Operating System

What is an Operating System?

- OS is a **software** that manages computer **hardware** .
- It acts as an **intermediary** between computer hardware and user applications.
- Purpose of OS is to provide an environment in which a user can execute programs conveniently and efficiently.

What is a Kernel?

- A kernel is the core component of an operating system that serves as a bridge between the computer hardware and the applications running on it.

Key Functions of a Kernel

- **Process Management**
  - The kernel manages processes. It allocates resources, schedules tasks, and controls the execution of processes.
- **Memory Management**
  - Allocates and deallocates memory for processes and manages the virtual memory space. It handles tasks such as address translation and page swapping.
- **File System Management**
  - Manages files and directories, providing an interface for reading, writing, and organizing data on storage devices. It handles file permissions and storage access.
- **Device Drivers**
  - Controls and communicates with hardware devices, such as printers, disk drives, network interfaces, and input/output devices. Device drivers are modules within the kernel that facilitate this communication.
- **System Calls**
  - Acts as an interface for system calls, allowing user-level applications to request services from the operating system. System calls are predefined routines that provide access to the kernel's functionality.

Contd...

- **Interrupt Handling**
  - Manages interrupts generated by hardware devices or exceptional conditions. The kernel handles these interrupts to ensure proper system operation.
- **Process Synchronization and Communication**
  - Facilitates communication and synchronization between processes. The kernel provides mechanisms like semaphores, mutexes, and inter-process communication (IPC) to coordinate activities.
- **Kernel Space and User Space**
  - Separates the operating system's critical components (kernel space) from user-level applications (user space).
  - This isolation enhances stability, security, and system integrity.
- **Bootstrapping**
  - The kernel is responsible for bootstrapping the system, initializing hardware components, loading necessary drivers, and starting the user-space processes.

Types of Kernels

- **Monolithic Kernel**
  - Both user services and kernel services are implemented under the same address space. This makes operating system execution faster.
  - This kernel provides CPU scheduling, memory management, file management, and other operating system functions through system calls.
  - Examples: Linux and older versions of Unix.
- **Microkernel:**
  - Essential functions, such as process and memory management, reside in a small, protected kernel space. Additional functionalities are implemented as separate user-level processes.
  - Examples: QNX and L4.
- **Hybrid Kernel:**
  - Combines elements of both monolithic and microkernel architectures, aiming to provide the benefits of both.
  - Examples: Windows NT and macOS.

What is a Process?

- A process is an executing instance of a program.
- Managed by the operating system
  - OS schedules them for execution, allocates resources, and ensures their proper termination.

Program Vs Process

Program

- Program contains a set of instructions designed to complete a specific task.
- Program is a passive entity as it resides in the secondary memory.
- The program exists in a single place and continues to exist until it is deleted.
- Program does not have any resource requirement, it only requires memory space for storing the instructions.

Process

- Process is an instance of an executing program.
- Process is an active entity as it is created during execution and loaded into the main memory.
- Process exists for a limited span of time as it gets terminated after the completion of a task.
- Process has a high resource requirement, it needs resources like CPU, memory address, and I/O during its lifetime.

Key Components of a Process

1. **Program Counter (PC)**
   - Keeps track of the address of the next instruction to be executed.
2. **Registers**
   - Used for storing intermediate data during process execution.
3. **Memory Space**
   - Contains the code, data, and stack segments.
4. **State Information**
   - Includes information about the process's state, such as whether it's ready, running, or waiting.

Different States of a Process

- **New**: The process is being created.
- **Ready** : The process is waiting for execution.
- **Running** : The process is currently being executed.
- **Waiting** : The process is waiting for an event (e.g., I/O operation) to complete.
- **Terminated** : The process has finished its execution.

What is a Process Control Block (PCB) ?

- It's a **data structure** maintained by the operating system for each process.
- It contains information about
  - Process's state
  - Program counter
  - Registers
  - Memory space ...
- Allows the operating system to save and restore the process's state during context switches.

What is a Process Table?

- An array of PCBs, that means logically contains a PCB for all of the **current processes** in the system.
- Use
  - To keep track of all **active processes.**
  - **Scheduling** : To decide which process to execute next based on their states and priorities.
  - **Resource Allocation** : Provides information about the resources allocated to each process.
- Key Components of a Process Table
  - **Process ID (PID)** : A unique identifier for each process.
  - **Program Counter (PC)** : The current value of the program counter for the process.
  - **Registers** : The values of the CPU registers for the process.
  - **Memory** : Information about the memory allocated to the process.
  - **State Information** : The current state of the process (e.g., ready, running, waiting).
  - **Priority** : The priority assigned to the process for scheduling purposes.

What is a Thread?

- A thread is
  - Smallest unit of execution within a process
  - Segment of a process
- A process has its own memory space and resources, while a thread shares the same resources with other threads within the same process.
- Threads within a process can run independently, allowing for concurrent execution.
- For example, in a browser, multiple tabs can be different threads.
- Key Concepts
  - **Thread ID** (TID): Each thread within a process is identified by a unique Thread ID.
  - **Program Counter** (PC): Threads within a process share the same program code, but they have their own program counter indicating the next instruction to be executed.
  - **Registers** : While threads share the same code, they have their own set of registers.
  - **Stack**: Threads have their own stack space, allowing for independent function calls and local variable storage.
  - **Shared Resources** : Threads within the same process share resources like file descriptors, heap memory, and other process-related data.

Advantages of Thread

- **Concurrency** : Threads allow multiple operations to be executed concurrently within the same process.
- **Responsiveness** : While one thread is waiting for resources, other threads can continue execution, improving overall system responsiveness.
- **Resource Sharing** : Threads within the same process share resources, making communication and data sharing between them more efficient than between separate processes.

Thread Synchronization

- Threads within a process may need to synchronize their activities to avoid conflicts.
- Synchronization mechanisms to ensure proper coordination and data consistency:
  - Locks
  - Semaphores
  - Monitors

Process vs Thread

Process

1. Has its own memory space and resources.(The process has its own Process Control Block, Stack and Address Space.)
2. Communicates through IPC mechanisms.
3. Process switching uses, another process interface in operating system.
4. Independent of other processes.
5. Creating and terminating processes is slower.
6. Processes run independently, achieving parallelism.
7. More robust since a failure in one process doesn't affect others.

Thread

1. Shares memory space and resources with other threads in the same process.(Thread has Parents' PCB, its own Thread Control Block and Stack and common Address space.)
2. Communicates more efficiently through shared variables and direct function calls.
3. Thread switching does not require to call an operating system and cause an interrupt to the kernel.
4. Threads within a process are not fully independent.
5. Creating and terminating threads is faster.
6. Threads run concurrently within the same process.
7. Less robust; a failure in one thread can impact the entire process.

