Operating System in One Video

▼ 1. What is an Operating System & Types of OS

An operating system (OS) is software that acts as an interface between computer hardware and user applications. It manages the resources and provides services for the efficient and secure execution of programs. The primary functions of an operating system include process management, memory management, file system management, device management, and user interface.

There are several types of operating systems, including:

1. **Windows**: Developed by Microsoft, Windows is a widely used operating system for personal computers. It offers a user-friendly interface, supports a vast range of software applications, and is compatible with various hardware configurations.
2. **macOS**: Developed by Apple, macOS is the operating system used on Apple Mac computers. It provides a sleek and intuitive user interface, seamless integration with other Apple devices, and a robust ecosystem of software applications.
3. **Linux**: Linux is an open-source operating system that is highly customizable and widely used in server environments and embedded systems. It offers a

high level of stability, security, and flexibility. Numerous distributions of Linux, such as Ubuntu, Fedora, and CentOS, cater to different user needs.

4. **Unix**: Unix is a powerful multi-user operating system that served as the foundation for many other operating systems, including Linux and macOS. It provides a stable and secure environment and is famous for server applications.
5. **Android**: Developed by Google, Android is an open-source operating system primarily designed for mobile devices such as smartphones and tablets. It offers a rich ecosystem of applications and customization options.
6. **iOS**: Developed by Apple, iOS is the operating system used on iPhones, iPads, and iPods. It provides a seamless and secure user experience, focusing on performance and integration with other Apple devices.
7. **Real-Time Operating Systems (RTOS)**: RTOS is designed for systems that require deterministic and real-time response. It is commonly used in embedded systems, control systems, and IoT devices.

▼ 2. Difference between Multiprogramming, Multiprocess, Multitasking, and Multithreading

1. **Multiprocess**:
   Multiprocess refers to the execution of multiple processes on a system with multiple CPUs or CPU cores. Each process is an instance of a running program, and multiple processes can execute concurrently. In multiprocess systems, each process has its own memory space and resources. Multiprocessing aims to increase system throughput and provide faster execution by distributing the workload across multiple processors.
2. **Multithreading**:
   Multithreading involves executing multiple threads within a single process. A thread is a lightweight unit of execution that can run concurrently with other threads within the same process. Threads share the same memory space and resources, such as file handles and network connections. Multithreading allows for parallel execution within a process, enabling better utilization of system resources and potentially improving performance by dividing tasks into smaller units of work that can be executed concurrently.
3. **Multiprogramming**:
   Multiprogramming is a technique where multiple programs are loaded into

memory simultaneously, and the CPU switches between them to execute instructions. The purpose of multiprogramming is to maximize CPU utilization and keep the CPU busy by quickly switching between different programs when one is waiting for I/O or other operations. Each program has its own separate memory space.

