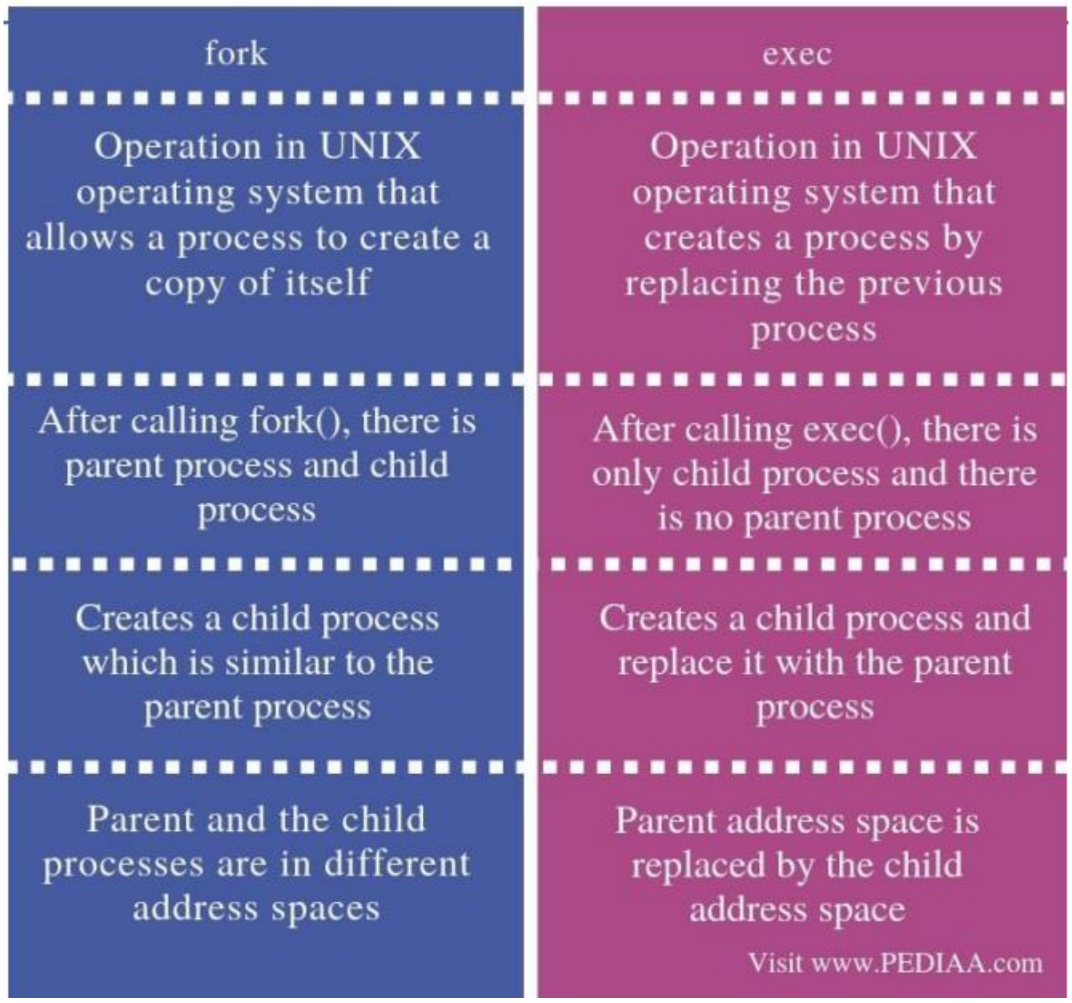
Differentiate between fork and exec system calls.

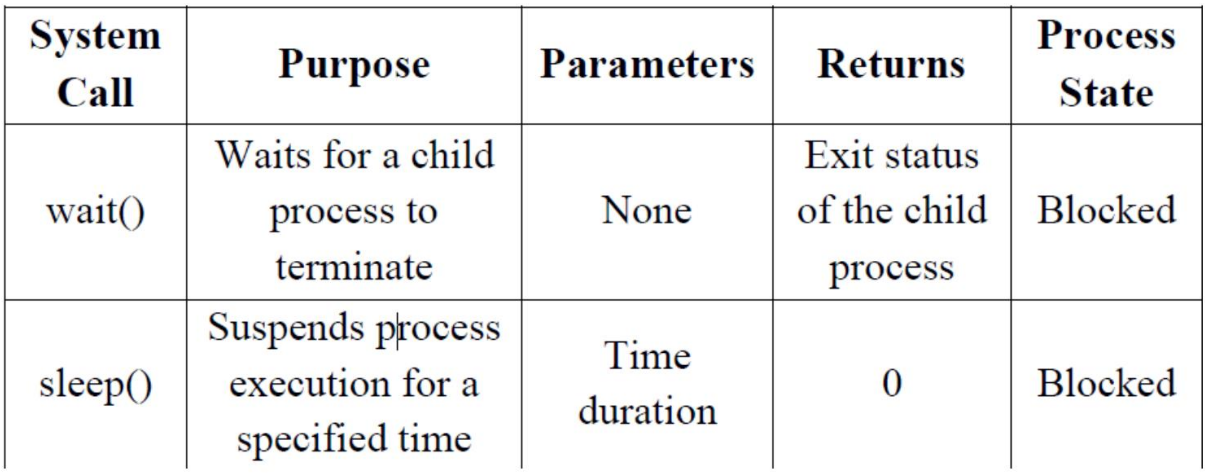


fork	exec
Operation in UNIX operating system that allows a process to create a copy of itself	Operation in UNIX operating system that creates a process by replacing the previous process
After calling fork(), there is parent process and child process	After calling exec(), there is only child process and there is no parent process
Creates a child process which is similar to the parent process	Creates a child process and replace it with the parent process
Parent and the child processes are in different address spaces	Parent address space is replaced by the child address space

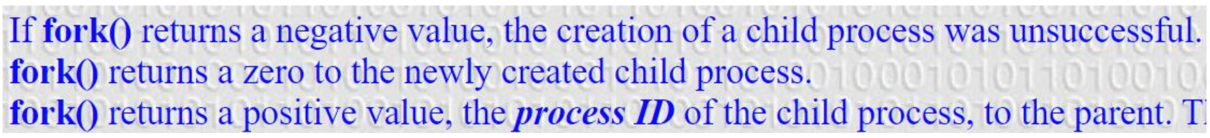
Visit www.PEDIAA.com

	fork()	exec()
1.	It is a system call in the C programming language	It is a system call of operating system
2.	It is used to create a new process	exec() runs an executable file
3.	Its return value is an integer type	It does not create a new process
4.	It does not takes any parameters.	Here the Process identifier does not changes

5.	It can return three types of integer values	In exec() the machine code, data, heap, and stack of the process are replaced by the new program.



System Call	Purpose	Parameters	Returns	Process State
wait()	Waits for a child process to terminate	None	Exit status of the child process	Blocked
sleep()	Suspends process execution for a specified time	Time duration	0	Blocked



If **fork()** returns a negative value, the creation of a child process was unsuccessful.
fork() returns a zero to the newly created child process.
fork() returns a positive value, the ***process ID*** of the child process, to the parent. T

Differentiate between monolithic and microkernel architectures along with neat diagrams.

What is a kernel ?
The kernel is a computer program at the core of a computer's operating system and has complete control over everything in the system. It manages the operations of the computer and the hardware.
There are five types of kernels:

1. A micro kernel, which only contains basic functionality.
2. A monolithic kernel, which contains many device drivers.
3. Hybrid Kernel
4. Exokernel
5. Nanokernel

But in this tutorial we will only look into Microkernel and Monolithic Kernel.

1. Microkernel :
kernel manages the operations of the computer, In microkernel the user services and kernel services are implemented in different address space.

The user services are kept in user address space, and kernel services are kept under kernel address space.

2. Monolithic kernel :

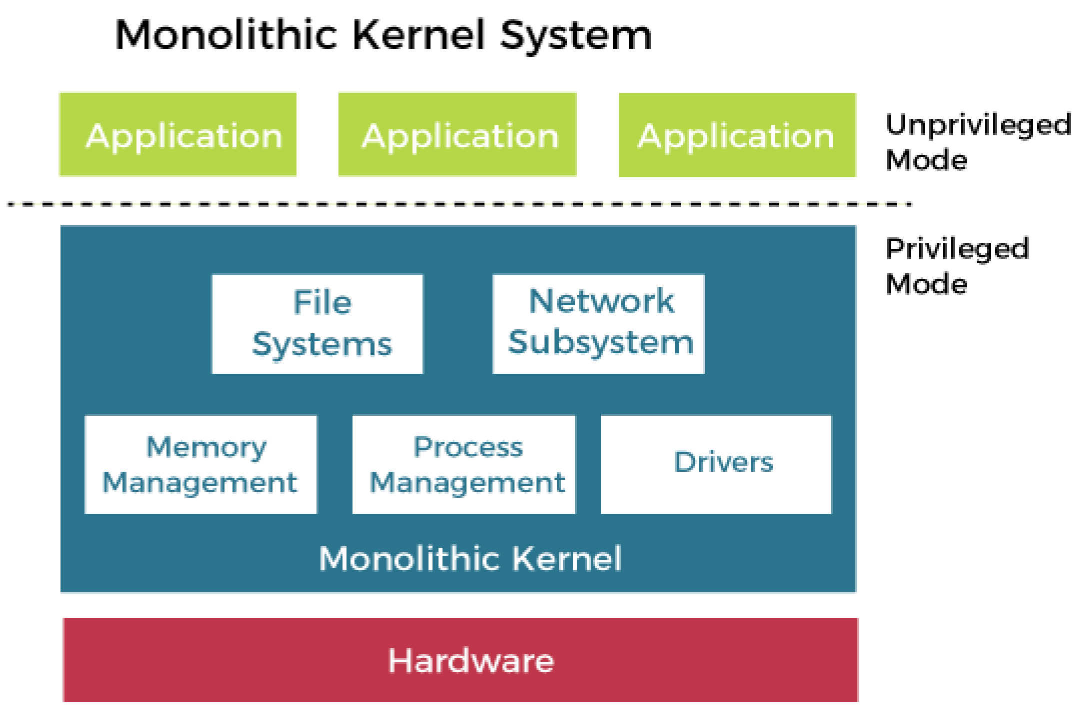
In Monolithic kernel, the entire operating system runs as a single program in kernel mode. The user services and kernel services are implemented in same address space.