What is Multithreading?

- Multithreading is the concurrent execution of multiple threads within the same process.
  - Multi-threading is sharing of computing resources among threads of a single process.
- It aims to improve application performance by taking advantage of modern multi-core processors.
- Benefits
  - The idea is to achieve parallelism by dividing a process into multiple threads.
  - **Responsiveness:** While one thread is waiting for an operation (e.g., I/O), other threads can continue their execution. This results in more responsive applications, especially in user interfaces.
  - **Efficient Resource Utilization:** Threads within the same process share the same resources, such as memory space and file descriptors, reducing overall resource consumption compared to separate processes.
  - **Enhanced Scalability:** Multithreading can lead to better scalability, especially in applications that need to handle a large number of concurrent tasks or users.

What is Multiprogramming?

- Multiprogramming is a management technique where multiple programs are kept in main memory simultaneously and executed concurrently.
- The goal of multiprogramming is to maximize CPU utilization and increase overall system throughput by allowing the processor to switch between executing different programs.
- The main objective of multi-programming is to keep multiple jobs in the main memory. If one job gets occupied with IO, the CPU can be assigned to other jobs.

What is Multitasking?

- Multitasking is a logical extension of a multiprogramming that allows multiple tasks or processes to run concurrently on a computer system.
    - Multitasking is sharing of computing resources(CPU, memory, devices, etc.) among processes.
- It refers to the ability of an operating system to manage and switch between different tasks in a way that gives the illusion of simultaneous execution.
- Multitasking enables users to perform several tasks seemingly at the same time, even though the central processing unit (CPU) is actually rapidly switching between tasks.

What is Context switching?

- Context switching occurs when a process is preempted or completes its execution.
- The context switch involves saving the current state of the process and loading the state of the next process.
- In Context Switching the process is stored in the Process Control Block to serve the new process so that the old process can be resumed from the same part it was left.

What is Thrashing?

- Thrashing is a phenomenon characterized by **excessive paging or swapping** of data between the main memory (RAM) and the secondary storage (usually a hard disk) without making any significant progress in terms of overall system performance.
- Thrashing results in
  - High Page Fault Rate
  - Continuous Swapping
  - Decreased Throughput
  - Increased Disk I/O
  - Low CPU Utilization
  - Poor System Responsiveness
- Causes of Thrashing
  - Insufficient Memory
  - Overloading the System
  - Ineffective Page Replacement Policies
  - High Degree of Multiprogramming

What is a Buffer?

- A buffer is a temporary storage area used to store data that is being transferred between two locations, typically at different speeds or with different data processing times.
- Commonly employed to manage the flow of data between different components.
- The primary purpose of a buffer is to provide a temporary holding place for data, allowing for smoother and more efficient data transfer.

What is Virtual Memory?

- Virtual memory is a memory management technique to provide an **illusion** to the application that it has access to a **contiguous block** of memory, while, in reality, parts of the program may be stored in the main memory (RAM) and other parts on secondary storage (usually a hard disk).
- It allows for the efficient and flexible use of the available physical memory by creating an abstraction layer between the application and the actual hardware.

Key Concepts of Virtual Memory

- **Address Space**
  - Each process is given its own address space, starting from 0 to the maximum address.
  - This address space is divided into segments, such as code, data, and stack.
- **Page Table**
  - OS maintains a page table that **maps** the virtual addresses used by the application to the corresponding physical addresses in the main memory.
- **Page Size**
  - Virtual memory and physical memory are divided into fixed-size blocks called pages.
  - The page size is determined by the hardware and operating system but is typically 4 KB.
- **Page Fault**
  - When a program accesses a portion of its virtual address space that is not currently in the main memory, a page fault occurs.OS then brings the required page from secondary storage into the RAM.
- **Swapping**
  - OS can move pages between the RAM and the secondary storage as needed. This process is known as swapping.
- **Demand Paging**
  - Only the necessary pages are loaded into the main memory when required, rather than loading the entire program into RAM at once. The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging.

Advantages of Virtual Memory

- **Efficient Memory Utilization:**
  - Virtual memory allows for the illusion of a larger address space than physically available, optimizing the utilization of available memory.
- **Simplified Programming:**
  - Programmers can write programs assuming a large amount of contiguous memory is available, even if physical memory is limited.
- **Flexibility in Program Loading:**
  - Virtual memory allows for demand loading of pages, reducing the initial load time of programs and enabling faster application startup.
- **Ease of Multiprogramming:**
  - Multiple programs can be loaded into the main memory simultaneously, and the operating system can efficiently manage the swapping of pages between programs.

Challenges with Virtual Memory

- **Prformance Overhead:**
  - Due to the need for page table lookups and page faults when accessing data not currently in RAM.
- **Page Replacement Policies:**
  - Requirement for efficient page replacement policies to decide which pages to swap out when a new page needs to be loaded.
- **Disk I/O Bottlenecks:**
  - Frequent swapping of pages between RAM and disk can lead to disk I/O bottlenecks
- **Fragmentation:**
  - Virtual memory systems can suffer from fragmentation, both internal (within a page) and external (in the disk storage).

What is Fragmentation?

- Phenomenon where memory or storage space becomes **divided into small, non-contiguous segments** , making it challenging to allocate large contiguous blocks of memory or store files in contiguous clusters.
- Fragmentation can occur in both physical and virtual memory, as well as in storage devices.
- There are two main types of fragmentation
    - External fragmentation
    - Internal fragmentation

External Fragmentation

- Occurs when free memory is scattered throughout the system, but the available space is not contiguous.
- As a result, even though the total amount of free space may be sufficient, it cannot be used to satisfy a request for a large, contiguous block.
- Causes
    - **Allocation and Deallocation of Memory:** Repeated allocation and deallocation of memory can lead to small, non-contiguous free spaces.
    - **Variable-Sized Memory Allocation**
    - **Swapping and Paging:** In virtual memory systems, swapping and paging can cause fragmentation as pages are moved between the main memory and secondary storage.

Internal Fragmentation

- Occurs when allocated memory is larger than what is required by the program or data it holds, resulting in wasted space within the allocated block.
- Causes
  - **Fixed-Size Allocation:** Allocating fixed-size blocks of memory can lead to internal fragmentation when the allocated block is larger than necessary.