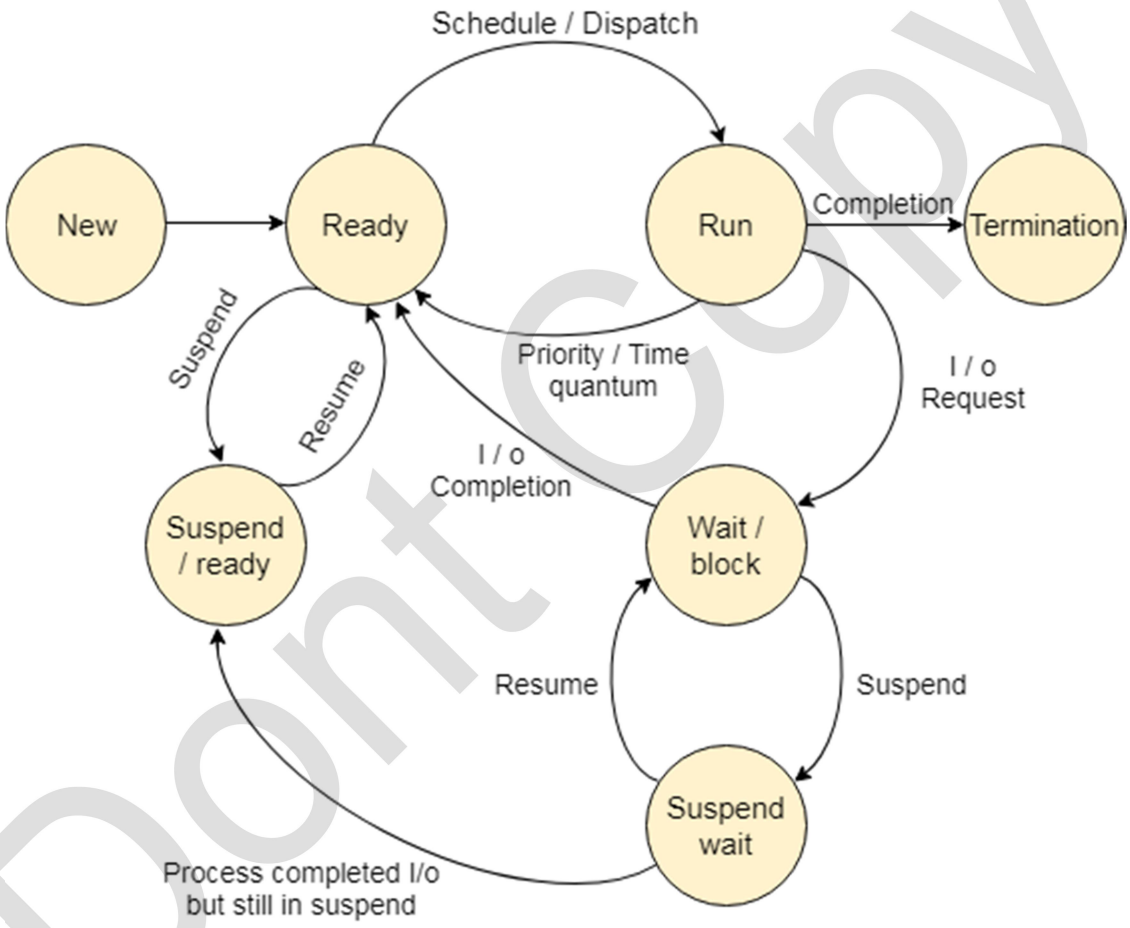
4. **Multitasking**:
   Multitasking is a technique that allows multiple tasks or processes to run concurrently on a single CPU. The CPU time is divided among the tasks, giving the illusion of parallel execution. The operating system switches between tasks rapidly, giving each task a time slice or quantum to execute. Multitasking is commonly used in modern operating systems to provide responsiveness and the ability to run multiple applications simultaneously.

▼ 3. Process vs Threads vs Programs

- **Program**:
  A program is a set of instructions written in a programming language that performs a specific task or set of tasks. It is typically stored in a file on disk and represents an executable entity. Programs can be compiled or interpreted, and they serve as a blueprint for the execution of tasks on a computer system.
- **Process**:
  A process is an instance of a program in execution. When a program is loaded into memory and executed, it becomes a process. A process is an independent entity with its own memory space, resources, and execution context. It has its own program counter, stack, and variables. Processes are managed by the operating system, and each process runs in its own protected memory space. Processes can be concurrent and communicate with each other through inter-process communication mechanisms.
- **Thread**:
  A thread is a unit of execution within a process. It represents a sequence of instructions that can be scheduled and executed independently. Threads share the same memory space and resources within a process. Multiple threads within a process can run concurrently, allowing for parallel execution of tasks. Threads within the same process can communicate and share data more easily compared to inter-process communication. However, each thread has its own program counter and stack.

In summary, a program is a set of instructions, a process is an instance of a program in execution with its own resources and memory space, and a thread is a unit of execution within a process that allows for the concurrent execution of tasks. Processes provide isolation and protection between different instances of a program, while threads within a process share resource and enable parallel execution of tasks.

▼ 4. Various Process States



▼ 5. CPU scheduling Algorithms

CPU scheduling algorithms are used by the operating system to determine the order in which processes are executed on the CPU.

