Concurrency

Concurrency refers to the execution of multiple tasks or processes simultaneously. In a concurrent system, tasks are interleaved, which means they progress simultaneously but not necessarily at the same time. Concurrency is essential for improving the efficiency and responsiveness of applications, especially those that involve I/O operations or need to handle multiple user requests simultaneously.

Key Points:

□ **Context Switching**: The process of storing the state of a process or thread and restoring the state of another, allowing multiple processes to share a single CPU.
□ **Synchronization**: Mechanisms to control the access to shared resources by multiple threads to prevent data races and ensure data consistency. Common synchronization primitives include locks, semaphores, and monitors.

Threads

Threads are the smallest units of execution within a process. A single process can contain multiple threads, all of which share the same memory space but can execute independently. This makes threads a powerful tool for achieving concurrency.

Multithreading: The ability of a CPU, or a single core in a multi-core processor, to execute multiple threads concurrently. This can lead to better utilization of resources and improved application performance.

Threads API: The POSIX threads (pthreads) library in C provides a set of functions to create and manage threads. Key functions include:

- `pthread_create`: Creates a new thread.
- `pthread_join`: Waits for a thread to terminate.
- `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`: Functions to work with mutexes. **Common Concurrency Problems**

Race Conditions: • Occur when multiple threads access and modify shared data concurrently.

- The outcome depends on the timing of the threads' execution, leading to unpredictable results.

- Example: Two threads incrementing the same global counter without proper synchronization can lead to incorrect final values.

Deadlocks:

- Occur when two or more threads are waiting indefinitely for each other to release resources.
- Example: Thread A holds a lock on resource X and waits for resource Y, while Thread B holds a lock on resource Y and waits for resource X.

Resource Starvation:

- Occurs when a thread is perpetually denied access to resources because other threads are constantly using them.
- Example: A low-priority thread never gets CPU time because higher-priority threads keep occupying the CPU.

Locked Data Structures: Locked data structures are designed to be accessed safely by multiple threads concurrently. They achieve this by using synchronization mechanisms like mutexes to ensure that only one thread can modify the data at a time, preventing race conditions and data corruption.

- **Examples of Locked Data Structures**:
  - **Thread-Safe Queue**: A queue that uses a mutex to protect its `enqueue` and `dequeue` operations. This ensures that only one thread can modify the queue at a time.

Condition Variables: Condition variables are synchronization primitives that allo+w threads to wait for certain conditions to be true. They are typically used in conjunction with a mutex to coordinate the execution of threads.

- **Usage**:
  - **Wait**: A thread can wait for a condition to be met using `pthread_cond_wait`. This function atomically releases the associated mutex and puts the thread to sleep until the condition is signaled. o **Signal**: Another thread can signal that the condition has been met using

    `pthread_cond_signal` or `pthread_cond_broadcast`. `pthread_cond_signal` wakes up one waiting thread, while `pthread_cond_broadcast` wakes up all waiting threads.
- **Key Functions**:
  - `pthread_cond_wait`: Waits for a condition to be signaled.
  - `pthread_cond_signal`: Signals one waiting thread. o `pthread_cond_broadcast`: Signals all waiting threads.

Mutex (Mutual Exclusion): A mutex is a synchronization primitive that ensures mutual exclusion. It allows only one thread to execute a critical section of code at a time, preventing race conditions.

- **Usage**:
  - **Lock**: A thread acquires the mutex before entering the critical section using `pthread_mutex_lock`. o **Unlock**: The thread releases the mutex after exiting the critical section using `pthread_mutex_unlock`.
- **Redundant Disk Arrays (RAID)**
- RAID (Redundant Array of Independent Disks) is a technology that combines multiple physical disk drives into a single logical unit for data redundancy or performance improvement.