Solution to Fragmentation

- **Memory Pooling:** Pre-allocate fixed-size memory blocks and manage it as a pool. This reduces fragmentation by ensuring that only blocks of a specific size are allocated and deallocated.
- **Buddy System:** Allocates memory in powers of two. This structured approach minimizes fragmentation by reducing the likelihood of small, non-contiguous spaces.
- **First Fit Allocation:** Where the first available block that is large enough to accommodate the process is selected. This can help reduce external fragmentation.
- **Best Fit Allocation:** Where the smallest available block that can accommodate the process is selected. While this may increase internal fragmentation, it can help maintain smaller free spaces for future allocations.

What are different Scheduling Algorithms?

1. First-Come First-Served (FCFS) Scheduling.
2. Shortest-Job-Next (SJN) Scheduling.
3. Priority Scheduling.
4. Shortest Remaining Time.
5. Round Robin (RR) Scheduling.
6. Multiple-Level Queues Scheduling.

First Come First Serve (FCFS)

- It follows the principle of serving tasks in the **order they arrive** in the ready queue.
- FCFS is a non-preemptive scheduling algorithm, meaning that once a process starts its execution, it continues until completion without being interrupted.
- Drawbacks
  - **Convoy Effect**
    - The convoy effect occurs when shorter tasks are stuck behind a long-running task. This can lead to inefficiencies as shorter tasks wait for the completion of the longer one.
  - **Waiting Time Variability**
    - The waiting time for tasks can vary significantly, depending on the order in which tasks arrive and the duration of preceding tasks.

Shortest Job Next (SJN)

- It is a non-preemptive scheduling algorithm that selects the process with the **shortest total remaining processing time** to execute first.
- It aims to minimize the total processing time and reduce waiting times for processes.
- Drawbacks
  - SJN requires knowledge of the burst times of all processes in advance, which is often not available in practical scenarios.
  - Predicting the exact burst time is challenging, and variations can lead to increased waiting times.
- Variation
  - There is a preemptive version of this algorithm known as Shortest Remaining Time First (SRTF), where a running process can be preempted if a new process with a shorter burst time arrives.

Priority Scheduling

- Each process is assigned a priority, and the process with the highest priority is selected for execution first.
- Priority can be assigned dynamically based on the characteristics and behavior of the process, or it can be predefined and remain static throughout the process's lifetime.
- Priority scheduling can be preemptive or non-preemptive.
    - In **preemptive** priority scheduling, a running process can be interrupted and replaced by a higher-priority process.
    - In **non-preemptive** priority scheduling, the currently running process continues until completion or voluntary relinquishment of the CPU.
- **Aging**
    - Aging is a technique used to avoid starvation of lower-priority processes. It involves gradually increasing the priority of a process if it waits in the ready queue for an extended period.

Contd..

- Advantages
  - **Customization:** Allows customization based on the importance and characteristics of processes.
  - **Flexibility:** Priorities can be adjusted dynamically to adapt to changing conditions.
  - **High-Priority Tasks:** Critical tasks or real-time tasks can be given higher priority to ensure timely execution.
- Drawbacks
  - **Starvation:** Lower-priority processes may starve if higher-priority processes consistently occupy the CPU.
  - **Indefinite Blocking:** If a process with higher priority continuously arrives, lower-priority processes may face indefinite blocking.
  - **Complexity:** Managing and adjusting priorities dynamically can add complexity to the scheduling algorithm.

Round Robin (RR)

- It is a preemptive CPU scheduling algorithm that assigns a fixed time unit (time quantum or time slice) to each process in the system, ensuring all processes get an equal share of the CPU.
- Processes are arranged in a first-come-first-served (FIFO) queue. The process at the front of the queue gets CPU time for the duration of its time quantum.
- If a process's execution is not completed within its time quantum, it is temporarily suspended, and the CPU is given to the next process in the ready queue.
- The suspended process is placed at the back of the queue and gets another chance when its turn comes up again.
- **Context Switching:**
  - A shorter time quantum provides better response times but increases context switching overhead.
  - A longer time quantum reduces context switching but may lead to higher waiting times.

Contd..

- Advantages
  - **Fairness:** By giving each process an equal share of the CPU over time
  - **Simple Implementation:** The algorithm is simple to implement and manage
  - **Prevents Starvation:** Since processes are given fixed time slices, no process can monopolize the CPU for an extended period
- Drawbacks
  - **High Waiting Times: I** n scenarios with long time quantums, processes may experience high waiting times.
  - **Inefficiency for Short Processes:** Short processes may have to wait for their turn, leading to inefficiency.
  - **Low Throughput:** Round Robin may have lower throughput compared to some other scheduling algorithms.
  - **Context Switching Overhead:** Frequent context switching can introduce overhead, impacting system performance.

Multi-Level Queue Scheduling

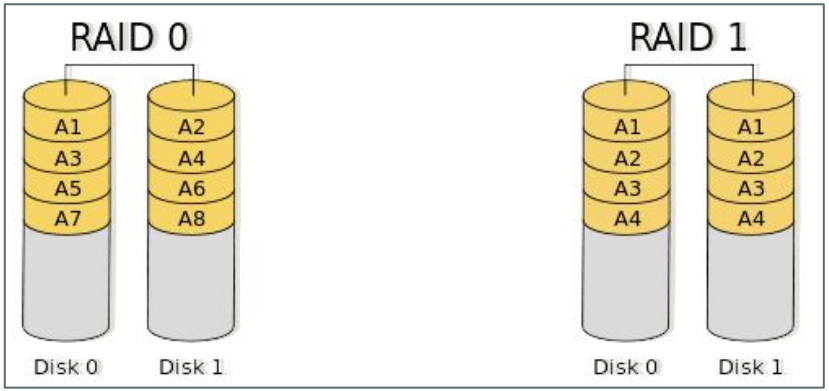
- It partitions the ready queue into multiple queues, each with its priority level.
- Processes are assigned to different priority levels based on their characteristics, and each queue may have its own scheduling algorithm.
- Processes are scheduled in a way that higher-priority queues have precedence over lower-priority ones.
- **Aging:**
  - To prevent starvation of processes in lower-priority queues.
  - If a process remains in a lower-priority queue for an extended period, its priority may be gradually increased.
- Drawbacks
  - **Complexity:** Implementing and managing multiple queues with different scheduling algorithms can add complexity to the scheduling algorithm.

What is RAID?

- RAID: **Redundant Array of Independent Disks**
- It is a storage technology that combines multiple physical disk drives into a single logical unit for the purpose of data redundancy, performance improvement, or both.
- It playes a significant role in narrowing the gap between increasingly fast processors and slow disk drives.