1. **First-Come, First-Served (FCFS):**
   In the FCFS scheduling algorithm, the process that arrives first is executed first. It follows a non-preemptive approach, meaning that once a process

starts running, it continues until it completes or voluntarily gives up the CPU. FCFS is simple to understand but may lead to poor utilization of the CPU if long processes arrive before shorter ones.

2. **Shortest Job Next (SJN) or Shortest Job First (SJF):**
   SJN or SJF scheduling selects the process with the shortest total execution time next. It can be either non-preemptive or preemptive. In the non-preemptive variant, the process continues executing until it completes, whereas in the preemptive variant, if a new process with a shorter burst time arrives, the currently running process may be preempted. SJN/SJF aims to minimize the average waiting time and is suitable when burst times are known in advance.
3. **Round Robin (RR):**
   Round Robin is a preemptive scheduling algorithm that assigns a fixed time quantum (e.g., 10 milliseconds) to each process in a circular manner. Once a process exhausts its time quantum, it is moved to the back of the ready queue, allowing the next process in line to execute. RR provides fair execution to all processes but may suffer from high context-switching overhead and may not be efficient for long-running processes.
4. **Priority Scheduling:**
   Priority scheduling assigns a priority value to each process, and the CPU is allocated to the process with the highest priority. It can be either preemptive or non-preemptive. In preemptive priority scheduling, if a higher-priority process arrives, the currently running process may be preempted. In non-preemptive priority scheduling, the process continues executing until it completes or voluntarily gives up the CPU. Priority scheduling can suffer from starvation if a lower-priority process never gets a chance to execute.
5. **Multilevel Queue Scheduling:**
   Multilevel queue scheduling divides the ready queue into multiple priority queues, each with its own scheduling algorithm. Processes are initially placed in the highest-priority queue and can move between queues based on predefined criteria. This approach allows for the differentiation of processes based on their priority or characteristics, such as foreground or background tasks. Each queue can use a different scheduling algorithm, such as FCFS, SJF, or RR, suitable for the processes within that queue.

▼ 6. Critical section Problem

The critical section represents a portion of code or a block where a process or thread accesses a shared resource, such as a variable, file, or database. It's a section that needs to be executed in an atomic or mutually exclusive manner.

To maintain data integrity and avoid conflicts, only one process or thread should be allowed to enter the critical section at a time. This ensures that no two processes interfere with each other and modify shared resources simultaneously, preventing inconsistencies or incorrect results.

▼ 7. Process synchronisation

Process synchronization is like a traffic signal that helps regulate the flow of vehicles at an intersection. In the context of computing, it refers to techniques and mechanisms used to coordinate the execution of processes or threads so that they can work together harmoniously.

Imagine multiple processes or threads working on different tasks simultaneously. Process synchronization ensures that they cooperate and communicate effectively to avoid conflicts and ensure proper order of execution. It helps prevent issues like race conditions, data inconsistencies, or deadlocks that can arise when multiple processes or threads access shared resources simultaneously.

Here are the key requirements of synchronization mechanisms:

1. **Mutual Exclusion**: The synchronization mechanism should enforce mutual exclusion, which means that only one process or thread can access a shared resource or enter a critical section at a time. It ensures that concurrent access to shared resources does not result in conflicts or inconsistencies.
2. **Progress**: The synchronization mechanism should allow processes or threads to make progress by ensuring that at least one process/thread can enter the critical section when it desires to do so. It avoids situations where all processes/threads are blocked indefinitely, leading to a deadlock.
3. **Bounded Waiting**: The synchronization mechanism should provide a guarantee that a process/thread waiting to enter a critical section will eventually be allowed to do so. It prevents a process/thread from being starved or waiting indefinitely to access a shared resource

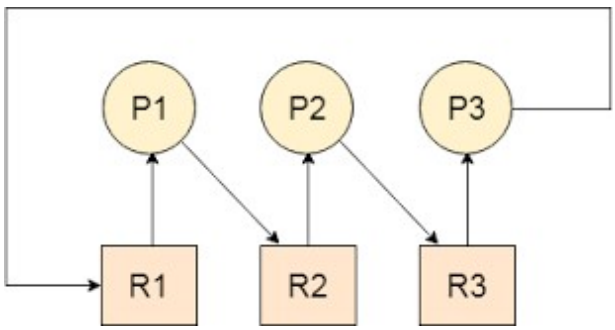
▼ 8. Process Synchronisation Mechanisms

Here are some of the mechanisms that fulfill the above process synchronization requirements:

1. **Locks/Mutexes**: Locks or mutexes (mutual exclusions) provide a simple and effective way to achieve mutual exclusion. They allow only one process or thread to acquire the lock at a time, ensuring exclusive access to a shared resource or critical section. Locks can be implemented using hardware instructions or software constructs.
2. **Semaphores**: Semaphores are synchronization objects that can be used to control access to shared resources. They can be implemented as binary semaphores (mutexes) or counting semaphores. Counting semaphores allow a specified number of processes or threads to access a shared resource simultaneously. Semaphores provide mechanisms for mutual exclusion, signaling, and coordination.
3. **Read-Write Locks**: Read-write locks provide synchronization mechanisms for scenarios where multiple readers can simultaneously access a shared resource without conflicts, but exclusive access is required for writers. Read locks can be acquired simultaneously by multiple readers, while write locks are exclusive. Read-write locks allow for better concurrency when reading is more frequent than writing.

▼ 9. Deadlock

A Deadlock is a situation where each of the computer processes waits for a resource that is being assigned to another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process that is also waiting for some other resource to be released.



Necessary Conditions for Deadlocks:

- **Mutual Exclusion**
  A resource can only be shared in a mutually exclusive manner. It implies that

two processes cannot use the same resource at the same time.

- **Hold and Wait**
  A process waits for some resources while holding another resource at the same time.
- **No preemption**
  The process once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.
- **Circular Wait**
  All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

▼ 10. Deadlock Handling Techniques

1. **Deadlock Prevention**: Deadlock prevention techniques aim to eliminate one or more of the necessary conditions for deadlock to occur. These conditions include mutual exclusion, hold and wait, no preemption, and circular wait. By ensuring that one or more of these conditions are not satisfied, deadlocks can be prevented from happening in the first place. However, prevention techniques may impose restrictions on resource allocation and may not be always feasible or efficient.
2. **Deadlock Avoidance**: Deadlock avoidance techniques use resource allocation algorithms and resource request protocols to avoid situations that may lead to deadlocks. These techniques involve the use of resource allocation graphs, bankers' algorithm, or other dynamic allocation strategies. The idea is to have a system that can predict whether granting a resource request will lead to a potential deadlock. If a request might cause a deadlock, it is delayed until granting it will not cause any issues.
3. **Deadlock Detection**: Deadlock detection techniques involve periodically examining the resource allocation state to determine if a deadlock has occurred. This can be achieved through algorithms such as the resource-allocation graph or the Banker's algorithm. When a deadlock is detected, the system can take appropriate actions to resolve it, such as terminating processes, resource preemption, or rolling back the system to a safe state.
4. **Deadlock Recovery**: Deadlock recovery techniques are used to recover from a deadlock once it has been detected. This involves terminating one or

more processes involved in the deadlock to break the circular wait and release the resources held by those processes. The terminated processes can be restarted later to continue their execution.

5. **Deadlock Ignorance**: Some systems choose to ignore the problem of deadlocks entirely. This approach assumes that deadlocks will rarely occur and focuses on other system aspects. However, it is generally not recommended unless deadlocks are extremely rare or have minimal impact on system functionality. Operating systems like Windows and Linux mainly focus on performance. However, the performance of the system decreases if it uses a deadlock handling mechanism all the time if a deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

▼ 11. Memory Management

Fixed partitioning and dynamic partitioning are two approaches used in memory management systems to allocate and manage memory resources. Let's discuss each approach:

1. **Fixed Partitioning:**
   In fixed partitioning, memory is divided into fixed-sized partitions or blocks, and each partition is assigned to a specific process or task. The system allocates a predetermined amount of memory to each partition, which remains fixed throughout the execution.