Differences between Microkernel and Monolithic Kernel :

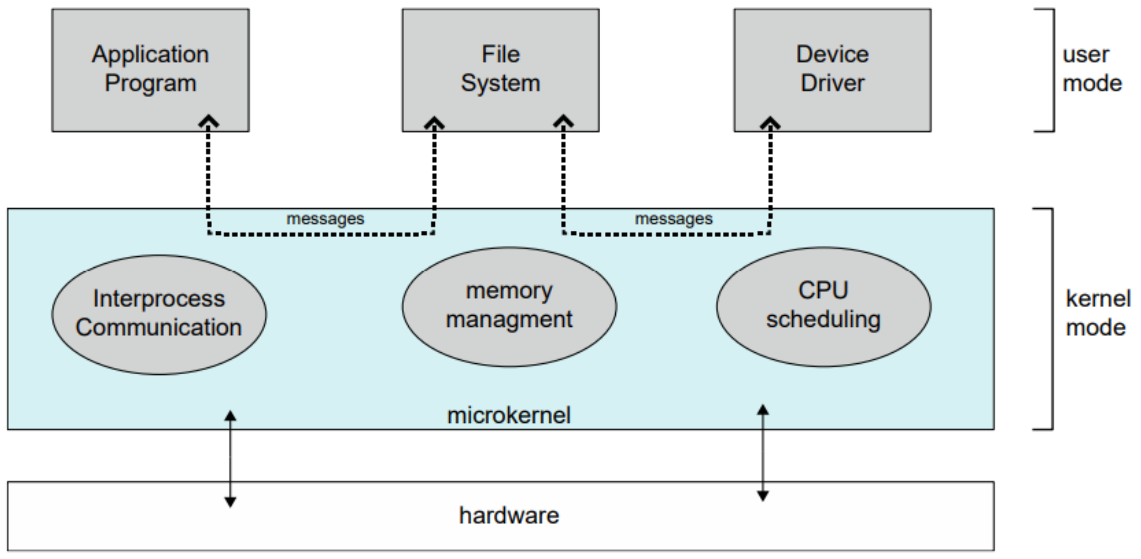
S. No.	Parameters	Microkernel	Monolithic kernel
1.	**Address Space**	In microkernel, user services and kernel services are kept in separate address space.	In monolithic kernel, both user services and kernel services are kept in the same address space.
2.	**Design and Implementation**	OS is complex to design.	OS is easy to design and implement.
3.	**Size**	Microkernels are smaller in size.	Monolithic kernel is larger than microkernel.
4.	**Functionality**	Easier to add new functionalities.	Difficult to add new functionalities.
5.	**Coding**	To design a microkernel, more code is required.	Less code when compared to microkernel
6.	**Failure**	Failure of one component does not affect the working of micro kernel.	Failure of one component in a monolithic kernel leads to the failure of the entire system.
7.	**Processing Speed**	Execution speed is low.	Execution speed is high.
8.	**Extend**	It is easy to extend Microkernel.	It is not easy to extend monolithic kernel.

9.	**Communication**	To implement IPC messaging queues are used by the communication microkernels.	Signals and Sockets are utilized to implement IPC in monolithic kernels.
10.	**Debugging**	Debugging is simple.	Debugging is difficult.
11.	**Maintain**	It is simple to maintain.	Extra time and resources are needed for maintenance.
12.	**Message passing and Context switching**	Message forwarding and context switching are required by the microkernel.	Message passing and context switching are not required while the kernel is working.
13.	**Services**	The kernel only offers IPC and low-level device management services.	The Kernel contains all of the operating system's services.
14.	**Example**	**Example:** Mac OS X.	**Example:** Microsoft Windows 95.

Monolithic



Microkernel



Explain multi-threading models with respect to user and kernel level threads.

There are three multi-threading models :

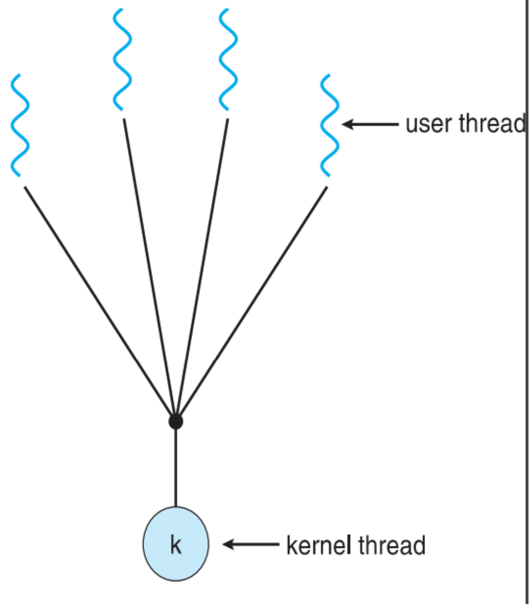
- Many-to-One

- One-to-One

- Many-to-Many

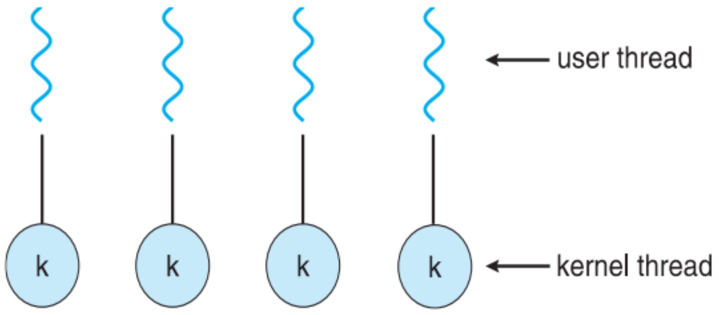
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



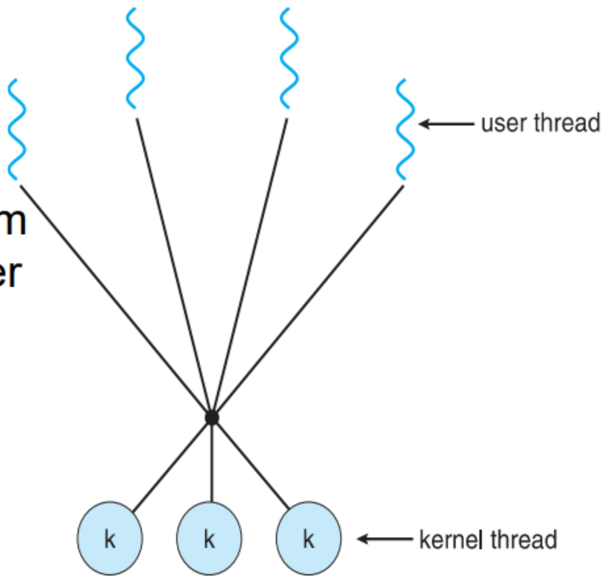
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Compare long term, short-term, medium-term scheduler with respect to the following points:

1. Selection of process

2. Frequency of Execution

Schedulers are an essential part of the operating system that decides which process to run next in the CPU. The operating system has three types of schedulers: long-term, short-term, and medium-term schedulers. Let's compare them based on two criteria: selection of the process and frequency of execution.

1. Selection of Process:

a. Long-term scheduler: The long-term scheduler is responsible for selecting new processes from the pool of waiting processes in secondary storage (e.g., hard disk) and placing them into the ready queue. The long-term scheduler is invoked infrequently, typically when a new process is created, and it decides which process to load into memory.

b. Medium-term scheduler: The medium-term scheduler is responsible for selecting the processes that are eligible for suspension and swapping them out to the secondary storage. It performs this operation to free up memory space and maintain the balance between CPU utilization and the degree of multiprogramming.

c. Short-term scheduler: The short-term scheduler is responsible for selecting the process from the ready queue and allocating the CPU to it. The short-term scheduler is invoked frequently, typically after each clock tick, and it decides which process to run next.

2. Frequency of Execution:

a. Long-term scheduler: The long-term scheduler is executed very rarely, typically when a new process is created. It is responsible for the overall performance of the system and is critical to maintaining a balance between the degree of multiprogramming and the memory utilization.

b. Medium-term scheduler: The medium-term scheduler is executed less frequently than the short-term scheduler, typically after a certain amount of time or when the system detects a memory shortage. It is responsible for maintaining a balance between CPU utilization and memory utilization by swapping processes in and out of memory.

c. Short-term scheduler: The short-term scheduler is executed very frequently, typically after each clock tick. It is responsible for allocating the CPU to the process in the ready queue, based on some criteria such as priority, burst time, and so on.

In summary, the long-term scheduler is responsible for selecting new processes from the pool of waiting processes, and it is executed infrequently. The medium-term scheduler is responsible for selecting the processes that are eligible for suspension and swapping them out to secondary storage, and it is executed less frequently than the short-term scheduler. The short-term scheduler is responsible for selecting the process from the ready queue and allocating the CPU to it, and it is executed very frequently.

Describe deadlock detection algorithm for multiple instances of a resource type.

Deadlock occurs when two or more processes are blocked and waiting for a resource that is being held by another process, which is also waiting for a resource that is being held by one of the blocked processes. In a system where multiple instances of a resource type are available, detecting deadlock becomes more complex.

To detect deadlock in a system with multiple instances of a resource type, we can use the deadlock detection algorithm. The algorithm works by creating a resource allocation graph and analysing it to check for the presence of a cycle. A cycle in the graph indicates the presence of deadlock.

The resource allocation graph is a directed graph that represents the relationships between processes and resources in the system. The graph has two types of nodes: process nodes and resource nodes. The edges in the graph represent the requests for and allocation of resources. A directed edge from a process node to a resource node indicates a request for the resource, while a directed edge from a resource node to a process node indicates the allocation of the resource to the process.

The steps to detect deadlock in a system with multiple instances of a resource type are as follows:

1. Create a resource allocation graph that represents the relationships between processes and resources in the system.
2. Perform a depth-first search (DFS) of the graph starting from each process node. During the DFS, mark each visited node and store it in a list.
3. If a marked node is encountered during the DFS, a cycle exists in the graph, indicating the presence of deadlock. The cycle can be detected by tracing the list of visited nodes back to the marked node.
4. If no cycle is found during the DFS, repeat the process with the next unmarked process node.
5. If all process nodes have been visited, and no cycle has been found, there is no deadlock in the system.

Once deadlock is detected, the system can take corrective measures to resolve the deadlock. These measures may include aborting one or more processes or releasing one or more resources to break the deadlock.

In summary, the deadlock detection algorithm for multiple instances of a resource type involves creating a resource allocation graph and performing a depth-first search to detect the presence of a cycle, which indicates the presence of deadlock. Once deadlock is detected, corrective measures can be taken to resolve the deadlock and prevent it from occurring again in the future.

Explain compare and swap construct to solve critical section problem.

The critical section problem arises when multiple processes or threads are executing a shared code section, and access to shared resources must be synchronized to avoid race conditions and data inconsistencies. One of the methods to solve the critical section problem is the use of synchronization primitives such as locks, semaphores, and mutexes. However, these primitives can cause problems such as deadlocks and priority inversion.

Another method to solve the critical section problem is by using the Compare-and-Swap (CAS) construct, which is a low-level synchronization primitive that provides a lock-free mechanism to solve the critical section problem. The CAS construct allows a process to read the value of a shared memory location, compare it with an expected value, and atomically swap the value if the comparison is true.

The steps to use the CAS construct to solve the critical section problem are as follows:

1. Define a shared variable (e.g., a flag) that represents the state of the critical section. The variable can take on two values, such as 0 (unlocked) and 1 (locked).
2. Each process that needs to access the critical section checks the value of the shared variable using the CAS construct. If the value of the shared variable is 0 (unlocked), the process sets the value of the variable to 1 (locked) using the CAS construct. If the value of the shared variable is already 1 (locked), the process retries the CAS operation until it succeeds.
3. After a process finishes executing the critical section, it sets the value of the shared variable to 0 (unlocked) using the CAS construct.

The use of the CAS construct ensures that only one process can access the critical section at a time, and the access to the shared variable is synchronized without the need for locks, semaphores, or mutexes. The CAS construct is also free from deadlocks and priority inversion problems.

In summary, the Compare-and-Swap (CAS) construct is a low-level synchronization primitive that provides a lock-free mechanism to solve the critical section problem. The CAS construct allows a process to read the value of a shared memory location, compare it with an expected value, and atomically swap the value if the comparison is true. The use of the CAS construct ensures that only one process can access the critical section at a time and provides a synchronization mechanism that is free from deadlocks and priority inversion.

State readers' writers' problem. Solve by writing pseudo code and explain the problem solution using semaphores.

