* Critical Section Problem and How to address it

- Process Synchronization techniques play a key role in maintaining the consistency of shared data.

- **Critical Section (C.S.).**

The critical section refers to the segment of code where processes / threads access shared resources, such as common variables and files, and perform write operations on them. Since processes / threads execute concurrently, any process can be interrupted mid-execution.

- **Major Thread scheduling issue**

(a) **Race Condition:-**

A condition occurs when two or more threads can access shared data and they try to change it at the same time. Because

Page No.:
Date:

the thread scheduling algorithm can swap between threads at any time. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads "racing" to access/change the data.

- Solution to Race condition:-

(a) Atomic operations :- Make critical code section an atomic operation, i.e., Executed in one CPU cycle.

(b) Mutual Exclusion using locks.

(c) Semaphores.

(8) Can we use a simple flag variable to solve the problem of race condition?

Ans No, as

- Solution of critical section should have 3 conditions:-

(1) Mutual exclusion (If one thread goes to critical section, other should wait until it finishes it's execution).

(2) Progress :- (There should be not any order in which thread goes to an critical section, like one thread T1 depends on T2 to go in critical section and it goes. One thread goes in any random order).

(3) Bonded Waiting :- (No thread will wait

indefinitely to go for execution in critical section. There it must be limited Waiting time.

- Point (1) and (2) are mandatory for an solution while (3) is optional.

- Now, Why simple flag is not sufficient here?

:- Turn = 0.

Thread 1	Thread 2
While (1)	While (1)
{	{
While (turn != 0);	While (turn != 1);
C.S.	C.S.
turn = 1	Turn = 0
R.S	R.S
}	}

Here mutual exclusion is achieved as one only one thread is executed and other wait, but the order of thread execution depends on turn initial values, so the point (2) is not followed here, so that's, Why flag is not the correct option here.

- An updated solution is Peterson's solution,

Page No.:
Date:

- **Peterson's Solution :-**

A Software-based algorithm for two processes that ensures mutual exclusion, progress and bounded Waiting when they need to access a shared resource in their critical section. It uses two shared variables - a boolean "flag" array and an integer "turn" variable. Applicable only for 2 threads.

- **Mutex / Locks**

(a) Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread / process to access critical section.

(b) **Disadvantages :-**

(i) Contention ⇒ One thread has acquired the lock, other threads will be busy waiting, What if thread that had acquired the lock dies, then other thread will be wait for infinite.

(ii) Deadlock ⇒ A Situation Where each participant is Waiting for a resource held by another participant in the group, creating a circular dependency.