`Disk scheduling algorithms:` https://www.geeksforgeeks.org/disk-scheduling-algorithms/

Data Integrity Protection

Data integrity ensures that data is accurate and consistent over its lifecycle. Key mechanisms include:

- **Checksums**: Simple error-detection schemes that use algorithms to create a unique value based on data. When data is read or transmitted, the checksum is recalculated and compared to detect errors.
- **Error Detection Codes (EDC)**: Techniques like parity bits, Hamming codes, and cyclic redundancy checks (CRC) used to detect and correct errors in data.
- **Validation Processes**: Procedures to ensure data meets predefined criteria. Examples include format validation (e.g., checking if an email address is valid) and consistency checks (e.g., verifying that related data fields align).

Real-Time Systems

Real-time systems are designed to respond to inputs or events within a specific time frame. They are classified into:

- **Hard Real-Time Systems**: Missing a deadline can cause catastrophic failure. Examples include pacemakers and automotive airbag systems.
- **Soft Real-Time Systems**: Missing a deadline results in degraded performance but not catastrophic failure. Examples include video streaming and online gaming.

Real-Time Kernels: Specialized operating systems or kernel extensions that provide deterministic scheduling and low-latency features required for real-time applications.

Implementing Real-Time Operating Systems

Implementing a real-time operating system involves:

Real-Time CPU Scheduling:

- **Priority-Based Scheduling**: Tasks are assigned priorities, and the scheduler ensures high-priority tasks are executed first.
- **Deadline-Based Scheduling**: Tasks are scheduled based on their deadlines, ensuring they complete within specified time constraints.

Protection Mechanisms:

- **Access Control**: Ensures that only authorized tasks or users can access specific resources or perform certain actions.
- **Memory Protection**: Prevents tasks from accessing or modifying each other's memory, ensuring system stability and integrity.

Example: Real-Time Task Scheduling in C:

```
#include <stdio.h>

#include <pthread.h> #include

<unistd.h> void*

realTimeTask(void* arg) {

   while (1) {        printf("Real-time task executing...\n");

usleep(1000000); // Simulate task execution time (1 second)

   }

   return NULL;

} int main() {     pthread_t rtThread;

pthread_create(&rtThread, NULL, realTimeTask, NULL);

pthread_join(rtThread, NULL);

   return 0;
```

}

Q6: Consider a user program of logical address of size **6** pages and page size is **4** bytes. The physical address contains **300** frames. The user program consists of **22** instructions **a, b, c, . . . u, v .** Each instruction takes **1 byte**. Assume at that time the free frames are **7, 26, 52, 20, 55, 6, 18, 21, 70,** and **90**.

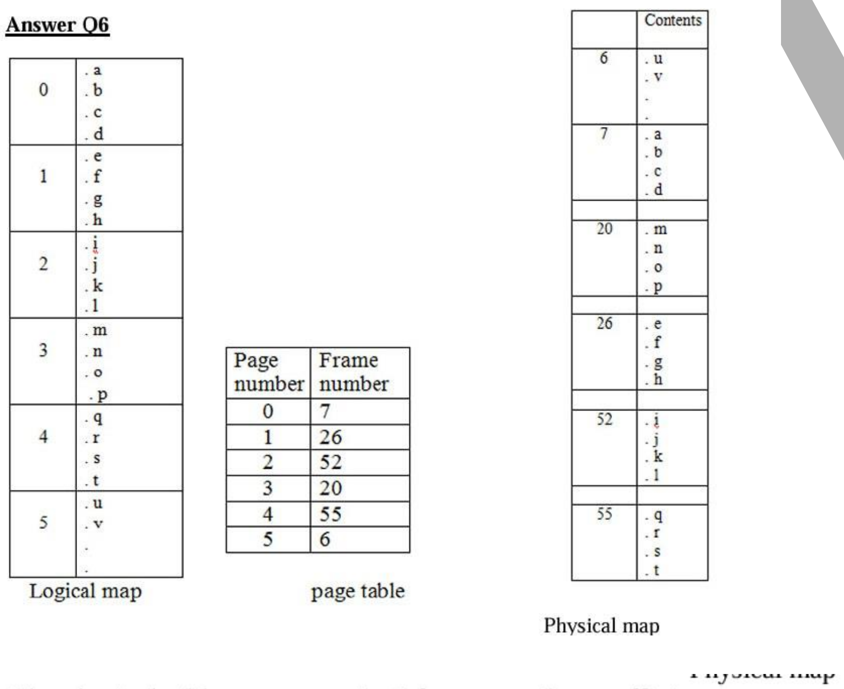
Find the following? (10 degrees)

A) Draw the logical and physical maps and page tables?