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e., when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers.
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

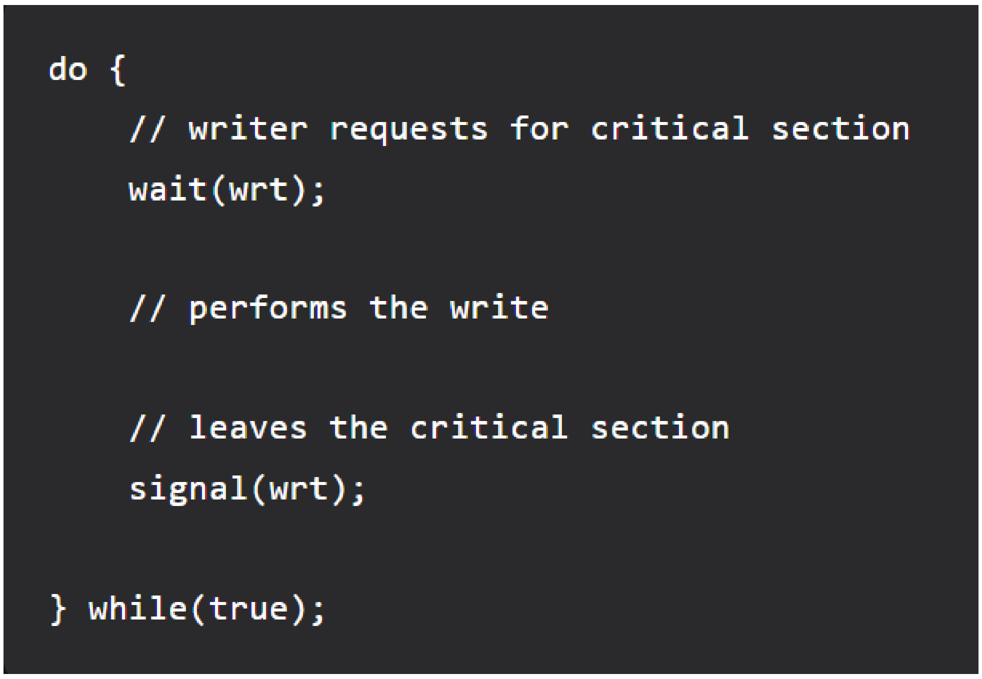
Functions for semaphore:

– wait (): decrements the semaphore value.

– signal (): increments the semaphore value.

Writer process:

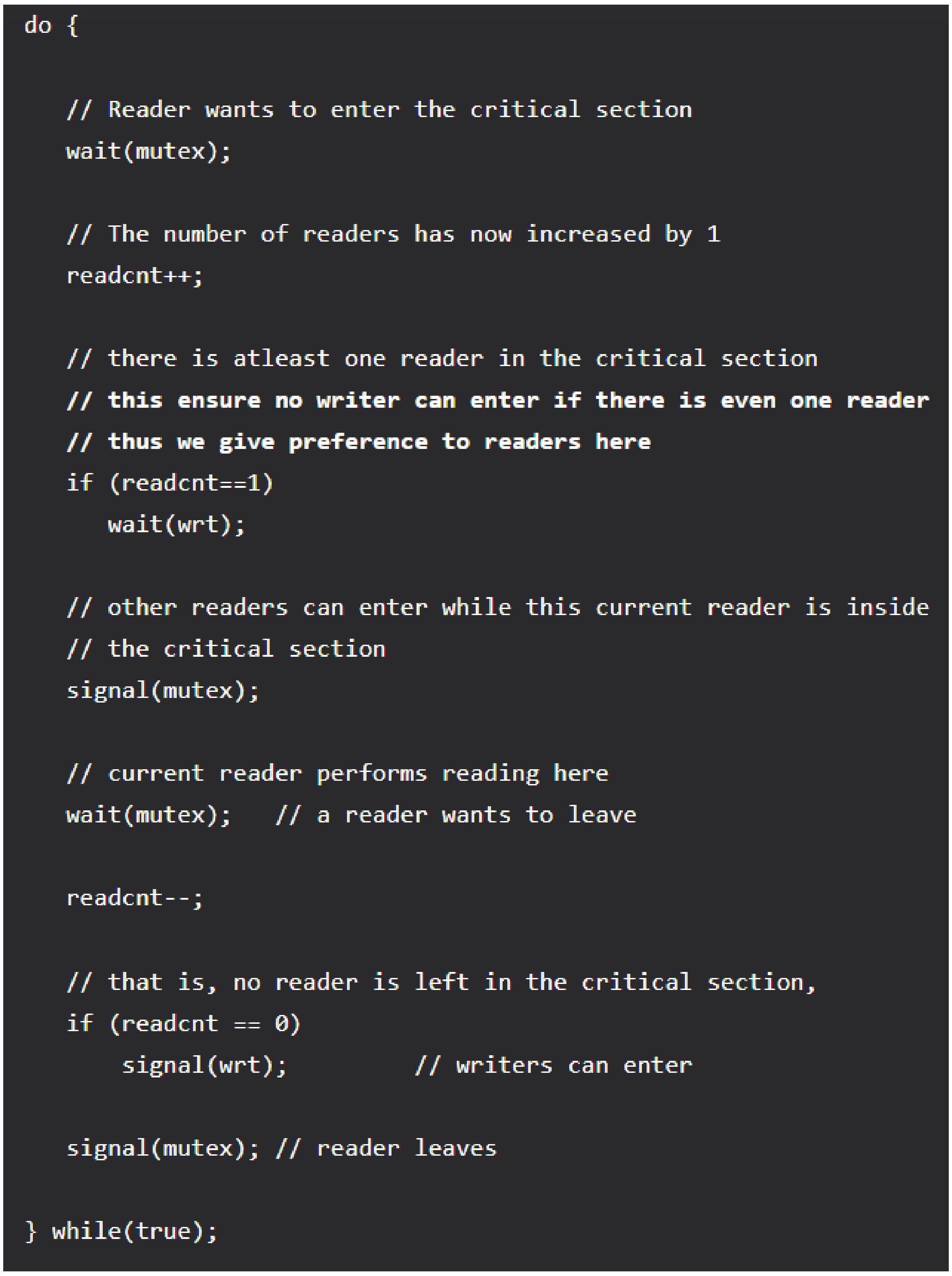
1. Writer requests the entry to critical section.
2. If allowed i.e. wait () gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.



```
do {
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);

} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.
    - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.



```
do {

   // Reader wants to enter the critical section
   wait(mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even one reader
   // thus we give preference to readers here
   if (readcnt==1)
      wait(wrt);

   // other readers can enter while this current reader is inside
   // the critical section
   signal(mutex);

   // current reader performs reading here
   wait(mutex);   // a reader wants to leave

   readcnt--;

   // that is, no reader is left in the critical section,
   if (readcnt == 0)
       signal(wrt);         // writers can enter

   signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore ‘**wrt**‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

State Producers’ Consumers' problem. Solve by writing pseudo code and explain the problem solution using semaphores.

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

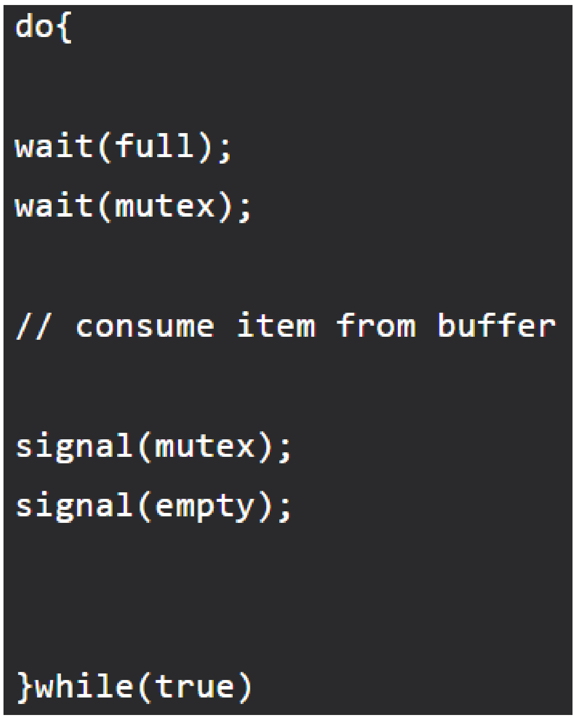
Initialization of semaphores –
mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

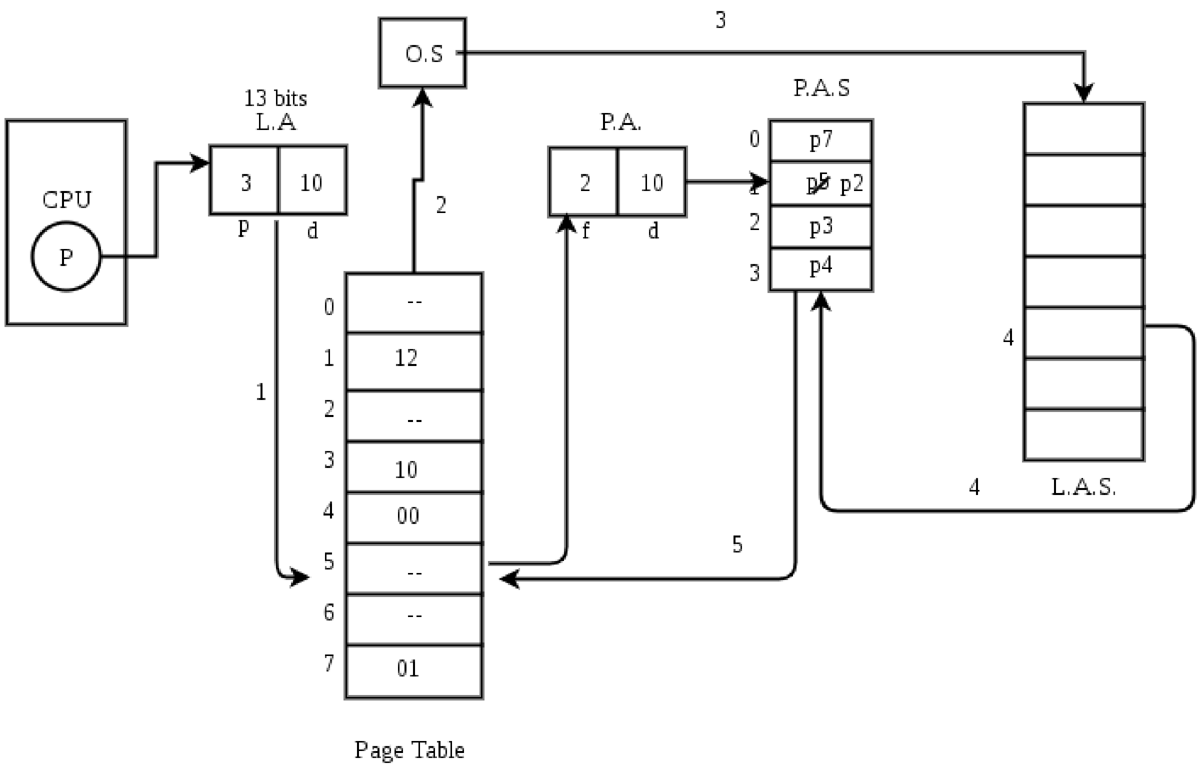


```
do{


wait(full);
wait(mutex);


// consume item from buffer


signal(mutex);
signal(empty);



}while(true)
```

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

What is demand paging? What are the six steps in handling the page fault?

Demand Paging: The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging. The process includes the following steps are as follows:



1. If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.

2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.