Enumerate the different RAID levels

- **RAID 0 (Striping):**
  - Data is distributed across multiple disks without redundancy.
  - RAID 0 improves performance by allowing parallel data access but provides **no** fault tolerance. If one disk fails, data loss occurs.
- **RAID 1 (Mirroring):**
  - Data is mirrored across pairs of drives. That means you still can restore the original data by using the mirrored disk.
  - This provides redundancy, as the same data is stored on two or more disks.
  - RAID 1 improves fault tolerance, but storage capacity is reduced since each data block is duplicated.



Contd..

- **RAID 5 (Striping with Parity):**
  - Data is striped across multiple disks, and parity information is distributed across all disks.
  - RAID 5 offers fault tolerance, as the failure of a single disk can be recovered using parity information.
  - It provides a good balance between performance and redundancy.
- **RAID 6 (Striping with Dual Parity):**
  - Similar to RAID 5 but with dual parity.
  - RAID 6 can withstand the failure of two disks simultaneously, providing higher fault tolerance compared to RAID 5.

What is the need for Logical Address Space?

- **Abstraction of Memory:** Provides an abstraction layer that allows programs to operate without needing knowledge of the physical organization of memory. This abstraction simplifies programming, as developers can work with a consistent and contiguous address space.
- **Isolation of Processes:** Enables the isolation of processes. Each process has its own logical address space, preventing interference with other processes. This isolation contributes to the stability and security of the system.
- **Virtual Memory Systems:** Logical addresses are essential for the implementation of virtual memory systems. In virtual memory, the logical address space can be larger than the physical address space, and the operating system handles the mapping between logical and physical addresses.

Logical Address Space VS Physical Address Space

- Logical Address Space
  - The logical address space refers to the range of addresses that a process can use to reference its memory.
  - These addresses are generated by the CPU during program execution and represent the program's view of memory.
  - The logical address space is typically contiguous and starts at zero.
  - Each process has its own logical address space, allowing it to operate independently of other processes.
- Physical Address Space
  - The physical address space represents the actual locations in the computer's physical memory (RAM) where data is stored.
  - Unlike the logical address space, the physical address space is not necessarily contiguous.
  - The translation from logical addresses to physical addresses is done by the memory management unit (MMU), a hardware component that works in conjunction with the operating system.

Contd..

	Logical Address Space	Physical Address Space
	Set of all logical addresses generated by the CPU in reference to a program is referred to as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred to as a Physical Address.
Basic	Virtual address generated by CPU.	A location in a memory unit.
Contiguity	Typically contiguous	Not necessarily contiguous
Generated by	CPU during program execution	MMU
Independence	Each process has its own logical address space	Shared among processes, isolated by the MMU
Access	The user uses the logical address to access the physical address.	The user can not directly access the physical address
Representation	Represents the program's view of memory	Represents the actual locations in physical memory

What is Dynamic Loading?

- Dynamic loading allows a program to load and execute a module (portion of code) into memory at runtime, rather than loading the entire program into memory when it starts.
- In dynamic loading, modules are loaded into memory only when they are explicitly called by the executing program.
- Dynamic loading allows for more efficient memory usage because only the necessary modules are loaded, conserving memory resources.
- Programs that use dynamic loading typically have faster start-up times since only essential components are loaded initially.
- Dynamic loading is often associated with the use of **dynamic link libraries (DLLs)** on Windows or shared libraries on Unix/Linux systems. These libraries contain code and data that can be shared among multiple programs.

How Does Dynamic Loading Aid in Better Memory Utilization?

- With dynamic loading, a routine is not loaded until it is called. This method is especially useful when large amounts of code are needed in order to handle infrequently occurring cases such as error routines.

What is the Basic Function of Paging?

- In paging, the logical address space of a process is divided into fixed-size blocks called pages, and the physical memory is divided into blocks of the same size called frames.
- The operating system manages the mapping of pages to frames, allowing processes to be loaded and executed in non-contiguous memory locations.

Key Concepts of Paging

- **Page Table:** The page table is a data structure maintained by the operating system that keeps track of the mapping between logical pages and physical frames. Each entry in the page table corresponds to a page and contains the frame number where the page is stored in physical memory.
- **Page Size:** Page size is the fixed-size unit into which the logical address space is divided. Common page sizes include 4 KB, 8 KB, or 16 KB. The choice of page size is a trade-off between internal and external fragmentation.
- **Frame Size:** Frame size is the fixed-size unit into which physical memory is divided. It corresponds to the size of a page. The size of a frame is the same as the size of a page to facilitate one-to-one mapping between logical pages and physical frames.
- **Address Translation:** When a program generates a logical address, the operating system translates this address using the page table to determine the corresponding physical address. This translation allows the program to access the correct location in physical memory.
- **Page Fault:** A page fault occurs when a program tries to access a page that is not currently in physical memory. The operating system handles page faults by loading the required page from secondary storage (such as a hard disk) into an available frame in physical memory.
- **Page Replacement:** In the event of a page fault and no available frames, the operating system must select a page to be replaced. Various page replacement algorithms, such as Least Recently Used (LRU) or First-In-First-Out (FIFO), determine which page to swap out.

What is Segmentation?

- To divide a process's logical address space into segments, each representing a different part of the program's memory.

Paging Vs Segmentation

	Paging	Segmentation
	Fixed-sized pages	Variable-sized segments
	logical address is split into that page number and page offset.	logical address is split into section number and section offset.
Address Translation	Singe step process: Page number used to index page table,which provides mapping to physical frame	Two step process: Segment number used to locate segment in table entry, then offset used to locate the desired data within segment
Fragmentation	Results internal fragmentation	Results external fragmentation

What is Swapping?

- Swapping is a memory management technique that involves moving a process or a part of a process from main memory (RAM) to secondary storage (usually a hard disk) and vice versa. The primary purpose of swapping is to improve memory management and overall system performance.
- **Benefits**
  - Swapping allows the operating system to temporarily move less frequently used parts of processes (pages or segments) to secondary storage. Which enables multiprogramming and multitasking.
  - Processes can be swapped out when they are waiting for external events (e.g., user input), allowing other active processes to use the available memory.
  - wapping is particularly useful for handling large processes that may not fit entirely in RAM. The operating system can selectively swap in and out portions of a process as needed.
- Excessive swapping can lead to performance degradation.
- Excessive swapping, known as thrashing, occurs when the system spends more time moving processes in and out of memory than executing actual instructions.