   Fixed partitioning is relatively simple to implement and provides fast memory allocation. However, it can lead to inefficient memory utilization due to internal fragmentation, especially when processes have varying memory requirements.

2. **Dynamic Partitioning:**
   Dynamic partitioning, also known as variable partitioning, addresses the limitation of fixed partitioning by allowing memory to be allocated and deallocated dynamically based on the size requirements of processes.

   Dynamic partitioning provides better memory utilization compared to fixed partitioning, as memory can be allocated based on actual requirements. However, managing fragmentation and efficiently allocating and deallocating memory can be more complex.

Memory management techniques, such as compaction or memory allocation algorithms like **first-fit, best-fit, or worst-fit,** are employed to optimize memory utilization and minimize fragmentation in dynamic partitioning systems.

▼ 12. Partition and Memory allocation

1. **First Fit**: The first-fit algorithm allocates the first available memory block that is large enough to accommodate the process. It starts searching from the beginning of the memory and selects the first suitable block encountered. This algorithm is simple and provides fast allocation, but it may result in relatively larger leftover fragments.
2. **Best Fit**: The best-fit algorithm searches for the smallest available memory block that is large enough to accommodate the process. It aims to minimize leftover fragments by choosing the most optimal block. This algorithm can lead to better overall memory utilization, but it may involve more time-consuming searches.
3. **Worst Fit**: The worst-fit algorithm allocates the largest available memory block to the process. This approach intentionally keeps larger fragments to accommodate potential future larger processes. While it may seem counterintuitive, it can help reduce fragmentation caused by small processes and improve overall memory utilization.