3. The OS will search for the required page in the logical address space.

4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.

5. The page table will be updated accordingly.

6. The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages:

- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any time.
- A process may be larger than all the main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

Disadvantages:

- It can slow down the system performance, as data needs to be constantly transferred between the physical memory and the hard disk.
- It can increase the risk of data loss or corruption, as data can be lost if the hard disk fails or if there is a power outage while data is being transferred to or from the hard disk.
- It can increase the complexity of the memory management system, as the operating system needs to manage both physical and virtual memory.

List file allocation methods and compare them with respect to following points:

1. External Fragmentation

2.Access Time

3.Suitable for which file access methods

File allocation is the process of assigning disk space to a file. There are three common file allocation methods: contiguous allocation, linked allocation, and indexed allocation. These methods can be compared based on the following points:

1. External Fragmentation:

Contiguous allocation is prone to external fragmentation, which occurs when there is not enough contiguous free space to allocate a new file. Linked allocation and indexed allocation do not suffer from external fragmentation.

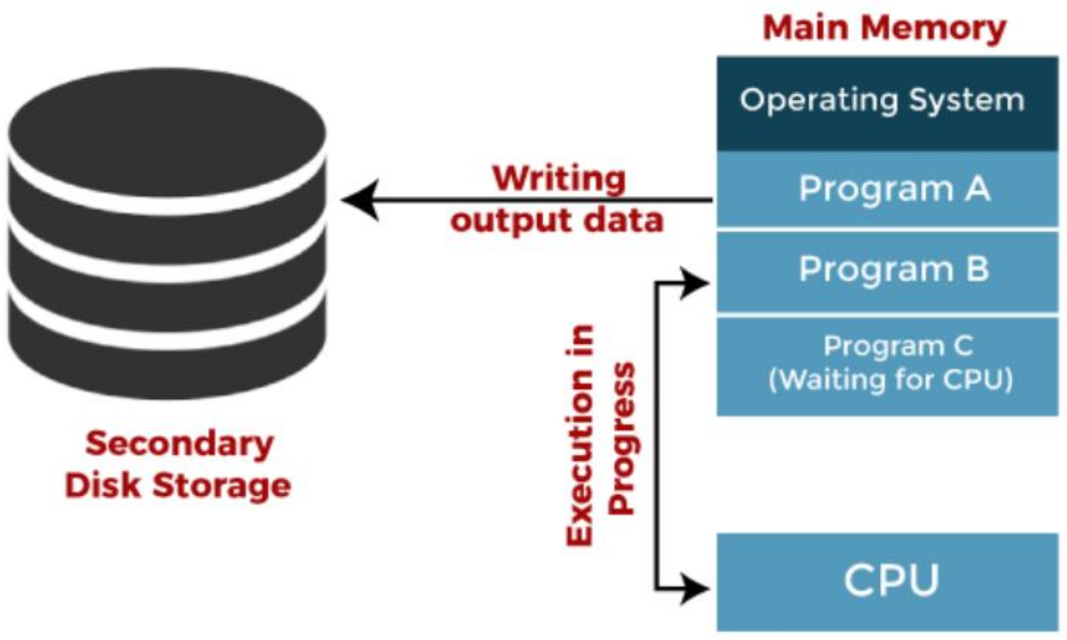
2. Access Time:

Contiguous allocation provides faster access times compared to linked allocation and indexed allocation because the files are stored in contiguous blocks on disk. Linked allocation and indexed allocation require additional processing time to follow the links or to access the index block before accessing the file data.

3. Suitable for which file access methods:

Contiguous allocation is suitable for sequential access files because the data is stored in contiguous blocks, allowing the system to read the data in one go. Linked allocation is suitable for variable-sized files that can grow and shrink dynamically because it allows for efficient use of space. Indexed allocation is suitable for random access files because it allows for direct access to the data blocks through an index.

Explain multiprogramming operating system.



Jobs in multiprogramming system

Multiprogramming operating system is an operating system that enables multiple programs to run concurrently on a single CPU. In other words, it allows the system to execute multiple programs simultaneously by dividing the CPU time among them.

In a multiprogramming operating system, the operating system loads multiple programs into memory and switches between them rapidly, giving the appearance that they are executing simultaneously. When a program requests I/O operations, the CPU switches to another program that can continue execution, minimizing the idle time of the CPU.

Multiprogramming operating systems are designed to maximize the CPU utilization and improve the throughput of the system. It allows the system to execute multiple tasks concurrently, increasing the efficiency of the system. However, it requires sophisticated scheduling algorithms to manage the CPU time among the processes effectively.

One of the benefits of a multiprogramming operating system is that it allows the system to respond to the user quickly by allowing the user to run multiple programs simultaneously. This feature is particularly useful for interactive systems that require rapid response times, such as graphical user interfaces and web browsers.

In summary, a multiprogramming operating system is an operating system that enables multiple programs to run concurrently on a single CPU. It increases the efficiency of the system by maximizing the CPU utilization and allowing the system to respond quickly to user requests.

Write four advantages and disadvantages of layered architecture of Operating Systems

Advantages of Layered Structure

There are several advantages of the layered structure of operating system design, such as:

1. **Modularity:** This design promotes modularity as each layer performs only the tasks it is scheduled to perform.
2. **Easy debugging:** As the layers are discrete so it is very easy to debug. Suppose an error occurs in the CPU scheduling layer. The developer can only search that layer to debug, unlike the Monolithic system where all the services are present.
3. **Easy update:** A modification made in a particular layer will not affect the other layers.
4. **No direct access to hardware:** The hardware layer is the innermost layer present in the design. So, a user can use the services of hardware but cannot directly modify or access it, unlike the Simple system in which the user had direct access to the hardware.
5. **Abstraction:** Every layer is concerned with its functions. So the functions and implementations of the other layers are abstract to it.

Disadvantages of Layered Structure

Though this system has several advantages over the Monolithic and Simple design, there are also some disadvantages, such as:

1. **Complex and careful implementation:** As a layer can access the services of the layers below it, so the arrangement of the layers must be done carefully. For example, the backing storage layer uses the services of the memory management layer. So, it must be kept below the memory management layer. Thus with great modularity comes complex implementation.

2. **Slower in execution:** If a layer wants to interact with another layer, it requests to travel through all the layers present between the two interacting layers. Thus, it increases response time, unlike the Monolithic system, which is faster than this. Thus, an increase in the number of layers may lead to a very inefficient design.
3. **Functionality:** It is not always possible to divide the functionalities. Many times, they are interrelated and can't be separated.
4. **Communication:** No communication between non-adjacent layers.

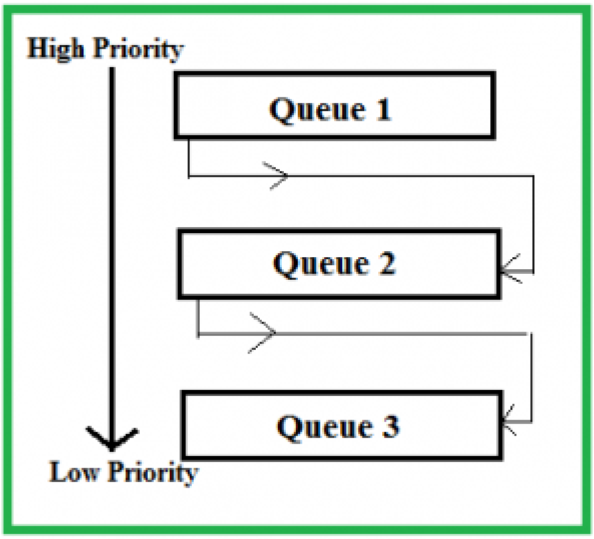
On a simple paged system, associative registers hold the most active page entries and the full page table is stored in the main memory. If references satisfied by assciative registers take 100 ns, and references through the main memory page table take 180ns , what must be the hit ratio to achieve an effective access time of 125ns ?

Associative registers, also known as translation lookaside buffers (TLBs), are small hardware caches that store recently accessed page table entries. These registers are used to speed up the process of translating a virtual address to a physical address during page fault service.

Differentiate between user level and kernel level thread.

S. No.	Parameters	User Level Thread	Kernel Level Thread
1.	**Implemented by**	User threads are implemented by users.	Kernel threads are implemented by Operating System (OS).
2.	**Recognize**	Operating System doesn't recognize user level threads.	Kernel threads are recognized by Operating System.
3.	**Implementation**	Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
4.	**Context switch time**	Context switch time is less.	Context switch time is more.
5.	**Hardware support**	Context switch requires no hardware support.	Hardware support is needed.
6.	**Blocking operation**	If one user level thread performs blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
7.	**Multithreading**	Multithread applications cannot take advantage of multiprocessing.	Kernels can be multithreaded.
8.	**Creation and Management**	User level threads can be created and managed more quickly.	Kernel level threads take more time to create and manage.

Multilevel feedback queue scheduling with diagram



Characteristics of Multilevel Feedback Queue Scheduling:

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system, and processes are allowed to move between queues.
- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,

Features of Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling:

Multiple queues: Like MLQ scheduling, MLFQ scheduling divides processes into multiple queues based on their priority levels. However, unlike MLQ scheduling, processes can move between queues based on their behaviour and needs.

Priorities adjusted dynamically: The priority of a process can be adjusted dynamically based on its behaviour, such as how much CPU time it has used or how often it has been blocked. Higher-priority processes are given more CPU time and lower-priority processes are given less.

Time-slicing: Each queue is assigned a time quantum or time slice, which determines how much CPU time a process in that queue is allowed to use before it is pre-empted and moved to a lower priority queue.

Feedback mechanism: MLFQ scheduling uses a feedback mechanism to adjust the priority of a process based on its behavior over time. For example, if a process in a lower-priority queue uses up its time slice, it may be moved to a higher-priority queue to ensure it gets more CPU time.

Pre-emption: Pre-emption is allowed in MLFQ scheduling, meaning that a higher-priority process can pre-empt a lower-priority process to ensure it gets the CPU time it needs.

Advantages of Multilevel Feedback Queue Scheduling:

- It is more flexible.
- It allows different processes to move between different queues.
- It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.

Disadvantages of Multilevel Feedback Queue Scheduling:

- The selection of the best scheduler, it requires some other means to select the values.
- It produces more CPU overheads.
- It is the most complex algorithm.

Multilevel feedback queue scheduling, however, allows a process to move between queues. Multilevel Feedback Queue Scheduling **(MLFQ)** keeps analysing the behaviour (time of execution) of processes and according to which it changes its priority.

Describe deadlock prevention by breaking the circular wait condition.

Deadlock prevention is a strategy used to prevent the occurrence of deadlocks, which happen when two or more processes are blocked and waiting for resources held by the other process, causing a deadlock situation. One way to prevent deadlocks is to break the circular wait condition.

Circular wait occurs when a set of processes are waiting for resources held by other processes in the set in a circular chain. For example, process A is waiting for a resource held by process B, process B is waiting for a resource held by process C, and process C is waiting for a resource held by process A. This creates a circular chain of dependencies, and no process in the chain can proceed.

To break the circular wait condition, a system can use a technique called resource allocation graph. This technique represents the allocation and request of resources in a graph form. Each process is represented by a node in the graph, and each resource is represented by a directed edge that connects the process node with the resource node.

To prevent deadlock, the system should avoid the formation of a circular wait in the resource allocation graph. This can be achieved by ensuring that the graph is acyclic. One way to ensure this is to impose a partial order on the resources and require that each process request resources in a certain order. This order can be imposed using a numbering system or by other means.