B) Allocate each page in the corresponding frame?

C) Find the physical addresses for the instructions **m, d, v, r**?

D) Calculate the fragmentation if exist?



Answer Q6

0	. a . b . c . d
1	. e . f . g . h
2	. i . j . k . l
3	. m . n . o . p
4	. q . r . s . t
5	. u . v . .

Logical map

Page number	Frame number
0	7
1	26
2	52
3	20
4	55
5	6

page table

	Contents
6	. u . v . .
7	. a . b . c . d
20	. m . n . o . p
26	. e . f . g . h
52	. i . j . k . l
55	. q . r . s . t

Physical map

Physical map

The physical address = page size * frame number + offset

The physical address of m = 4*20 +0 = 80

The physical address of d = 4*7+3 = 31

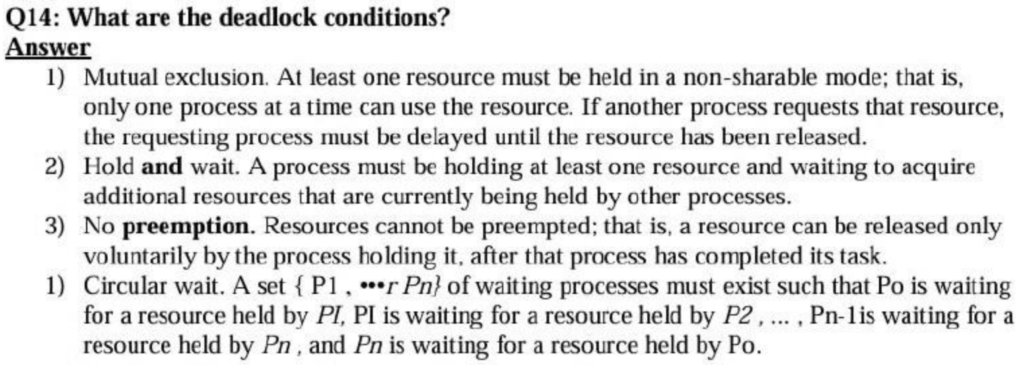
The physical address of v = 4* 6 +1 = 25

The physical address of r = 4*55+ 1= 221

The external fragmentation = 0

The internal fragmentation = 2

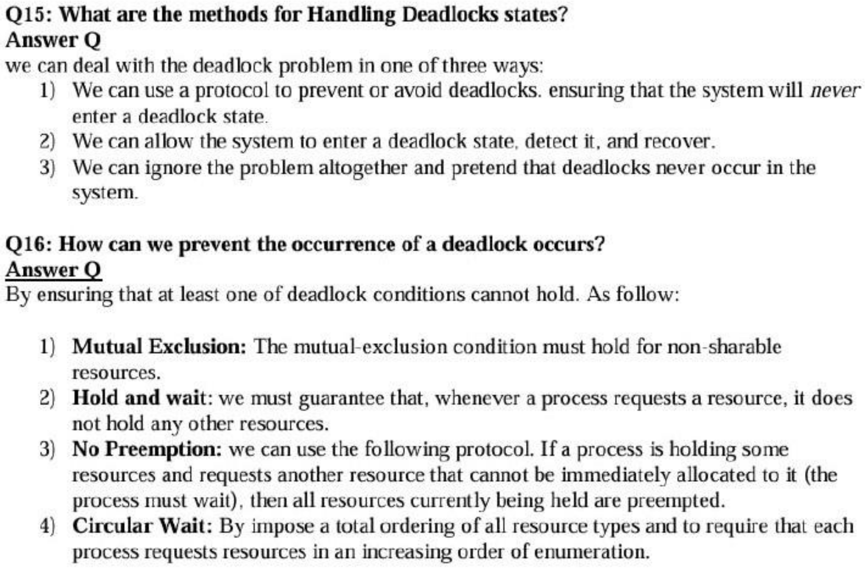
**IMP



Q14: What are the deadlock conditions?

