Assignment 4 :-
Synchronization Problems

- Solve the following problems :-

1] Reader writer problem using semaphore

Ans :- The reader-writer problem can be solved using semaphores as follows :-

1] Initilize two semaphore : "mutex" and "read_count". "mutex" controls access to the shared resources, and "read_count" keeps track of the number of readers currently accessing resource.

2] When a reader wants to access the shared resource, it first acquires the "mutex" semaphore, increements the "read_count" semaphore, releases the "mutex" semaphore, and then proceeds to read from shared resources.

3] When reader is done reading, it acquires the "mutex" semaphore again, decrements the "read_count" semaphore and releases "mutex" semaphore.

4] When a writer wants to access shared resource, it acquires the "mutex" semaphore,

DATE / /

and waits untill the "read_count" semaphore is zero.

5. Once "read_count" semaphore is zero, writer proceeds to write to shared resource, releases the "mutex" semaphore and then proceeds to read from shared resource.

- This soln ensures that muliple readers can access shared resource simultaneously, but a writer must wait until all readers are done before it can access resource.

2] Solve reader writer problem using mutex.

- The reader-writer problem can also be solved using mutex locks, which are on a way of controlling access to shared resources in a multi-threaded or multi-process envieonment.

- mutex lock has two states : locked and unlocked. When a thread acquires a locked mutex, it blocks untill mutex is unlocked by another thread.

- To solve reader-writer problem using mutex locks, you can use two mutex locks: one to control access to shared resource and one to keep track of number of readers currently accessing the resource.

1] Initialize two mutex locks: "resource_mutex" and "read_count_mutex". "resource_mutex" controls access to shared resource and "read_count_mutex" controls access to read_count variable.

2] When a reader wants to access the shared resource it first acquires "read_count_mutex" lock and then it acquires the "resource_mutex" lock, reads from the shared resource and releases "resource_mutex" lock.

3] When reader is done reading, it acquires "read_count_mutex" lock, decrements the read_count variable, releases the "read_count_mutex" lock.

4] When a writer wants to access the shared resource, it acquires the "resource_mutex" lock, and waits until read_count variable is zero.

5] Once read_count variable is zero, writer proceeds to write to shared resource, releases "resource_mutex" lock, and proceeds to read from shared resource.

This soln ensures multiple readers can access shared resource simultaneously, but a writer must wait until all readers are done before it can access resource.

DATE / / /

3] Solve Producer Consumer problem using Semaphore.

- Producer-consumer problem is a classic example of a multi-process synchronization problem. It can be solved using semaphores, which are a way of controlling access to shared resources in a multi-threading or multi-process environment.

- A semaphore is a variable that is used to control access to a shared resource. It has two operations: wait & signal. The wait operation decrements the semaphore value and signal operation increments it.

- In producer-consumer problem, there are multiple producers and consumers that need to access a shared buffer. The soln is to use two semaphores: one to control access to shared buffer and one to keep track of the number of items in buffer.

- When producer wants to add an item to buffer, it first acquires "empty" semaphore, adds item to buffer, releases the "empty" semaphore, and signals "full" semaphore.

- When a consumer wants to remove an item from the buffer, it first acquires 'full' semaphore removes an item from buffer, releases 'full' semaphore, and signals "empty" semaphore.

- The "empty" semaphore is initialized to size of buffer, and "full" semphore is initialized to 0.

- ~~The~~ 'empty' ~~semaphore~~
- This soln ensures that producers and blocked when buffer is full, and consumers are blocked when buffer is empty and thus avoided the buffer overflow and underflow situations.

- It is important to note that all semaphores should be initialized to appropriate value and that wait and signals operations should be implemented using atomic operations to avoid race condn.

4] Producer consumer problem using mutex

- The producer-consumer problem can also be solved using mutex locks, which are a way of controlling access to shared resources in a multithreaded or multi-process environment.

- A mutex lock has two states: locked & unlocked When a thread acquires a locked mutex it blocks until the mutex is unlocked by another thread.

- To solve producer-consumer problem using mutex locks, you can two mutex locks:
  - one to control access to shared buffer and one to control access to the buffer size variable.

You will need to use condn variables to synchronize the producer & consumer threads.

1. Initialize two mutex locks: "buffer_mutex" and count_mutex" and two condn variables: "not_full" and "not_empty".
"buffer_mutex" controls access to shared buffer and 'count_mutex" controls access to buffer size variable.

2. When a producer wants to add an item to buffer, it first acquires "count_mutex' lock, waits until buffer size is less than the maximum capacity, release "count_mutex" lock, acquires "buffer_mutex" lock, add items to buffer, release "buffer_mutex" lock and signals 'not_empty" condn variable.

3. When a consumer wants to remove an item from buffer, it first acquire "count_mutex" lock, waits until buffer size is greater than 0, release 'countor_mutex' lock, acquires "buffer_mutex", removes an item from buffer release "buffer_mutex" lock and signal "not_full" condn variable.

- This soln ensures that producers are blocked when buffer is full, & consumers are blocked when buffer is empty, & thus avoiding buffer overflow & underflow situations.
- Additionally, it's imp to note that all mutex locks should be initialized to an unlocked state, and the lock & unlock operations should be implemented using atomic operations to avoid race condn.
- It's imp that to note that order of acquiring locks is important to avoid deadlocks. And also that the condn variable should be used in conjuction with mutex lock to avoid race condn on buffer size variable.

5] Solve Dinning Philosophers problem. using semphore

- The dining philosophers problem is a classic problem in a field of concurrent programming & synchronization. It can be solved using semaphores, which are a way of controlling access to shared resources in a multi-threaded or multi-process environment.

- A semaphore is a variable that is used to control access to a shared resource. It has two operations: wait & signal. The wait operation decrements semaphore value & signal operation increment it.

- In Dining philosophers problem, there are five philosophers sitting around a table, each with their own plate of food and fork to the left & right of them. The soln is to use five semaphores, one for each philosopher, to control access to forks.

- When a philosopher wants to eat, it first acquires the semaphore for fork to its left, then it acquires the semaphore for fork to its right, eats releases the semaphore for fork to its left & release semaphore for fork to its right

- This soln ensures that a philosopher can only eat if it has acquired both left & right fork semaphores, & preventing deadlock situation where all philosophers are waiting on a fork to be ~~eated~~ released.