Name Classic Synchronization Problems

- Synchronization problems arise when multiple threads or processes access share resources concurrently, leading to unexpected or undesirable outcomes.
- Classic Synchronization Problem
    - **Race Conditions:** When the outcome of a program depends on the relative timing or interleaving of multiple threads or processes. If the execution order is not controlled, it can lead to unexpected behavior and data corruption.
    - **Data Inconsistency:** When multiple threads or processes modify shared data concurrently without proper synchronization, data inconsistency can occur.
    - **Deadlocks:** Deadlocks occur when two or more threads or processes are blocked indefinitely, each waiting for the other to release a resource.
    - **Starvation:** Starvation happens when a thread or process is unable to access a shared resource it needs for an extended period, leading to its inability to make progress. This can occur if other threads continuously acquire the resource.

Contd..

- **Producer-Consumer Problem:** Multiple threads or processes must coordinate the production and consumption of data in a shared buffer.
- **Readers-Writers Problem:** Managing access to shared data by multiple readers and writers.
- **Dining Philosophers Problem:** Involves multiple philosophers sitting around a dining table. Each philosopher alternates between thinking and eating, but they need to pick up two forks to eat. Without proper synchronization, deadlocks can occur.
- **Bounded-Buffer Problem:** Producers and consumers share a fixed-size buffer. Without proper synchronization, issues like buffer overflow or underflow can arise, leading to data loss or blocking.

Deadlocks

- The conditions necessary for a deadlock to occur - the four Coffman conditions:
  - **Mutual Exclusion:** At least one resource must be held in a non-shareable mode. This means that only one process at a time can use the resource.
  - **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently held by other processes.
  - **No Preemption:** Resources cannot be forcibly taken away from a process; they must be explicitly released by the process holding them.
  - **Circular Wait:** A cycle must exist in the resource allocation graph, where each process in the cycle is waiting for a resource held by the next process in the cycle.
- Prevention:
  - Modify the system to ensure that at least one of the Coffman conditions is never true.

What is the Direct Access Method

- Direct access is a method of accessing data in a storage device where each data unit is assigned a unique identifier or address. This allows for the direct retrieval of data without the need to sequentially traverse the entire dataset.
- Direct access is commonly associated with random access memory (RAM) in computer systems and direct access storage devices like hard drives.
- Indexing is commonly used in direct access methods to maintain a data structure (index) that maps logical keys or identifiers to the corresponding physical addresses.

What is Caching?

- To store and manage copies of **frequently accessed** or recently used data in a location that allows for faster retrieval.
- The goal of caching is to improve system performance and response times by reducing the need to fetch the data from slower, primary storage locations.
- Key Concepts of Caching:
  - **Cache Memory:** The cache is typically a smaller, faster, and more expensive type of memory compared to the main memory (such as RAM or disk). It stores a subset of data that is frequently accessed or likely to be accessed soon.
  - **Cache Hit and Cache Miss:** A cache hit occurs when the data requested by the CPU is found in the cache, resulting in a faster retrieval. A cache miss occurs when the requested data is not in the cache, and the system needs to fetch it from a slower storage location.

What is Spooling?

- Spooling - **S**imultaneous **P**eripheral **O**perations **O**n-**L**ine
- Allows to manage I/O operations in a way that allows the system to overlap the processing of one task with the execution of another. This is particularly useful when dealing with devices that have varying speeds, such as a fast CPU and a slower peripheral device like a printer or disk drive.
- A spooling area, often located on a disk or other storage medium, is used to temporarily hold the data associated with the tasks.
- While the CPU is working on executing one task, the spooler system can simultaneously process another task in the background.
    - For example, if a user is printing a document, the CPU can continue processing other tasks while the printing operation occurs in the background.

What are Interrupts?

- An interrupt is a mechanism by which a device or a software component can request the attention of CPU.
- Interrupts are signals that temporarily halt the normal execution of a program to transfer control to a specific interrupt-handling routine. This allows the CPU to respond to external events or conditions promptly.
- Interrupts are essential for efficient multitasking, real-time processing, and device interaction in computer systems.
- Key features of interrupts include:
    - **Event-Driven:** Interrupts are triggered by specific events or conditions rather than being part of the regular program flow. Events may include hardware signals (e.g., keyboard input, timer expiration) or software-generated signals.
    - **Asynchronous:** Meaning they can occur at any time, independent of the current state of the CPU. The CPU must be prepared to handle interrupts at any point during its execution.
    - **Interrupt Vector:** Each type of interrupt is associated with a unique interrupt vector, which is an address pointing to the corresponding interrupt service routine (ISR) or handler. When an interrupt occurs, the CPU jumps to the specified address to execute the corresponding routine.
    - **Maskable and Non-Maskable:** Interrupts can be classified as maskable or non-maskable. Maskable interrupts can be temporarily disabled or masked by the CPU, while non-maskable interrupts cannot be disabled.
    - **Context Switching:** Interrupts play a crucial role in context switching, allowing the CPU to switch between different tasks or processes seamlessly.

What is IPC?

- IPC - Inter Process Communication
- Refers to the mechanisms and techniques used by various processes to communicate with each other.
- Processes in an operating system may run independently and need a way to exchange information, coordinate activities, or synchronize their execution.
- IPC provides a set of communication tools and methods for achieving these objectives.

What are different IPC Mechanisms?

1. **Message Passing:** Processes exchange messages with each other using a communication channel. Messages can be sent asynchronously or synchronously, and they may contain data or signals. This approach is common in distributed systems and can be implemented using various techniques, such as sockets or message queues.
2. **Shared Memory:** Processes can communicate by sharing a region of memory that is accessible to each participating process. This shared memory allows processes to read and write data directly into the shared space, providing a fast communication method. Proper synchronization mechanisms (e.g., semaphores or mutexes) are needed to avoid conflicts.
3. **Sockets:** Sockets are communication endpoints that processes can use to establish connections and exchange data over a network or between processes on the same machine.
4. **Pipes:** Pipes provide a unidirectional communication channel between two related processes. One process writes data to the pipe, and the other reads from it. Pipes are often used for communication between parent and child processes.
5. **Signals:** Signals are software interrupts sent by one process to another to notify it about specific events or conditions. While signals are lightweight, they are generally limited to simple notifications and lack the ability to carry extensive data.
6. **Semaphores:** Semaphores are synchronization objects used to control access to shared resources. They help coordinate multiple processes by providing a way to signal and wait for events or to control critical sections of code.
7. **Message Queues:** Message queues are mechanisms that allow processes to send and receive messages in the form of queues. Each message has a priority, and processes can retrieve messages in order of their priority.
8. **RPC (Remote Procedure Call):** RPC allows a process to execute procedures or functions in another address space, typically on a remote machine. It abstracts the complexity of network communication, making it appear as if the function is executed locally.