Answer

1) Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2) Hold **and** wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3) No **preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
1) Circular wait. A set { P1 , •••*r Pn}* of waiting processes must exist such that Po is waiting for a resource held by *PI,* PI is waiting for a resource held by *P2* , ... , Pn-1is waiting for a resource held by *Pn* , and *Pn* is waiting for a resource held by Po.



Q15: What are the methods for Handling Deadlocks states?

Answer Q

we can deal with the deadlock problem in one of three ways:

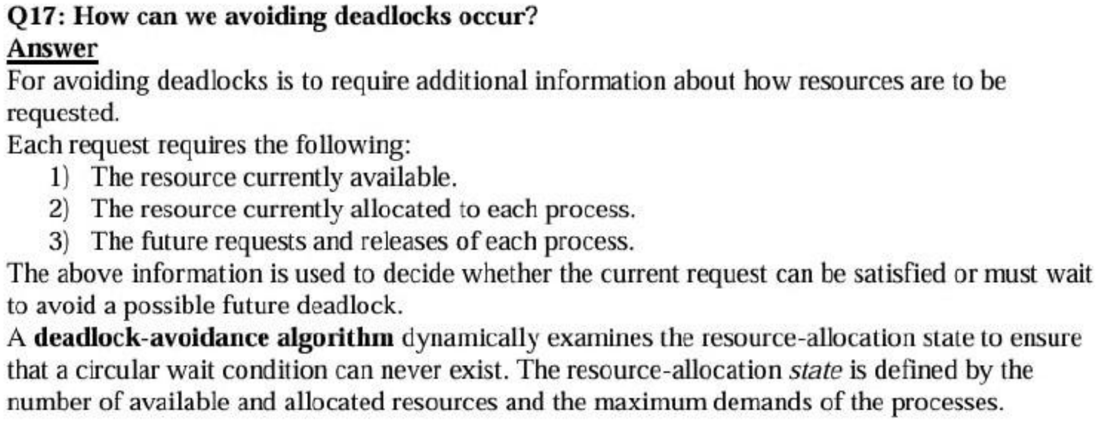
1) We can use a protocol to prevent or avoid deadlocks. ensuring that the system will *never* enter a deadlock state.
2) We can allow the system to enter a deadlock state, detect it, and recover.
3) We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Q16: How can we prevent the occurrence of a deadlock occurs?

Answer Q

By ensuring that at least one of deadlock conditions cannot hold. As follow:

1) **Mutual Exclusion:** The mutual-exclusion condition must hold for non-sharable resources.
2) **Hold and wait**: we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
3) **No Preemption:** we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (the process must wait), then all resources currently being held are preempted.
4) **Circular Wait:** By impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.



Q17: How can we avoiding deadlocks occur?