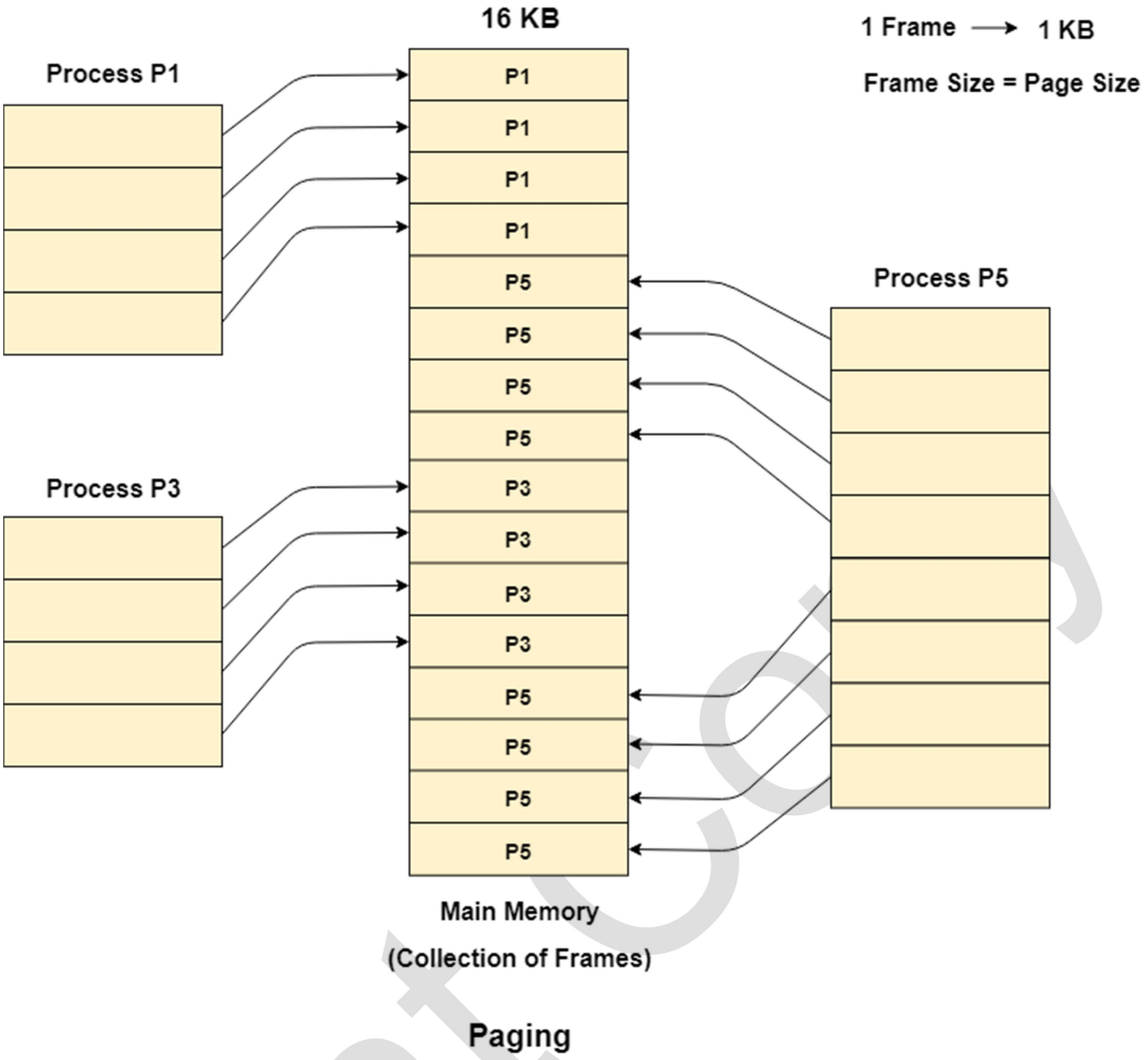
▼ 13. Paging

In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.

One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.



Paging

▼ 14. Virtual Memory

Virtual memory is a concept that lets a computer use more memory than it actually has. It creates an imaginary memory space by combining physical memory (RAM) and secondary storage (like a hard disk). When a program needs more memory than is available in RAM, it temporarily moves some data to the secondary storage. This allows the computer to run larger programs and handle multiple tasks at once. Virtual memory also provides memory protection, ensuring that programs can't access each other's memory directly.

Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory.

By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

▼ 15. Page replacement algorithms

1. **First-In-First-Out Page Replacement Algorithm**

   In FIFO page replacement, the operating system maintains a queue or list of pages in the order they were brought into memory. When a page fault happens (i.e., the required page is not in memory), the page at the front of the queue, which was the earliest one to be brought in, is selected for replacement. The new page is then brought into memory, and the page at the front of the queue is removed.

   6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0

S. no	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F3				2	2	2	4	4	4	2	2	2	2	2	2	2	2	2	2	0
F2		1	1	1	1	3	3	3	0	0	0	0	0	0	0	3	3	3	3	3
F1	6	6	6	6	0	0	0	6	6	6	1	1	1	1	1	1	1	1	1	1
Hit (H)/ Fault (F)	F	F	H	F	F	F	F	F	F	F	F	H	H	H	H	F	H	H	H	F

2. **Optimal Page replacement**

   In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

   In theory, optimal page replacement provides the best possible performance by minimizing the number of page faults. However, in practice, it is not feasible to implement the optimal page replacement algorithm in most real-world scenarios. The main reason for this is that the algorithm requires knowledge of future page references, which is generally impossible to determine in advance.

6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 4, 0

S. no	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F3				2	2	2	4	4	4	4	4	4	4	4	4	3	3	3	4	4
F2		1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
F1	6	6	6	6	0	0	0	6	6	2	2	2	2	2	2	2	2	2	2	0
Hit (H)/ Faul t (F)	F	F	H	F	F	F	F	H	H	F	F	H	H	H	H	F	H	F	F	F

3. **Least Recently Used**

   In this algorithm, the page will be replaced which is least recently used.