What are Semaphores?

- Semaphores are used to control access to shared resources and coordinate the execution of processes.
- A semaphore is essentially a **variable** that can be accessed by multiple processes or threads.
- Semaphores can be of two types:
  - Binary semaphores
  - Counting semaphores

Binary Semaphores

- Binary semaphores, also known as mutexes, have two states: 0 and 1.
- They are often used to protect critical sections of code, ensuring that only one process or thread can access a particular resource at a time.
- **Operations on Binary Semaphores:**
  - wait() or P():
    - If the semaphore value is 1, decrement it to 0 and proceed.
    - If the value is 0, the process or thread is blocked until the semaphore becomes 1.
  - signal() or V():
    - Increment the semaphore value by 1.
    - If there are blocked processes or threads waiting, wake up one of them.

Counting Semaphores

- Counting semaphores can have an integer value greater than 1.
- They are used to control access to a pool of identical resources, allowing a specified number of processes or threads to access the resource simultaneously.
- **Operations on Counting Semaphores:**
  - wait() or P():
    - If the semaphore value is greater than 0, decrement it.
    - If the value is 0, the process or thread is blocked until the semaphore becomes greater than 0.
  - signal() or V():
    - Increment the semaphore value by 1.
    - If there are blocked processes or threads waiting, wake up one or more of them.

What are the advantages of Semaphores?

- They are machine-independent.
- Easy to implement.
- Correctness is easy to determine.
- Semaphores allow the management of many different critical sections by using different semaphores
- Semaphores acquire many resources simultaneously.
- Semaphores prevent resource waste associated with busy waiting, contributing to resource efficiency
    - **Busy waiting** : A process or thread repeatedly checks for a condition to be satisfied in a loop without relinquishing the CPU. Instead of waiting passively, the process actively consumes CPU cycles by continuously checking the condition.

What is a Bootstrap Program in OS?

- A special program that initializes the operating system (OS) during the system's startup process.
- The primary function of the bootstrap program is to load the operating system into the computer's main memory (RAM) and to transfer control to the starting address of the OS.

Preemptive Scheduling VS Non-Preemptive scheduling

Preemptive Scheduling

- Allows the operating system to interrupt a currently running process and start or resume another process.
- Higher context-switching overhead due to frequent switches between processes.
- If a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve.
- Commonly used in real-time systems where tasks must be executed within specific time constraints.
- Example: Round Robin, Priority Scheduling, Multilevel Queue Scheduling.

Non-Preemptive Scheduling

- Does not allow interruption of a running process until it voluntarily gives up control or completes its execution.
- Lower context-switching overhead, as switches occur only when a process completes its execution.
- If CPU is allocated to the process having a larger burst time then the processes with a small burst time may have to starve.
- Less suitable for real-time systems as response times can be less predictable.
- Example: First-Come-First-Serve (FCFS), Shortest Job Next (SJN), Priority Scheduling (without preemption).

What ia a Zombie Process?

- A state assigned to a child process after it has terminated, but its parent process has not yet retrieved its exit status.
  - The primary reason for a parent process not collecting the exit status of its child process could be that the parent process is busy with other tasks or has terminated unexpectedly.
- A zombie process is a process that has completed its execution but still has an entry in the process table.
- Here's how the life cycle of a typical process works:
  - A parent process creates a child process.
  - The child process executes its task.
  - Once the child process completes its execution, it becomes a "zombie" if the parent process does not collect its exit status.
- Zombie processes pose no harm to the system's functionality, but they consume system resources (such as an entry in the process table) until the parent process collects their exit status.
- To resolve the issue of zombie processes, the parent process should use the wait() system call to retrieve the exit status of its child processes.

What is an Orphan Process?

- An orphan process is a process whose parent process has terminated or completed before the child process finishes its execution.
- Here's how the life cycle of an orphan process typically unfolds:
  - A parent process creates a child process.
  - The parent process terminates or completes its execution before the child process.
  - The child process, now detached from its original parent, gets adopted by the init process (or another system-level process).
- The init process, which has a process ID (PID) of 1, is the ancestor of all processes in a Unix-like operating system. It automatically adopts orphan processes, ensuring that they have a parent process for the remainder of their execution.

Starvation & Aging

- **Starvation**
  - Starvation is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes.
- **Aging**
  - Aging is a technique to avoid starvation in a scheduling system.
  - It involves gradually increasing the priority of a process over time, making it more likely to be selected for execution as it waits in the ready queue. Aging ensures that processes that have been waiting for a long time eventually receive the opportunity to run, even if they have lower initial priorities.

Operating System VS Kernel

Operating System

- An interface b/w the user and the hardware.
- It is the first program to load when the computer boots up.
- manages hardware resources, provides user interfaces, and supports applications.
- Provides user interfaces (UI) and applications, allowing users to interact with the system.

Kernel

- An interface b/w the application and hardware.
- It is the first program to load when the operating system loads
- Core component of the operating system responsible for managing system resources and providing essential services.
- Does not directly handle user interfaces; its functions are critical for the execution of OS services.

What is a Critical Section?

- A critical section is a specific part of a program where shared resources are accessed and manipulated by multiple threads or processes.
- In a critical section, only one thread or process is allowed to execute at a time to prevent data corruption or race conditions, ensuring the integrity of shared resources.

What is bounded waiting?

- Bounded waiting is a property in synchronization algorithms that ensures that every process or thread attempting to enter a critical section will do so within a certain bounded or limited number of attempts.
    - In other words, no process should be unfairly kept waiting indefinitely, preventing it from accessing a shared resource.
- Without bounded waiting, a process could potentially be starved indefinitely, causing a lack of progress and violating fairness principles.

What are the solutions to Critical Section problem?

1. **Hardware Solutions:**
    - Atomic Instructions
        - Hardware instructions that perform operations atomically without interruption.
        - Example: Compare-and-swap (CAS) instruction
    - Test-and-Set Instruction
        - Test-and-set instruction is used to implement locks. If the lock is not set, a thread sets it and enters the critical section.
2. **Software Solutions:**
    - Mutex (Mutual Exclusion)
    - Semaphores
    - Condition Variables
    - Spinlocks
    - Read-Write Locks
    - Transaction Memory

Talk about different synchronization Techniques

1. **Mutex (Mutual Exclusion):**
   - **Description:** It ensures that only one thread at a time can access a shared resource.
   - **Usage:** Threads acquire the mutex before entering the critical section and release it when they are done. This prevents multiple threads from concurrently executing code that modifies shared data.
   - **Example:** In POSIX threads, a mutex can be implemented using pthread_mutex_lock and pthread_mutex_unlock functions.
2. **Semaphore:**
   - **Description:** It controls access to a shared resource by maintaining a count. It can be used to limit the number of threads entering a critical section simultaneously.
   - **Usage:** Threads wait on the semaphore using sem_wait and release it using sem_post. The semaphore count represents the number of available slots for threads.
   - **Example:** In POSIX threads, semaphores can be created and manipulated using sem_init, sem_wait, and sem_post.