Another way to prevent deadlocks by breaking the circular wait is to use a timeout mechanism. This mechanism involves setting a timeout period for each process that requests a resource. If the process does not receive the resource within the timeout period, it is released, and the process can try again later. This ensures that no process remains blocked indefinitely, breaking the circular wait condition.

In summary, breaking the circular wait condition is an effective way to prevent deadlocks. Techniques such as resource allocation graph and timeout mechanism can be used to achieve this.

Explain the test and set construct to solve the critical section problem with example.

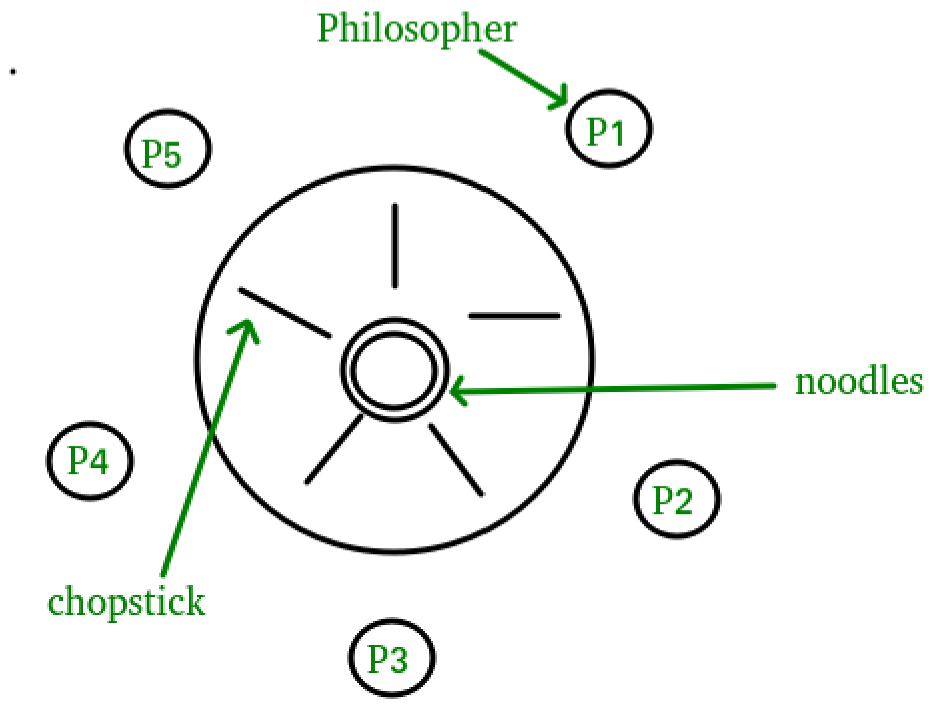
Here, the shared variable is lock which is initialized to false. TestAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as TestAndSet(lock) will return false and it’ll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true. Mutual exclusion is ensured. Once the first process gets out of the critical section, lock is changed to false. So, now the other processes can enter one by one. Progress is also ensured. However, after the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

Test and Set Pseudocode –

```
//Shared variable lock initialized to false
boolean lock;

boolean TestAndSet (boolean &target){
    boolean rv = target;
    target = true;
    return rv;
}

while(1){
    while (TestAndSet(lock));
    critical section
    lock = false;
    remainder section
}
```

State the dining philosophers problem. Solve by writing pseudo code and explain the solution using monitors.

Dining-Philosophers Problem – N philosophers seated around a circular table



- There is one chopstick between two philosophers.
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

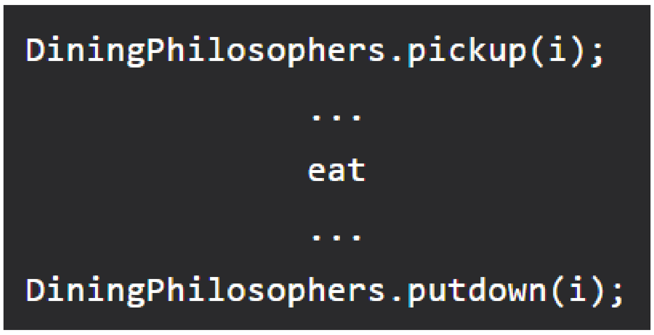
Monitors alone are not sufficiency to solve this, we need monitors with *condition variables* **Monitor-based Solution to Dining Philosophers**. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure: **THINKING** – When philosopher doesn't want to gain access to either fork. **HUNGRY** – When philosopher wants to enter the critical section. **EATING** – When philosopher has got both the forks, i.e., he has entered the section. Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating (state[(i+4) % 5]! = EATING) and (state[(i+1) % 5]! = EATING).

```
monitor DP
{
    status state[5];
    condition self[5];

    // Pickup chopsticks
    Pickup(int i)
    {
        // indicate that I’m hungry
        state[i] = hungry;

        // set state to eating in test()
        // only if my left and right neighbors
        // are not eating
        test(i);

        // if unable to eat, wait to be signaled
        if (state[i] != eating)
            self[i].wait;
    }

    // Put down chopsticks
    Putdown(int i)
    {

        // indicate that I’m thinking
        state[i] = thinking;

        // if right neighbor R=(i+1)%5 is hungry and
        // both of R’s neighbors are not eating,
        // set R’s state to eating and wake it up by
        // signaling R’s CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }

    test(int i)
    {

        if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {

            // indicate that I’m eating
            state[i] = eating;

            // signal() has no effect during Pickup(),
            // but is important to wake up waiting
            // hungry philosophers during Putdown()
            self[i].signal();
        }
    }

    init()
    {

        // Execution of Pickup(), Putdown() and test()
        // are all mutually exclusive,
        // i.e. only one at a time can be executing
for
    i = 0 to 4

        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
} // end of monitor
```

Above Program is a monitor solution to the dining-philosopher problem. We also need to declare.

condition self[5];

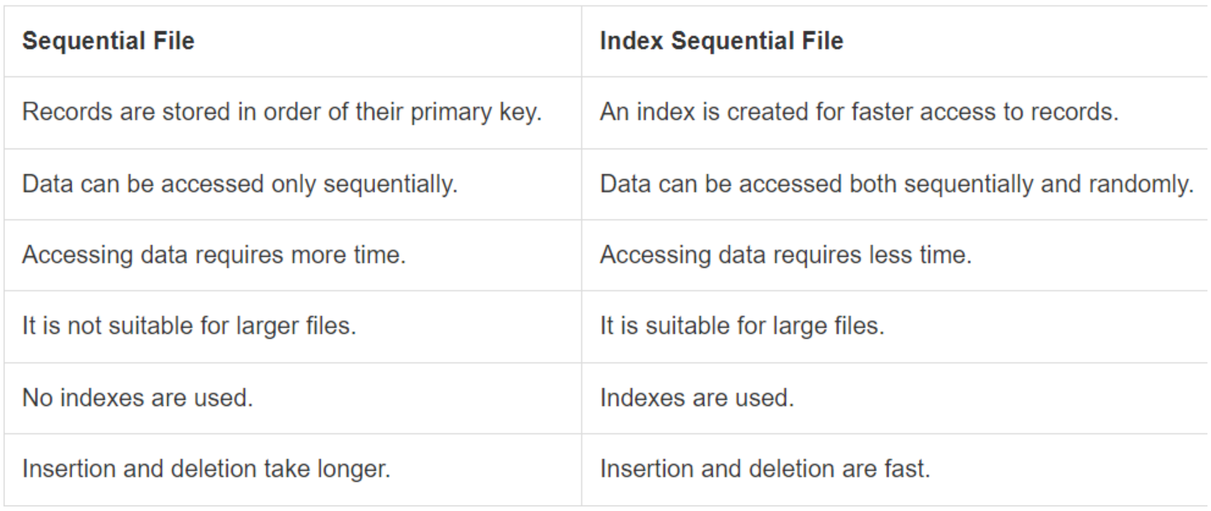
This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup (). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:



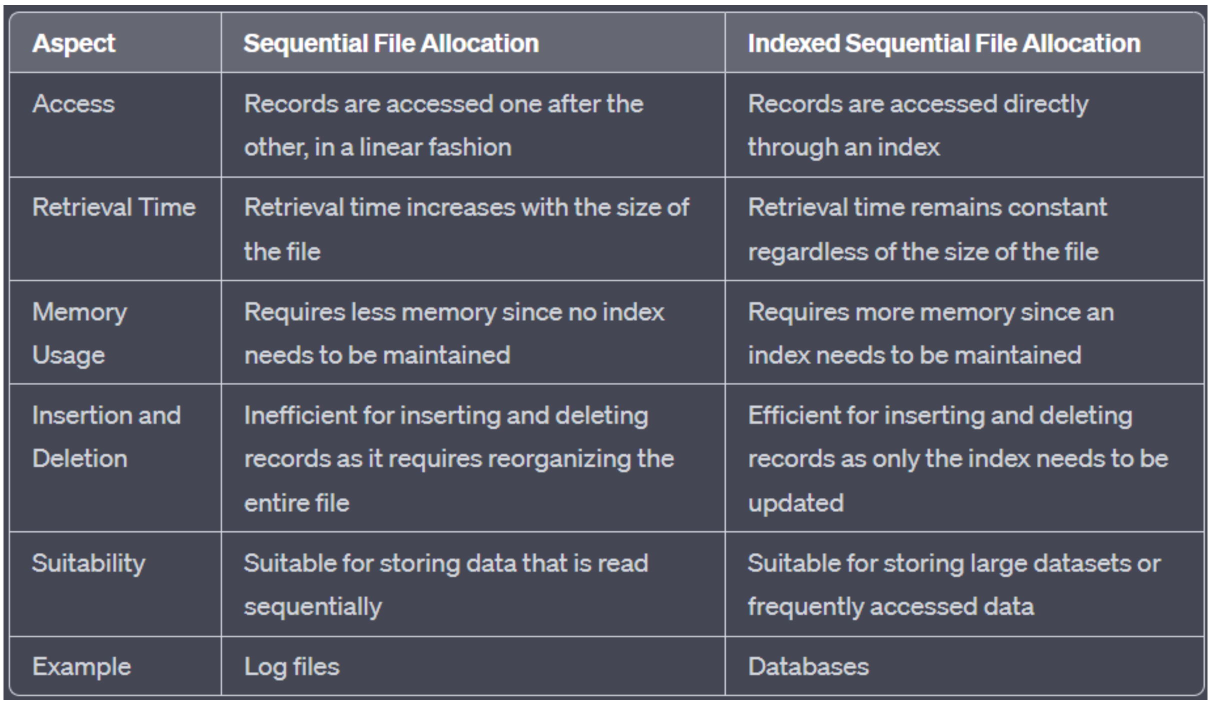
```
DiningPhilosophers.pickup(i);
              ...
              eat
              ...
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

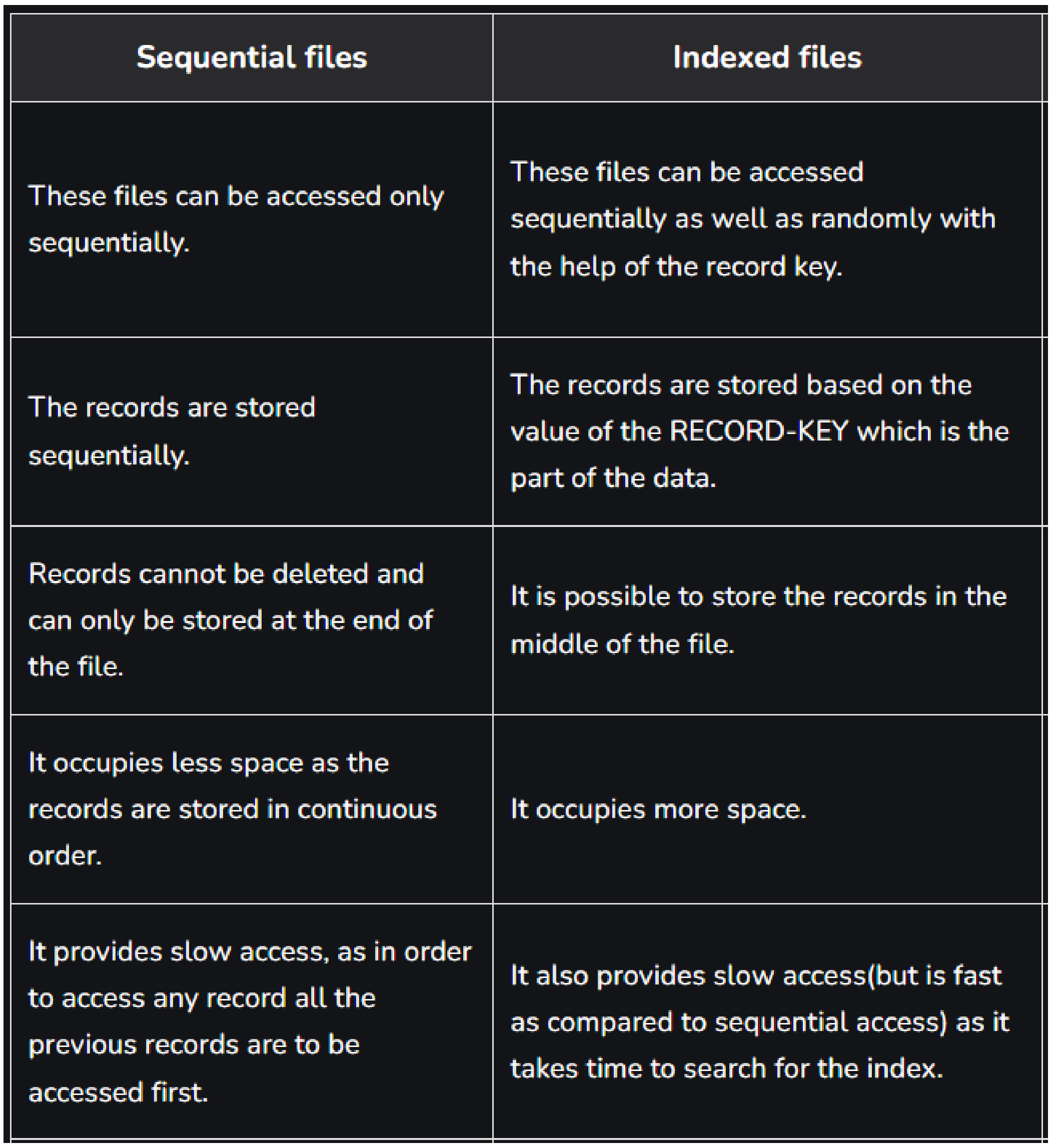
Compare sequential and indexed sequential file organisation methods.



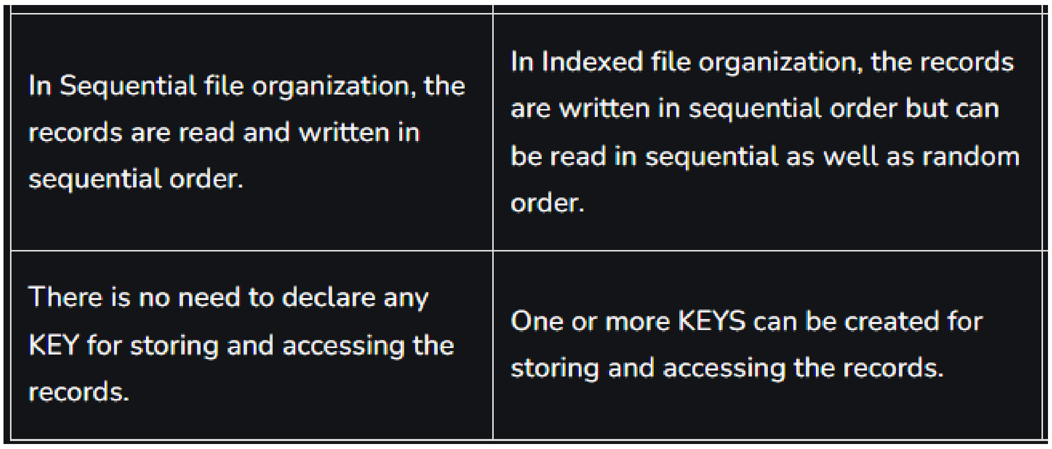
Sequential File	**Index Sequential File**
Records are stored in order of their primary key.	An index is created for faster access to records.
Data can be accessed only sequentially.	Data can be accessed both sequentially and randomly.
Accessing data requires more time.	Accessing data requires less time.
It is not suitable for larger files.	It is suitable for large files.
No indexes are used.	Indexes are used.
Insertion and deletion take longer.	Insertion and deletion are fast.



Aspect	Sequential File Allocation	Indexed Sequential File Allocation
Access	Records are accessed one after the other, in a linear fashion	Records are accessed directly through an index
Retrieval Time	Retrieval time increases with the size of the file	Retrieval time remains constant regardless of the size of the file
Memory Usage	Requires less memory since no index needs to be maintained	Requires more memory since an index needs to be maintained
Insertion and Deletion	Inefficient for inserting and deleting records as it requires reorganizing the entire file	Efficient for inserting and deleting records as only the index needs to be updated
Suitability	Suitable for storing data that is read sequentially	Suitable for storing large datasets or frequently accessed data
Example	Log files	Databases



Sequential files	Indexed files
These files can be accessed only sequentially.	These files can be accessed sequentially as well as randomly with the help of the record key.
The records are stored sequentially.	The records are stored based on the value of the RECORD-KEY which is the part of the data.
Records cannot be deleted and can only be stored at the end of the file.	It is possible to store the records in the middle of the file.
It occupies less space as the records are stored in continuous order.	It occupies more space.
It provides slow access, as in order to access any record all the previous records are to be accessed first.	It also provides slow access(but is fast as compared to sequential access) as it takes time to search for the index.



In Sequential file organization, the records are read and written in sequential order.	In Indexed file organization, the records are written in sequential order but can be read in sequential as well as random order.
There is no need to declare any KEY for storing and accessing the records.	One or more KEYS can be created for storing and accessing the records.

5 objectives of file management systems

1. Data storage: One of the main objectives of file management systems is to provide efficient storage and retrieval of data. The system should be

designed to store data in a way that optimizes the use of storage resources and enables fast and reliable retrieval of data.

2. Data access: Another objective of file management systems is to provide fast and reliable access to data. The system should be designed to provide quick and efficient access to data, even when the data is stored on multiple physical devices.

3. Data security: File management systems should be designed to protect data from unauthorized access, modification, or deletion. The system should provide security features, such as user authentication, access control, and encryption, to ensure that data is only accessible to authorized users.

4. Data backup and recovery: File management systems should provide mechanisms for backing up data and recovering data in case of system failures, disasters, or other unforeseen events. The system should be designed to ensure that data is not lost or corrupted due to hardware or software failures.

5. File organization: Another objective of file management systems is to provide an organized and logical structure for storing and accessing files. The system should provide features for creating, renaming, and deleting files, as well as for organizing files into directories or folders, to help users easily locate and manage their data.

What is record blocking? explain three methods of record blocking.

Record blocking is a technique used in database systems to improve I/O performance by grouping together multiple records into a single physical block on disk. This reduces the number of disk accesses required to retrieve a set of records, thereby improving query performance.

Here are three methods of record blocking:

1. Fixed-Length Block Technique: In this technique, a fixed number of records are grouped together into a single block. Each block is of a fixed

size, and it contains the same number of records. The block size is determined based on the record size and the available disk space. This technique is simple to implement and easy to understand, but it may result in some wasted space in each block if the records are of different sizes.

2. Variable-Length Block Technique: In this technique, the block size is variable, and it is determined based on the number of records that can fit into a block. The block size is adjusted dynamically based on the record size, and it is optimized to minimize the wasted space in each block. This technique requires more sophisticated algorithms to manage the variable block size, but it can be more efficient in terms of space utilization.

3. Clustered Block Technique: In this technique, related records are grouped together into a single block based on a specific attribute, such as a key. For example, all the records with the same value of a particular attribute can be grouped together into a single block. This technique reduces the number of disk accesses required to retrieve related records and can improve query performance for certain types of queries. However, it requires more complex indexing structures to manage the clustering, and it may not be suitable for all types of data.

May 2022

What is the role of linker and loader? Draw the loader diagram and state it's three functions.

A loader is a system program that performs the loading function. In other words we can say that the loader is a program which **accepts the object program,** prepares this program for **execution** by computer and initiates the execution. See Fig. 5.1.

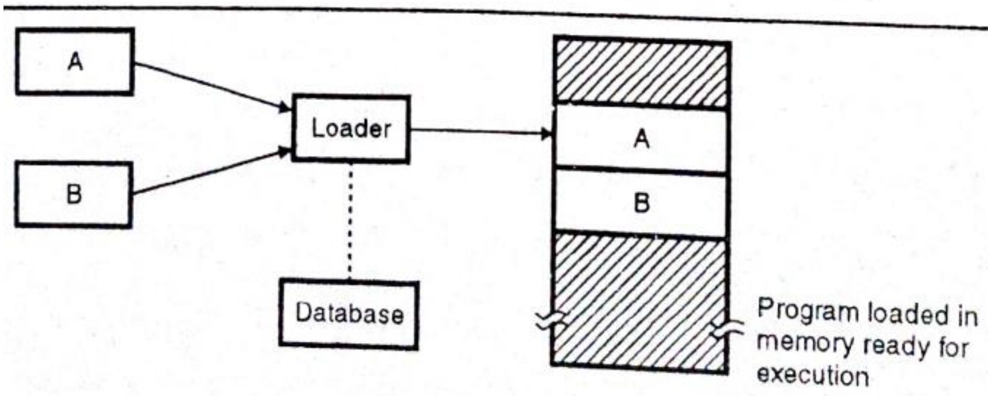


Fig. 5.1 : General loading

Many loaders also support relocation and linking. Some systems have a linker to perform the linking operations and a separate loader to handle relocation and loading.

In short a loader must perform following 4 functions :

(a) Allocate space in memory for the programs (allocation).

(b) Resolve symbolic references between object programs (linking).

(c) Adjust all address dependent locations, such as address constant to correspond to the allocated space (relocation).

(d) Physically place the machine instructions and data into memory (loading).

What is meant by dual mode of operation in context of execution of a process.

There are two modes of execution kernel mode and user mode for a process.

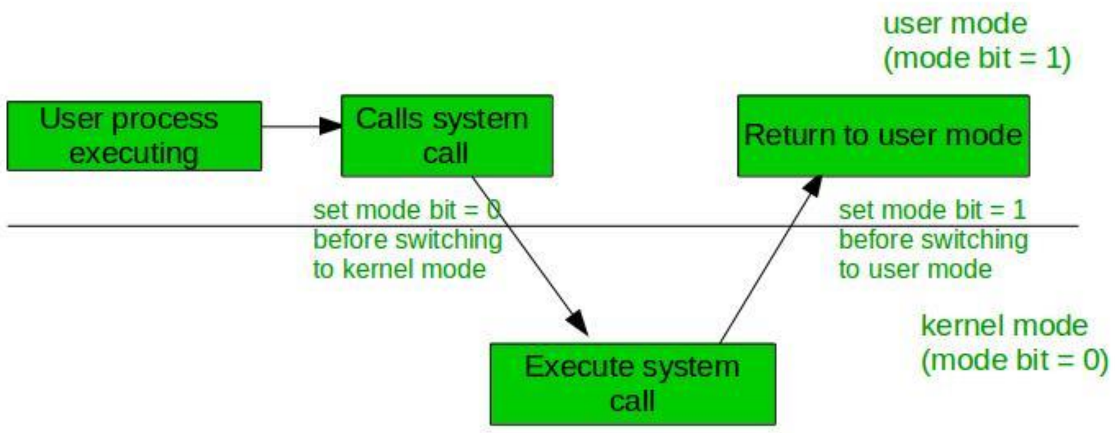
User mode –

When the computer system is run by user applications like creating a text document or using any application program, then the system is in user mode.

When the user application requests for a service from the operating system or an interrupt occurs or system call, then there will be a transition from user to kernel mode to fulfil the requests.

Note: *To switch from kernel mode to user mode, the mode bit should be 1.*

Given below image describes what happens when an interruption occurs:



Kernel Mode: When the system boots, the hardware starts in kernel mode and when the operating system is loaded, it starts user application in user mode. To provide protection to the hardware, we have privileged instructions which execute only in kernel mode. If the user attempts to run privileged instruction in user mode, then it will treat instruction as illegal and traps to OS. Some of the privileged instructions are:

1. Handling Interrupts
2. To switch from user mode to kernel mode.
3. Input-Output management.

Advantages:

- Protection: Dual-mode operation provides a layer of protection between user programs and the operating system.
- Stability: Dual-mode operation helps to ensure system stability by preventing user programs from interfering with system-level operations.

- Flexibility: Dual-mode operation allows the operating system to support a wide range of applications and hardware devices.
- Debugging: Dual-mode operation makes it easier to debug and diagnose problems with the operating system and applications.
- Security: Dual-mode operation enhances system security by preventing unauthorized access to critical system resources.
- Efficiency: Dual-mode operation can improve system performance by reducing overhead associated with system-level operations.
- Compatibility: Dual-mode operation ensures backward compatibility with legacy applications and hardware devices.
- Isolation: Dual-mode operation provides isolation between user programs, preventing one program from interfering with another.
- Reliability: Dual-mode operation enhances system reliability by preventing crashes and other errors caused by user programs.

Disadvantages:

- Performance: Dual-mode operation can introduce overhead and reduce system performance.
- Complexity: Dual-mode operation can increase system complexity and make it more difficult to develop and maintain operating systems.
- Security: Dual-mode operation can introduce security vulnerabilities.
- Reliability: Dual-mode operation can introduce reliability issues.
- Compatibility: Dual-mode operation can create compatibility issues.
- Development complexity: Dual-mode operation requires a higher level of technical expertise and development skills.
- Maintenance complexity: Dual-mode operation can make maintenance and support more complex.

What is meant by microkernel architecture? What are its advantages? It is suitable for which environment.

Microkernel Architecture –
Since the kernel is the core part of the operating system, so it is meant for handling the most important services only. Thus, in this architecture, only the most important services are inside the kernel and the rest of the OS services are present inside the system application program. Thus, users can interact with those not-so-important services within the system application. And the microkernel is solely responsible for the most important services of the operating system they are named as follows:

- Inter process-Communication.
- Memory Management
- CPU-Scheduling

Advantages of Micro-Kernel architecture:

- Modularity: greater modularity due to independent development and maintenance of kernel and servers
- Fault isolation: isolation of faults to prevent affecting the entire system.
- Performance: improved performance due to kernel containing only essential functions
- Security: improved security due to reduced attack surface
- Reliability: increased reliability due to less complexity
- Scalability: easily scalable to support different hardware architectures
- Portability: easily portable to different platforms with minimal effort

Suitable for which environment:

Microkernel architecture is suitable for environments that require high levels of modularity, security, reliability, and scalability. It is particularly useful in embedded systems, real-time systems, and other specialized applications where system stability and security are critical. Additionally, microkernel architectures can be useful in environments where hardware resources are limited, as they have a smaller footprint than monolithic kernels and can be optimized for specific hardware architectures.

The collection of tracks under the head at any time is known as ***cylinder.***

What is Inode? List 3 points about the information it contains.

In a Unix-style file system, an inode (short for index node) is a data structure that stores information about a file or directory. Each inode contains metadata about a single file system object, including its ownership, permissions, timestamps, and location on disk. The inode also contains a list of pointers to the blocks that make up the file's contents.

Information it contains:

1. File attributes: Inodes contain metadata about a file, including its permissions, ownership, size, timestamps (such as creation, modification, and access times), and file type.
2. Data block pointers: Inodes also contain pointers to the data blocks on disk that contain the actual contents of the file. For small files, these pointers may point directly to the data blocks, while for larger files, they may point to indirect blocks, double indirect blocks, or triple indirect blocks that in turn point to data blocks.
3. File system metadata: In addition to file-specific information, inodes also contain metadata about the file system itself, such as the total number of blocks and inodes, the number of free blocks and inodes, and the location of the root directory.