Answer

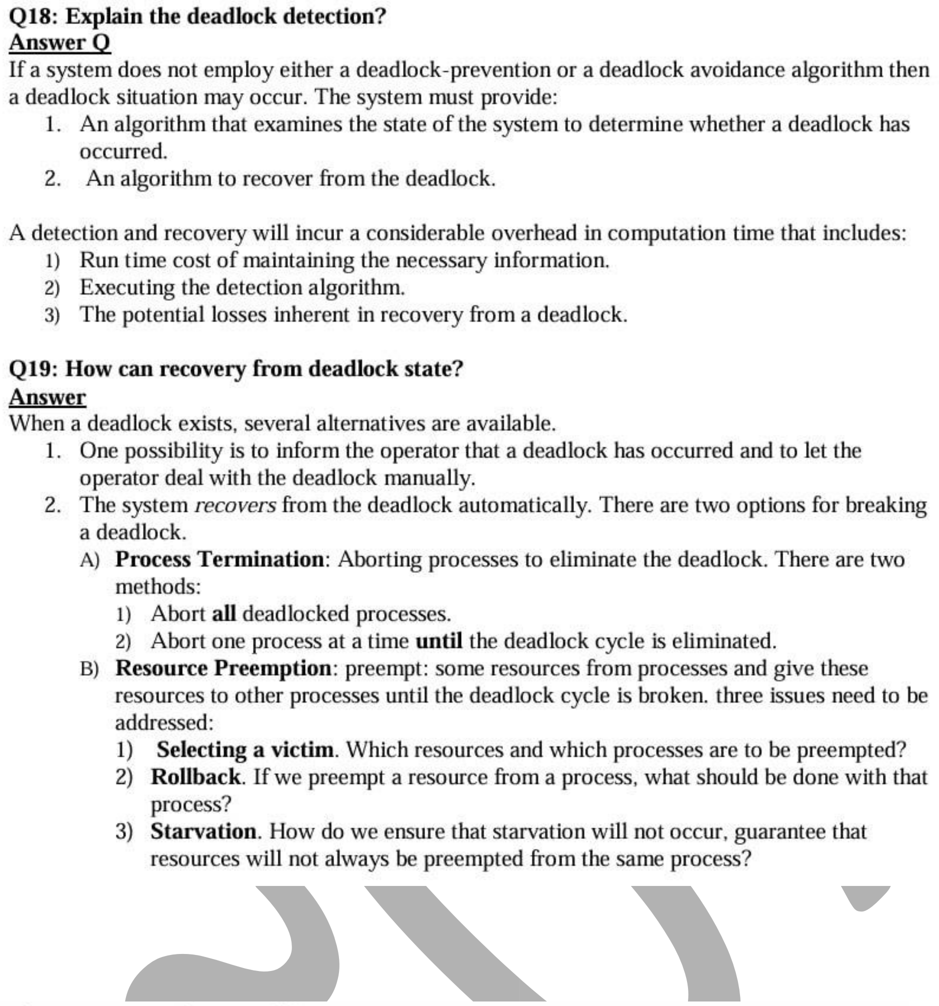
For avoiding deadlocks is to require additional information about how resources are to be requested.

Each request requires the following:

1) The resource currently available.
2) The resource currently allocated to each process.
3) The future requests and releases of each process.

The above information is used to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

A **deadlock-avoidance algorithm** dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes.



Q18: Explain the deadlock detection?

Answer Q

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm then a deadlock situation may occur. The system must provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred.
2. An algorithm to recover from the deadlock.

A detection and recovery will incur a considerable overhead in computation time that includes:

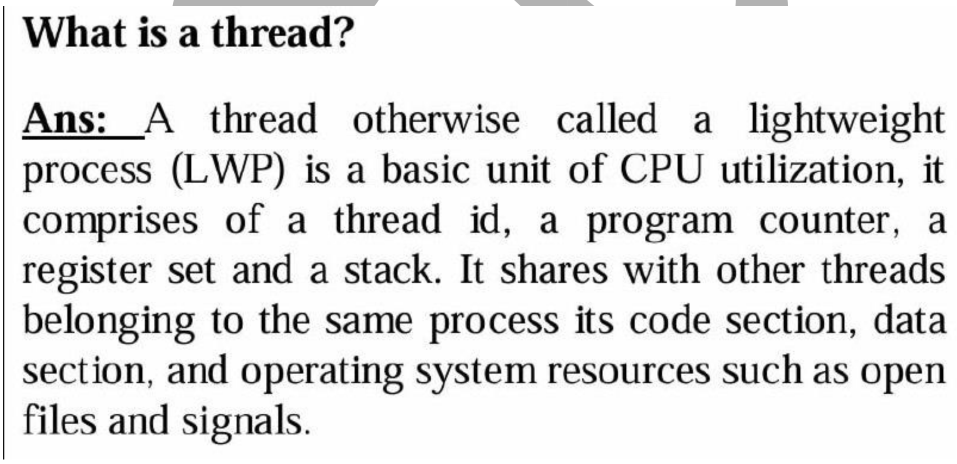
1) Run time cost of maintaining the necessary information.
2) Executing the detection algorithm.
3) The potential losses inherent in recovery from a deadlock.