Contd...

3. **Conndition Variables:**
    - **Description:** Used for signaling and waiting on specific conditions in a program. They are often used in conjunction with mutexes to control the execution flow.
    - **Usage:** Threads can wait on a condition variable using pthread_cond_wait and signal others using pthread_cond_signal or pthread_cond_broadcast. Typically used to coordinate activities between threads.
    - **Example:** In POSIX threads, condition variables are associated with a mutex, and threads use the associated mutex to wait or signal.
4. **File Locks:**
    - **Description:** File locks are mechanisms to control access to files in a shared environment. They prevent multiple processes or threads from simultaneously modifying or reading a file.
    - **Usage:** File locks can be advisory or mandatory. Advisory locks are set voluntarily by programs, while mandatory locks are enforced by the operating system. Functions like fcntl in Unix systems are used for file locking.
    - **Example:** In Unix, the fcntl system call is commonly used for file locking. Advisory locks can be set using F_SETLK or F_SETLKW commands.

User level Threads vs Kernel Level Threads

User-Level-Thread

- User threads are implemented by users.
- OS doesn't recognize user-level threads.
- Implementation of User threads is easy.
- Context switch time is less.
- Context switch requires no hardware support.
- If one user-level thread performs a blocking operation then entire process will be blocked.
- User-level threads are designed as dependent threads.

Kernel-Level-Thread

- Kernel threads are implemented by OS.
- Kernel threads are recognized by OS.
- Implementation of kernel thread is complicated.
- Context switch time is more.
- Hardware support is needed.
- If one kernel thread perform a the blocking operation then another thread can continue execution.
- Kernel level threads are designed as independent threads.

Peterson’s Algorithm

- Peterson's Algorithm is a simple and classic software-based solution to the **mutual exclusion** problem.
- It is designed for **two** processes that share a common, single-use resource and need to ensure exclusive access to it.
- The algorithm uses shared variables and atomic operations to achieve mutual exclusion.
- Shared Variables:
  - int **turn**: Indicates whose turn it is to enter the critical section. It alternates between the two processes.
  - bool **flag[2]** : An array of flags, one for each process, indicating their interest in entering the critical section.

Contd...

```
int turn = 0;
bool flag[2] = {false};
void enter_critical_section(int thread_id) {
    int other = 1 - thread_id;
    // Declare interest in entering critical section
    flag[thread_id] = true;
        // Set turn to the other process
    turn = other;
    // Wait while the other process is in the critical section
    while (flag[other] && turn == other);
    // Critical Section
    printf("Thread %d is in the critical section.\n", thread_id);
    // Release interest in entering critical section
    flag[thread_id] = false;
}
```

Contd...

- **Characteristics:**
  - **Mutual Exclusion** : Ensures that only one process can be in the critical section at a time.
  - **Deadlock-Free** : If one process is stuck in the waiting loop, the other process will eventually release the lock, allowing progress.
  - **Starvation-Free:** Both processes have equal chances of entering the critical section.
- **Limitations:**
  - Peterson's Algorithm is specifically designed for two processes and may need modification to handle more than two.
  - It relies on the assumption that each process executes its critical section quickly, as busy-waiting is involved.

What is Address Binding?

- Address binding is a process in which symbolic addresses in a program are bound to physical memory addresses during the various stages of program execution.
- There are mainly three types of address binding:
  - Compile Time (Static Binding)
  - Load Time (Dynamic Binding)
  - Execution Time (Dynamic/Runtime Binding)

Contd...

Compile Time (Static Binding)

- Address binding is performed at compile time.
- The physical memory addresses are generated by the compiler and embedded directly into the executable code.
- Advantages:
  - Fast execution, as there is no need for additional address resolution at runtime.
  - No runtime overhead
- Disadvantages:
  - Lack of flexibility, as the program's location in memory is fixed at compile time.
  - Programs may need recompilation if they need to be loaded at different memory locations.

Contd...

Load Time (Dynamic Binding)

- Address binding is deferred until load time.
- The compiler generates symbolic addresses, and the linker generates the final physical memory addresses during the program's loading phase.
- Advantages:
  - Programs can be loaded at different memory locations without recompilation.
  - More flexibility than compile-time binding.
- Disadvantages:
  - Still requires recompilation for changes in memory location.
  - Slightly slower execution compared to compile-time binding due to the additional address resolution at load time.

Contd...

Execution Time (Dynamic/Runtime Binding)

- Address binding is performed during the program's execution.
- The program contains symbolic addresses, and the final binding to physical addresses happens dynamically during runtime.
- Advantages:
  - Maximum flexibility, as the program can adapt to changes in memory layout during runtime.
  - No need for recompilation or relinking for changes in memory location.
- Disadvantages:
  - Slightly slower execution due to runtime address resolution.
  - Increased runtime overhead for address binding.

What is Dynamic Allocation?

- Dynamic allocation refers to the process of allocating memory or other resources during program execution. This typically involves requesting memory from the system's heap at runtime to accommodate data structures or objects whose size cannot be determined at compile time.
- Dynamic loading and dynamic allocation are different concepts.
    - Dynamic loading pertains to the loading of routines or libraries into memory as needed during program execution, while Dynamic allocation involves the allocation of memory or resources at runtime

Advantages of Dynamic Allocation

- When we do not know how much amount of memory would be needed for the program beforehand.
- When we want data structures(i.e linked lists, structures) without any upper limit of memory space.
- When you want to use your memory space more efficiently.
- Dynamically created lists insertions and deletions can be done very easily just by the manipulation of addresses whereas in the case of statically allocated memory insertions and deletions lead to more movements and wastage of memory.

What is Compaction?

- The process of collecting fragments of available memory space into contiguous blocks by moving programs and data in a computer's memory or disk.

What is Locality of Reference?

