has a list of objects it can access.

Disk Scheduling Algorithms

Disk scheduling algorithms determine the order in which disk I/O requests are served to **minimize seek time** and improve efficiency.



1. FCFS (First Come, First Serve)

- Requests are processed **in arrival order**, without optimization.
- **Pros:** Simple and fair.
- **Cons:** Inefficient for large workloads, causes **long wait times**.

2. SSTF (Shortest Seek Time First)

- Selects the **nearest request** to reduce seek time.
- **Pros:** Faster than FCFS.
- **Cons:** Can cause **starvation** (far requests may never get served).

3. SCAN (Elevator Algorithm)

- Moves in **one direction**, serving requests, then reverses.
- **Pros:** Reduces seek time compared to FCFS.
- **Cons:** Higher wait times for requests just behind the head.

4. C-SCAN (Circular SCAN)

- Moves in **one direction only**, jumping back to the start after reaching the last request.
- **Pros:** Provides **uniform wait times**.
- **Cons:** May result in **higher overall seek time**.

5. LOOK & C-LOOK

- Like SCAN & C-SCAN, but **stops at the last request** instead of going to the disk's end.
- **Pros:** Avoids unnecessary movement, improving efficiency.
- **Cons:** Slightly complex compared to SCAN and C-SCAN.

Source Links:

geeksforgeeks/operating-systems

tutorialspoint.com/operating_system

techtarget/operating-system-OS

byjus.operating-systems

Operating-System.pdf