Q19: How can recovery from deadlock state?

Answer

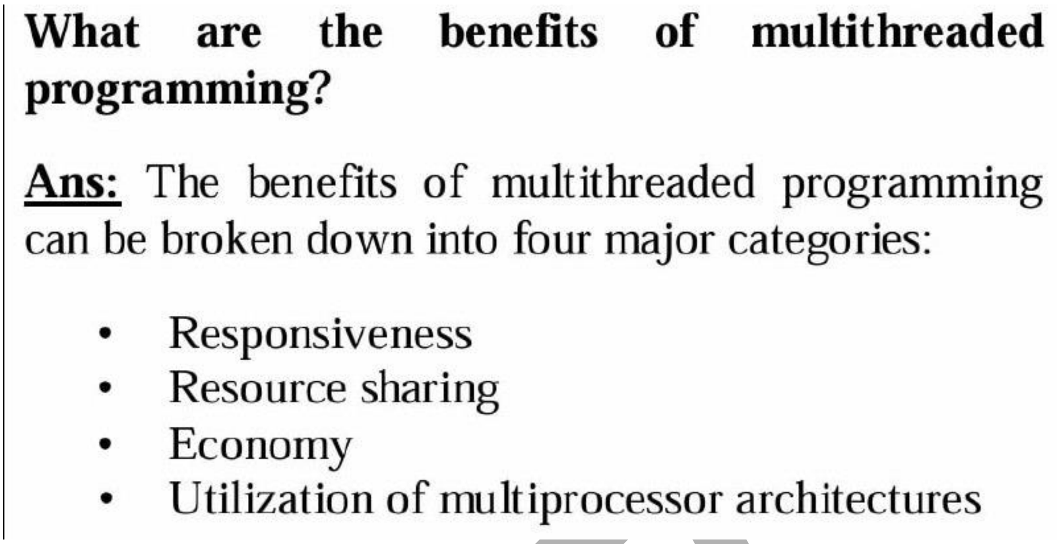
When a deadlock exists, several alternatives are available.

1. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
2. The system *recovers* from the deadlock automatically. There are two options for breaking a deadlock.
   A) **Process Termination**: Aborting processes to eliminate the deadlock. There are two methods:
      1) Abort **all** deadlocked processes.
      2) Abort one process at a time **until** the deadlock cycle is eliminated.
   B) **Resource Preemption**: preempt: some resources from processes and give these resources to other processes until the deadlock cycle is broken. three issues need to be addressed:
      1) **Selecting a victim**. Which resources and which processes are to be preempted?
      2) **Rollback**. If we preempt a resource from a process, what should be done with that process?
      3) **Starvation**. How do we ensure that starvation will not occur, guarantee that resources will not always be preempted from the same process?



What is a thread?