- It is the tendency of programs to access memory locations that are close to each other or that have been recently accessed.
- It refers to the observation that programs often exhibit spatial and temporal locality,
  - **Spatial Locality**
    - Spatial locality refers to the tendency of programs to access memory locations that are near each other in memory.
    - Such as iterating over an array or traversing a data structure.
  - **Temporal Locality**
    - Temporal locality refers to the tendency of programs to access the same memory locations repeatedly over short periods of time.
    - Such as accessing frequently used variables in the program

What is file system?

- A file system is a method used by operating systems to organize and store data on storage devices such as hard drives, solid-state drives (SSDs), and flash drives.
- It provides a structured way to store, retrieve, and manage files and directories.
- The file system manages the physical location of data on the storage device and provides an interface for users and applications to interact with files and directories.

Contd...

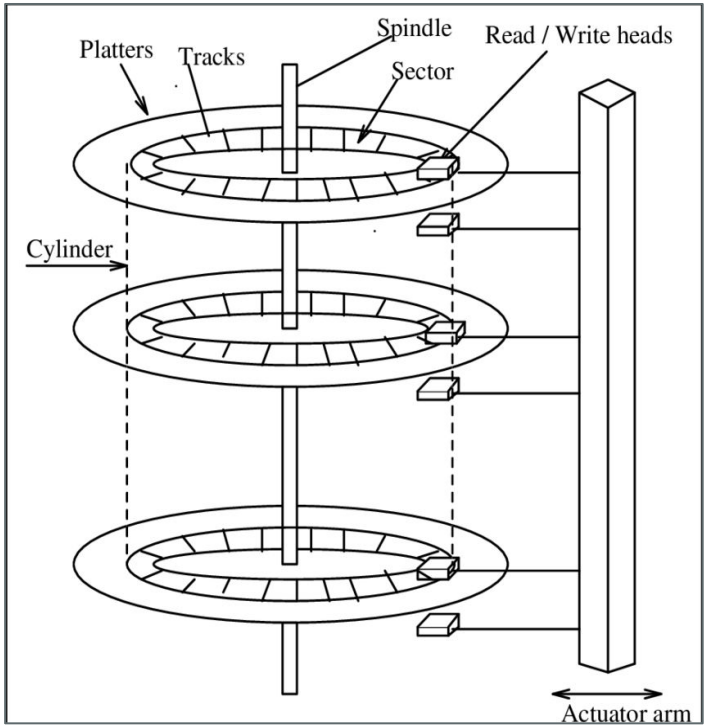
- **File:**
  - A file is a collection of related data or information stored on a storage device. Files can contain text, documents, images, videos, programs, and other types of data.
  - Each file is identified by a unique name and can have associated metadata, such as file size, permissions, creation date, and modification date.
- **Directory (Folder):**
  - A directory, also known as a folder, is a container used to organize and group related files and subdirectories.
  - Directories can be nested within other directories to create a hierarchical structure, allowing for logical organization of files and folders.
- **Path:**
  - A path is a unique identifier that specifies the location of a file or directory within the file system hierarchy.
  - Paths can be absolute or relative. An absolute path specifies the full location of a file or directory from the root of the file system, while a relative path specifies the location relative to the current directory.
- **File System Hierarchy:**
  - A file system organizes files and directories in a hierarchical structure, typically starting from a root directory.
  - The hierarchy may include multiple levels of directories, with each directory containing files and subdirectories.

Contd...

- **File Operations:**
  - File systems support various operations for managing files and directories, including creating, deleting, renaming, moving, copying, reading, and writing files.
  - These operations are performed through system calls or file system APIs provided by the operating system

Rotational Delay & Seek Time

- **Seek Time**
  - Refers to the time it takes for the disk's read/write head to move to the correct track on the disk where the desired data is located.
- **Rotational Delay**
  - Refers to the time it takes for the desired sector of the disk to rotate under the read/write head after the head has been positioned over the correct track.



What is a bit vector?

- It is a data structure that represents a fixed-size sequence of bits, typically implemented as an array.
- Each bit in the bit vector corresponds to a specific position - a disk block, allowing for efficient storage.
- The bit can take two values - 0 and 1:
  - 0 indicates that the block is allocated
  - 1 indicates a free block.

What is Belady's Anomaly?

- It describes a situation where increasing the number of page frames (available memory) can lead to an increase in page faults, rather than reducing them, as expected.
- This anomaly challenges the common intuition that more available memory should always result in better performance by allowing more pages to be kept in memory.
- **Example** : When eviction of pages in FIFO results in pages that are frequently accessed being removed from memory, leading to more frequent page faults even with additional memory.

What is a Non-Recursive Mutex?

- Recursive mutexes
  - Allows the **same** thread to acquire the mutex multiple times without deadlock
- Non-recursive mutexes
  - Only permits one lock acquisition per thread.

What happens if a non-recursive mutex is locked more than once?

- If a non-recursive mutex is locked more than once by the same thread, it typically leads to a **deadlock** situation.
- Here's what happens step-by-step:
  - **First Lock Attempt:**
    - When a thread attempts to acquire the non-recursive mutex for the first time, it successfully acquires the mutex, and the lock count is incremented.
  - **Subsequent Lock Attempts:**
    - If the same thread attempts to acquire the non-recursive mutex again while it is already held by the thread, the thread will become blocked.
  - **Deadlock:**
    - Since the mutex is already locked by the same thread, and the thread is waiting for itself to release the mutex, a deadlock occurs.
    - The thread will remain blocked indefinitely, unable to proceed with execution, and the program may become unresponsive.
- Solution: An operating system implementer can exercise care in identifying the owner of the mutex and return it if it is already locked by the same thread to prevent deadlocks.

How to recover from a deadlock?

- Recovering from deadlock typically involves breaking the circular wait condition by
  a. Either **Terminating one or more processes** involved in the deadlock
  b. Or **Preempting resources** held by one of the processes.

Contd...

- **Terminating one or more processes**
  - Detect the deadlock by analyzing the system's state, such as resource allocation graphs, process wait-for graphs.
  - Choose one or more processes involved in the deadlock for termination. This selection can be based on factors such as process priority, resource usage, or other criteria.
  - Terminate the selected processes. This releases the resources held by the terminated processes.
  - After terminating the processes, the system may need to **roll back** or undo any operations performed by the terminated processes to restore consistency and integrity.

Contd...

- **Preempting Resources**
  - Identify the resource(s) held by one of the processes involved in the deadlock that can be preempted without causing inconsistency or violating system integrity.
    - This involves forcibly taking back the resources from the process, which may require rolling back or undoing any operations performed using those resources.
  - After preemption, allocate the preempted resources to other processes waiting for them.