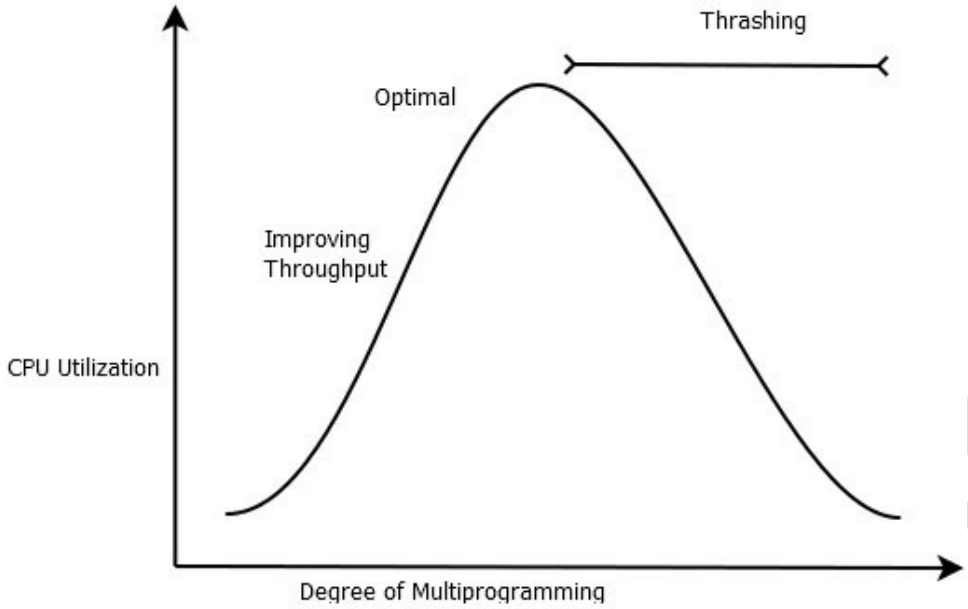
   LRU works on the principle that pages that have been recently accessed are more likely to be accessed again in the near future. By replacing the least recently used pages, it aims to retain the frequently accessed pages in memory, reducing the number of page faults and improving overall system performance.

6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0

S. no	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F3				2	2	2	4	4	4	2	2	2	2	2	2	2	2	2	2	2
F2		1	1	1	1	3	3	3	0	0	0	0	0	0	0	0	0	1	1	1
F1	6	6	6	6	0	0	0	6	6	6	1	1	1	1	1	3	3	3	3	0
Hit (H)/ Fault (F)	F	F	H	F	F	F	F	F	F	F	F	H	H	H	H	F	H	F	H	F

▼ 16. Thrashing

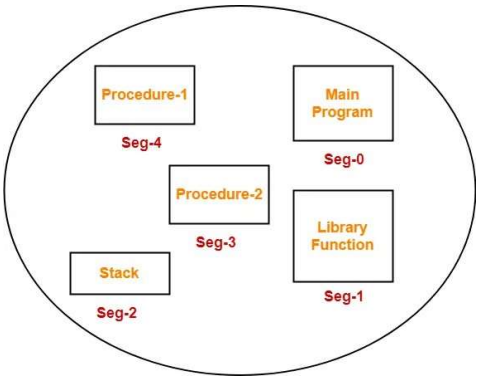
Thrashing refers to a situation in computer systems where the system spends a significant amount of time and resources continuously swapping pages between physical memory (RAM) and secondary storage (such as a hard disk) due to excessive paging activity. In thrashing, the system is busy moving pages in and out of memory rather than executing useful work, leading to severe degradation in performance.