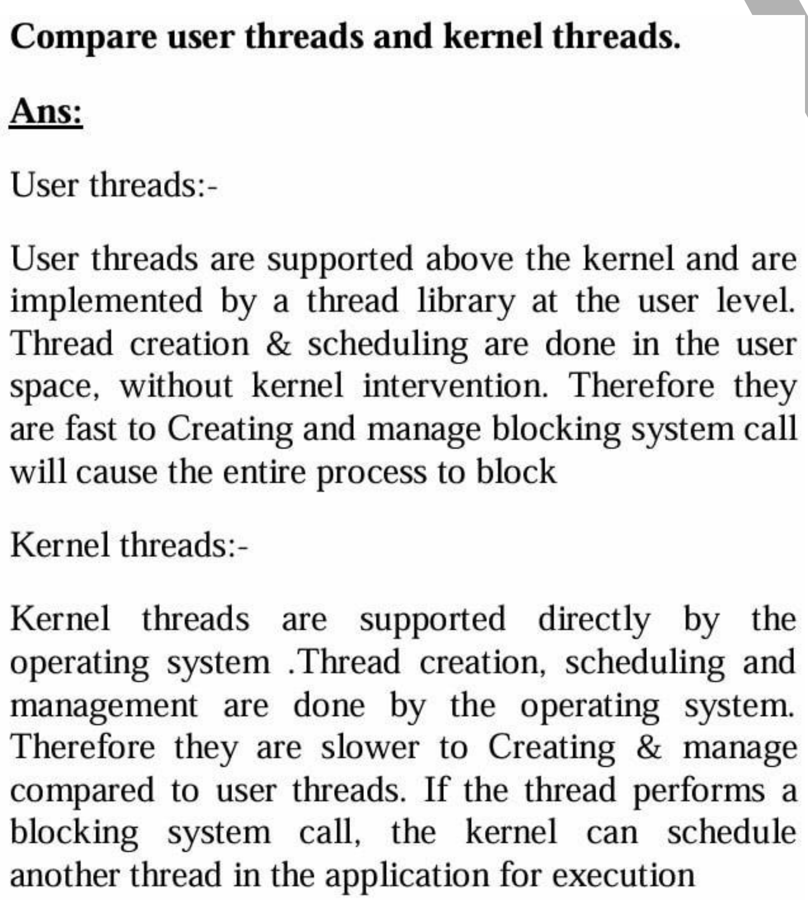
Ans: A thread otherwise called a lightweight process (LWP) is a basic unit of CPU utilization, it comprises of a thread id, a program counter, a register set and a stack. It shares with other threads belonging to the same process its code section, data section, and operating system resources such as open files and signals.



What are the benefits of multithreaded programming?

<u>Ans:</u> The benefits of multithreaded programming can be broken down into four major categories:

- Responsiveness
- Resource sharing
- Economy
- Utilization of multiprocessor architectures



Compare user threads and kernel threads.

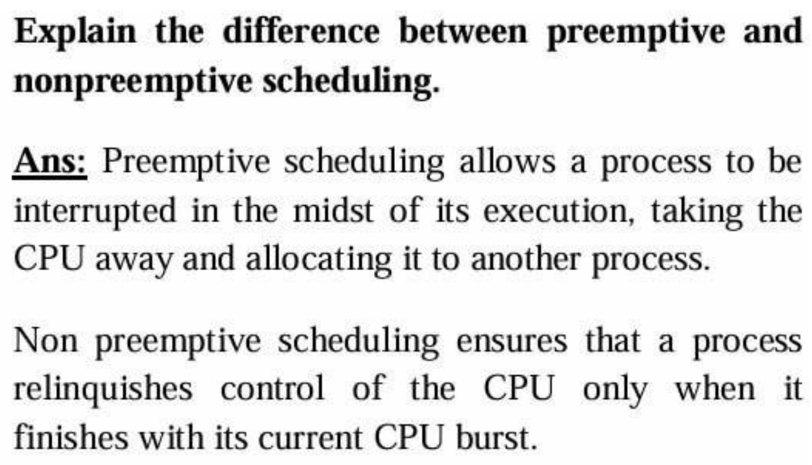
<u>Ans:</u>

User threads:-

User threads are supported above the kernel and are implemented by a thread library at the user level. Thread creation & scheduling are done in the user space, without kernel intervention. Therefore they are fast to Creating and manage blocking system call will cause the entire process to block

Kernel threads:-

Kernel threads are supported directly by the operating system .Thread creation, scheduling and management are done by the operating system. Therefore they are slower to Creating & manage compared to user threads. If the thread performs a blocking system call, the kernel can schedule another thread in the application for execution