Differentiate between paging and segmentation.

S.NO	Paging	Segmentation
1.	In paging, the program is divided into fixed or mounted size pages.	In segmentation, the program is divided into variable size sections.
2.	For the paging operating system is accountable.	For segmentation compiler is accountable.
3.	Page size is determined by hardware.	Here, the section size is given by the user.
4.	It is faster in comparison to segmentation.	Segmentation is slow.
5.	Paging could result in internal fragmentation.	Segmentation could result in external fragmentation.
6.	In paging, the logical address is split into a page number and page offset.	Here, the logical address is split into section number and section offset.
7.	Paging comprises a page table that encloses the base address of every page.	While segmentation also comprises the segment table which encloses segment number and segment offset.
8.	The page table is employed to keep up the page data.	Section Table maintains the section data.
9.	In paging, the operating system must maintain a free frame list.	In segmentation, the operating system maintains a list of holes in the main memory.
10.	Paging is invisible to the user.	Segmentation is visible to the user.
11.	In paging, the processor needs	In segmentation, the processor uses

S.NO	Paging	Segmentation
	the page number, and offset to calculate the absolute address.	segment number, and offset to calculate the full address.
12.	It is hard to allow sharing of procedures between processes.	Facilitates sharing of procedures between the processes.
13	In paging, a programmer cannot efficiently handle data structure.	It can efficiently handle data structures.
14.	This protection is hard to apply.	Easy to apply for protection in segmentation.
15.	The size of the page needs always be equal to the size of frames.	There is no constraint on the size of segments.
16.	A page is referred to as a physical unit of information.	A segment is referred to as a logical unit of information.
17.	Paging results in a less efficient system.	Segmentation results in a more efficient system.

Analyse necessary conditions for deadlock.

Deadlock is a state in a system where each process is unable to proceed because it is waiting for a resource that is held by another process in the system. Analyzing the necessary conditions for deadlock can help understand and prevent such situations. The four necessary conditions for deadlock are:

1. Mutual Exclusion: At least one resource in the system must be non-shareable, meaning only one process can use it at a time. For example,

if two processes need exclusive access to a printer, only one process can use it at a given time. This condition ensures that once a process acquires a resource, it cannot be taken away until it is released voluntarily.

2. Hold and Wait: Processes in the system must be capable of holding resources while waiting for additional resources to become available. In other words, a process that is already holding resources can request additional resources without releasing its current ones. If a process is unable to acquire the additional resources it needs while holding others, it can result in a deadlock.

3. No Pre-emption: Resources cannot be forcibly taken away from a process. Only the process holding a resource can release it voluntarily. Pre-emption means that a resource can be forcibly taken from one process and given to another. By disallowing pre-emption, a process that is holding a resource can block other processes from accessing it, potentially leading to a deadlock.

4. Circular Wait: There must be a circular chain of two or more processes, where each process is waiting for a resource held by the next process in the chain. For example, process P1 is waiting for a resource held by process P2, which is waiting for a resource held by process P3, and so on, until the last process in the chain is waiting for a resource held by process P1. This circular dependency among processes creates a deadlock situation.

To avoid deadlock, at least one of these necessary conditions must not hold. Different strategies and algorithms can be employed to prevent or detect deadlock, such as resource allocation graphs, deadlock avoidance algorithms (e.g., Banker's algorithm), and deadlock detection algorithms (e.g., the use of resource allocation matrices and cycle detection).

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**

Proportional allocation – Allocate according to the size of process

- Dynamic as degree of multiprogramming, process sizes change
  - $s_i =$ size of process p_i
  - $S = \sum s_i$
  - $m =$ total number of frames
  - $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Number of Entries in Page Table-

Number of pages the process is divided

= Process size / Page size

Process Size-

Number of bits in virtual address space = 32 bits

Thus,

Process size

= 2^{32} B

Number of frames in main memory

= Size of main memory / Frame size

Page table size

= Number of entries in page table x Page table entry size

= Number of entries in page table x Number of bits in frame number

For Main Memory-

Physical Address Space = Size of main memory

Size of main memory = Total number of frames x Page size

Frame size = Page size

If number of frames in main memory = 2X, then number of bits in frame number = X bits

If Page size = 2X Bytes, then number of bits in page offset = X bits

If size of main memory = 2X Bytes, then number of bits in physical address = X bits

For Process-

Virtual Address Space = Size of process

Number of pages the process is divided = Process size / Page size

If process size = 2X bytes, then number of bits in virtual address space = X bits

For Page Table-

Size of page table = Number of entries in page table x Page table entry size

Number of entries in pages table = Number of pages the process is divided

Page table entry size = Number of bits in frame number + Number of bits used for optional fields if any

- **Calculate the number of bits required in the address for memory having size of 16 GB. Assume the memory is 4-byte addressable**.

Let ‘n’ number of bits are required. Then, Size of memory = 2^n x 4 bytes.
Since, the given memory has size of 16 GB, so we have-
2^n x 4 bytes = 16 GB
2^n x 4 = 16 G
$2^n \times 2^2 = 2^{34}$
$2^n = 2^{32}$
∴ n = 32 bits

Implementation of Page Table

Page table is kept in main memory

Page-table base register (**PTBR**) points to the page table

Page-table length register (**PTLR**) indicates size of the page table

In this scheme every data/instruction access requires two memory accesses

- One for the page table and one for the data / instruction

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

- Otherwise need to flush at every context switch

TLBs typically small (64 to 1,024 entries)

On a TLB miss, value is loaded into the TLB for faster access next time

- Replacement policies must be considered
- Some entries can be **wired down** for permanent fast access

Optimal Page Size

In paging scheme, there are mainly two overheads-

1. **Overhead of Page Tables-**

Paging requires each process to maintain a page table.

So, there is an overhead of maintaining a page table for each process.

2. **Overhead of Wasting Pages-**

There is an overhead of wasting last page of each process if it is not completely filled.

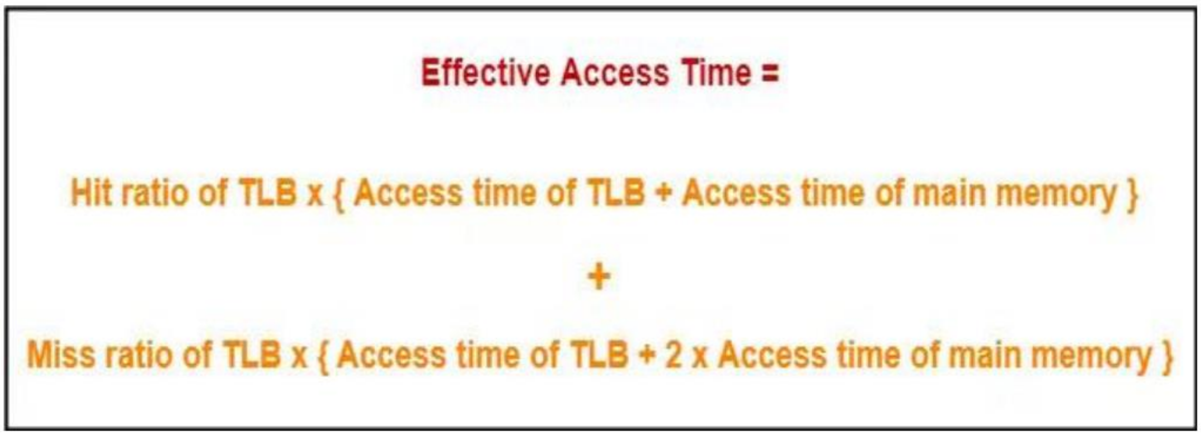
On an average, half page is wasted for each process.

Thus,

Total overhead for one process

= Size of its page table + (Page size / 2)

$$\boxed{\text{Optimal page size} = \sqrt{2 \times \text{Process size} \times \text{Page table entry size}}}$$



Effective Access Time =

Hit ratio of TLB x { Access time of TLB + Access time of main memory }

+

Miss ratio of TLB x { Access time of TLB + 2 x Access time of main memory }

Access Latency = **Average access time** = average seek time + average latency

Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead

Seek time ≈ seek distance

Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Seek Time:

Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

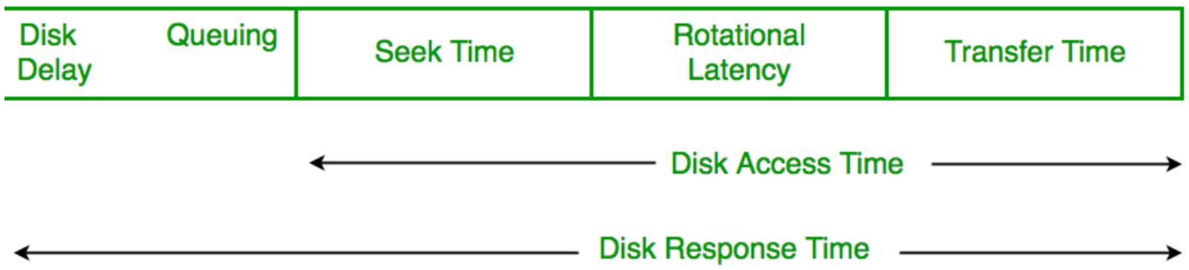
Rotational Latency:

Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

Transfer Time:

the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

- Disk Access Time = Seek Time + Rotational Latency + Transfer Time



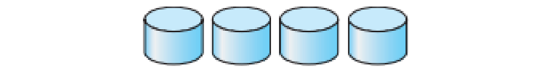
RAID Structure

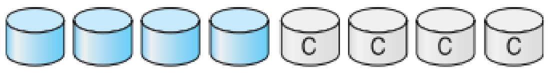
- RAID – redundant array of inexpensive disks
  - multiple disk drives provides reliability via **redundancy**
- Increases the **mean time to failure**
- **Mean time to repair –** exposure time when another failure could cause data loss
- **Mean time to data loss** based on above factors
- If mirrored disks fail independently, consider disk with 1300,000 mean time to failure and 10 hour mean time to repair
  - Mean time to data loss is $100,000^2 / (2 * 10) = 500 * 10^6$ hours, or 57,000 years!
- Frequently combined with **NVRAM** to improve write performance
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively

RAID (Cont.)

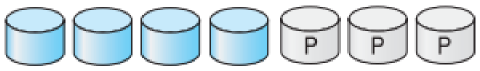
- Disk **striping** uses a group of disks as one storage unit
- RAID is arranged into six different levels
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
  - **Mirroring** or **shadowing** (**RAID 1**) keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity** (**RAID 4, 5, 6**) uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic **replication** of the data between arrays is common
- Frequently, a small number of **hot-spare** disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

RAID Levels

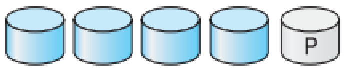

(a) RAID 0: non-redundant striping.



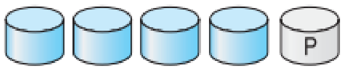
(b) RAID 1: mirrored disks.



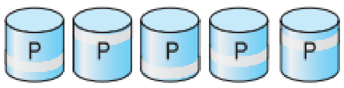
(c) RAID 2: memory-style error-correcting codes.



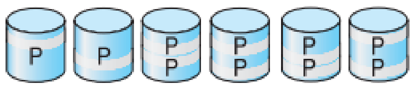
(d) RAID 3: bit-interleaved parity.



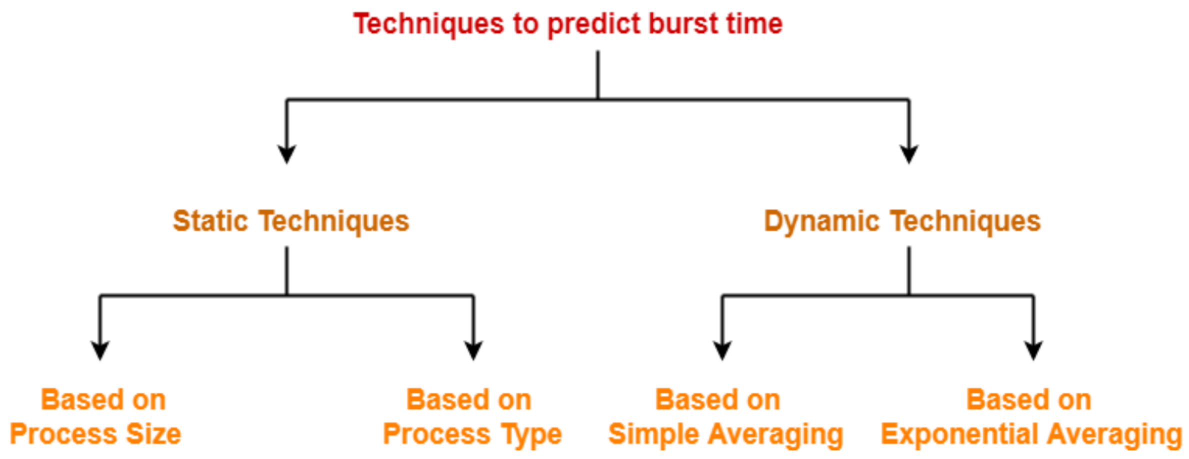
(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.


Techniques to predict burst time
Static Techniques
Dynamic Techniques
Based on Process Size
Based on Process Type
Based on Simple Averaging
Based on Exponential Averaging

1. Based on Process Size-

- This technique predicts the burst time for a process based on its size.
- Burst time of the already executed process of similar size is taken as the burst time for the process to be executed.
- Example-
- Consider a process of size 200 KB took 20 units of time to complete its execution.
- Then, burst time for any future process having size around 200 KB can be taken as 20 units.
- *NOTE*
  - *The predicted burst time may not always be right.*
  - *This is because the burst time of a process also depends on what kind of a process it is.*

Based on Simple Averaging-

Burst time for the process to be executed is taken as the average of all the processes that are executed till now.

Given n processes P_1, P_2, … , P_n and burst time of each process P_i as t_i, then predicted burst time for process P_{n+1} is given as-

$$\boxed{T_{n+1} = \frac{1}{n} \sum_{i=1}^{n} t_i}$$

2. Based on exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

$$\boxed{T_{n+1} = \alpha t_n + (1 - \alpha) T_n}$$

- where-
- α is called smoothening factor (0<= α <=1)
- t_n = actual burst time of process P_n
- T_n = Predicted burst time for process P_n
- Commonly, α set to ½

Practice for process synchronization:

https://questions.examside.com/past-years/gate/gate-cse/operating-systems/synchronization-and-concurrency

https://practicepaper.in/gate-cse/process-synchronization

https://www.computersciencejunction.in/2017/09/13/process-synchronization-based-questions-for-gate-html/

Memory management:

https://practicepaper.in/gate-cse/memory-management

https://www.computersciencejunction.in/2018/10/25/memory-management-gate-questions-and-answers-in-operating-system-html/

Deadlocks

https://practicepaper.in/gate-cse/deadlock

https://questions.examside.com/past-years/gate/gate-cse/operating-systems/deadlocks

https://www.gatevidyalay.com/deadlock-in-os-deadlock-problems-questions/

Process concepts and CPU scheduling

https://questions.examside.com/past-years/gate/gate-cse/operating-systems/process-concepts-and-cpu-scheduling

https://testbook.com/question-answer/consider-four-processes-p-q-r-and-s-scheduled-o--62171babffc71b6dee173b33

https://www.gatevidyalay.com/cpu-scheduling-practice-problems-numericals/

File system

https://questions.examside.com/past-years/gate/gate-cse/operating-systems/file-system-io-and-protection

https://testbook.com/question-answer/a-file-system-with-300-gbyte-disk-uses-a-file-desc--60925d286ba62d2dd254db09

https://gateoverflow.in/tag/file-system

Semaphores

Semaphore in OS | Practice Problems | Gate Vidyalay

https://gateoverflow.in/tag/semaphore

https://testbook.com/question-answer/the-following-program-consists-of-3-concurrent-pro--60b7113802405ba747c90946

https://www.youtube.com/watch?v=MRza1t01xgE&ab_channel=GeeksforGeeksGATEComputerScience

The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