▼ 17. Segmentation

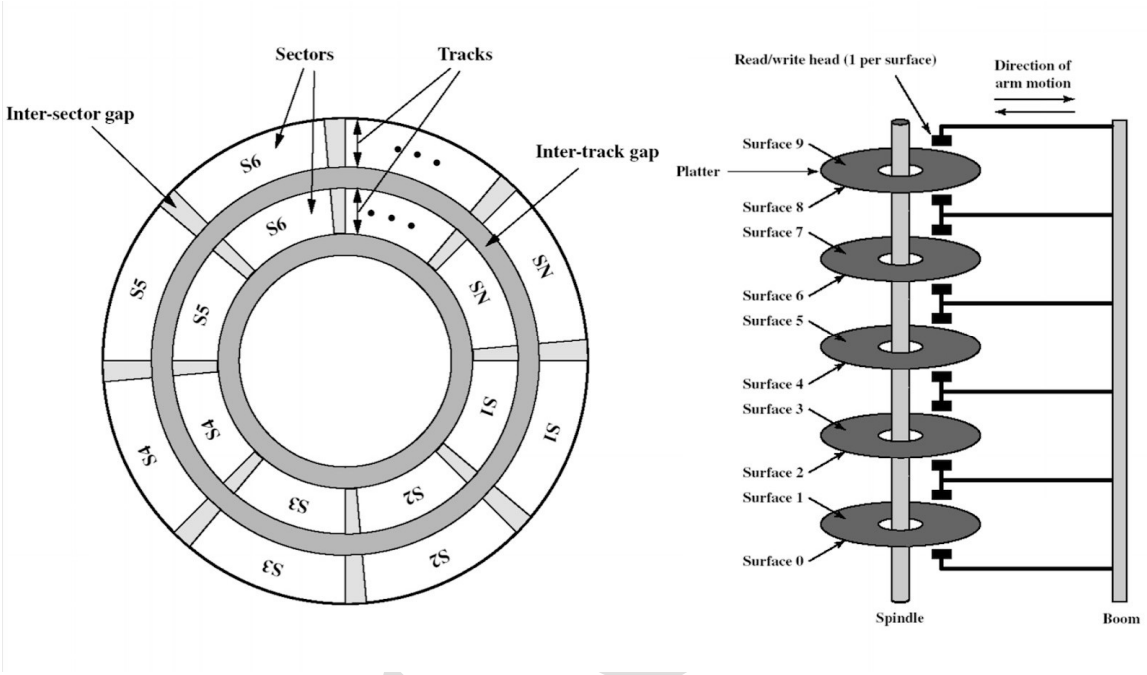
Segmentation divides processes into smaller subparts known as **modules**. The divided segments need not be placed in contiguous memory. Since there is no contiguous memory allocation, internal fragmentation does not take place. The length of the segments of the program and memory is decided by the purpose of the segment in the user program.

Segmentation came into existence because of the problems with the paging technique. In the case of the paging technique, a function or piece of code is divided into pages without considering that the relative parts of code can also get divided. Hence, for the process in execution, the CPU must load more than one page into the frames so that the complete related code is there for execution. Paging took more pages for a process to be loaded into the main memory.



Hence, segmentation was introduced in which the code is divided into modules so that related code can be combined in one single block.

▼ 18. Disk Management



Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

▼ 19. Disk scheduling algorithms

Disk scheduling is a key component of disk management that determines the order in which disk I/O (input/output) requests are processed and serviced by the disk controller. It aims to minimize the seek time and rotational latency and maximize the overall disk performance.

1. **FCFS (First-Come, First-Served)**: This is the simplest disk scheduling algorithm that processes requests in the order they arrive. It does not

consider the position of the disk head or the proximity of the requests, resulting in potential delays if there are long seek times between requests.

2. **SSTF (Shortest Seek Time First)**: This algorithm selects the request with the shortest seek time from the current position of the disk head. It minimizes the average seek time and reduces the overall disk access time. However, it may lead to starvation of requests located farther away from the current position.
3. **SCAN**: Also known as the elevator algorithm, SCAN moves the disk head in one direction (e.g., from the outermost track to the innermost or vice versa) and services requests along the way. Once it reaches the end, it changes direction and continues the same process. This algorithm provides a fair distribution of service and prevents starvation, but it may result in longer response times for requests at the far ends of the disk.
4. **C-SCAN (Circular SCAN)**: Similar to SCAN, C-SCAN moves the disk head in one direction, but instead of reversing direction, it jumps to the other end of the disk and starts again. This ensures a more consistent response time for all requests, but it may cause delays for requests that arrive after the head has passed their location.
5. **LOOK**: LOOK is a variant of SCAN that only goes as far as the last request in its current direction. Once there are no more requests in that direction, it reverses direction. This reduces unnecessary traversal of the entire disk and improves response times for requests.
6. **C-LOOK (Circular LOOK)**: Similar to C-SCAN, C-LOOK jumps to the other end of the disk without servicing requests along the way. This reduces seek time and improves disk throughput.

For images: https://www.geeksforgeeks.org/disk-scheduling-algorithms/