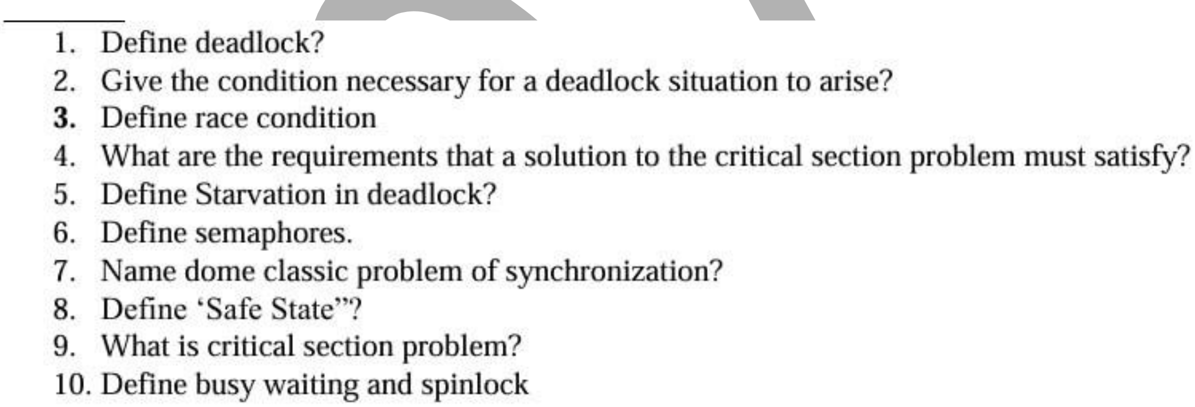


Explain the difference between preemptive and nonpreemptive scheduling.

Ans: Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.

Non preemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

CO-3 IMPORTANT QUESTIONS



1. Define deadlock?
2. Give the condition necessary for a deadlock situation to arise?
3. Define race condition
4. What are the requirements that a solution to the critical section problem must satisfy?
5. Define Starvation in deadlock?
6. Define semaphores.
7. Name dome classic problem of synchronization?
8. Define 'Safe State"?
9. What is critical section problem?
10. Define busy waiting and spinlock

CO-4 IMP QUESTIONS

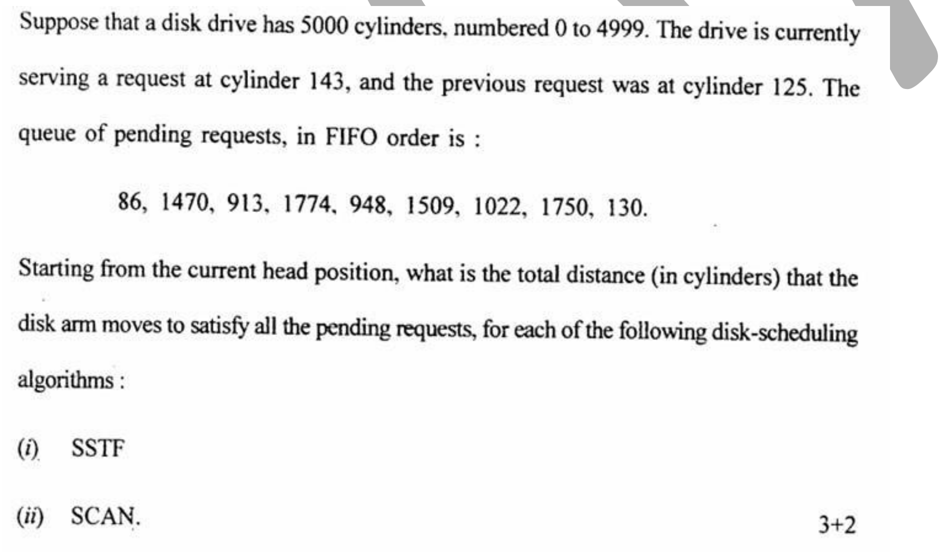
1. What is critical section problem? Explain with example?
2. What is Semaphore? Explain producer consumer problem using semaphore?
3. Define process synchronization and explain Peterson solution algorithms?
4. What is Monitor? Explain with any example using monitor?
5. Explain the solution for Dining-Philosophers Problem
6. a) What are the methods for handling deadlock.

   b) Write about deadlock and starvation?
7. a) Explain about Deadlock Avoidance?

   b) Explain how recovery from deadlock?
8. Explain Dead lock detection (Banker's Algorithm) with Example?
9. Write about Deadlock Prevention Methods?
10. Discuss about the following

    A) Semaphore

    B) Monitor



Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order is :

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk-scheduling algorithms :

(*i*) SSTF

(*ii*) SCAN. 3+2

Other resources:

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

practice all disk scheduling algorithms