```
P1( ) {
   C = B - 1;
   B = 2 * C;
}


P2( ) {
   D = 2 * B;
   B = D - 1;
}
```

The number of distinct values that B can possibly take after the execution is__________.

Initial value of B is 2 //value stored in main memory

<u>Value of B = 4</u>

First execute P2()

P2 ( ) {

D = 2 * B;   // 2 × 2 = 4

B = D – 1; // B = 4 – 1 = 3

}

write the value of B = 3 to memory

execute P1()

P1 ( ) {

C = B – 1; // C = 3 – 1 = 2

B = 2 * C; // B = 2 × 2 = 4

}

write the value of B = 4 is memory.

Distinct value of B is 4.

<u>Value of B =3</u>

First Execute P1()

```
P1 ( ) {
C = B – 1; // C = 2 – 1 = 1
B = 2 * C; // B = 2 × 1 =2
}
```

write the value of B = 2 to memory

then execute P2()

```
P2 ( ) {
D = 2 * B;   // 2 × 2 = 4
B = D – 1; // B = 4 – 1 = 3
}
```

write the value of B = 3 to memory

One of the distinct values of B is 3.

<u>Value of B = 2</u>

First execute P2()

```
P2 ( ) {
D = 2 * B;   // 2 × 2 = 4
B = D – 1; // B = 4 – 1 = 3
}

//don't write to memory

execute P1()  // with initial value of B = 2

P1 ( ) {

C = B – 1; // C = 2 – 1 = 1

B = 2 * C; // B = 2 × 1 = 2

}
```

write the value B = 3 by P2() in memory.

overwrite the value written by P2() as B = 2 by P1() in memory.

One of the distinct values of B is 2.

Therefore, the distinct values of B are 2, 3 and 4.

The number of distinct values that B can possibly take after the execution is 3.