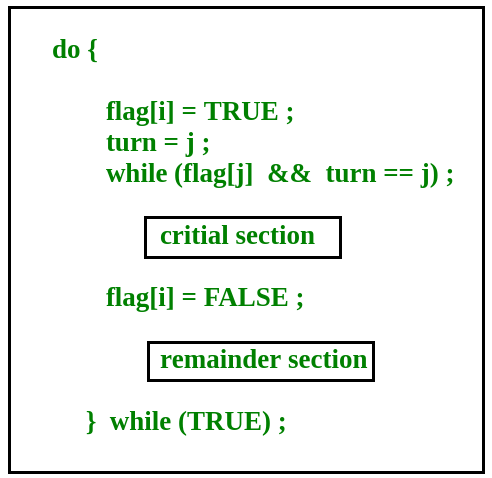
**Unit – III Process Synchronization**

### Peterson Solution

Peterson's solution is widely used solution to critical section problems. In this solution, when a process is executing in a critical state, then the other process only executes the rest of the code, and the opposite can happen. This method also helps to make sure that only a single process runs in the critical section at a specific time.

In Peterson’s solution, we have two shared variables:

* boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
* int turn : The process whose turn is to enter the critical section.



Peterson’s Solution preserves all three conditions :

* Mutual Exclusion is assured as only one process can access the critical section at any time.
* Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
* Bounded Waiting is preserved as every process gets a fair chance.

### Synchronization Hardware

Some times the problems of the Critical Section are also resolved by hardware. Some operating system offers a lock functionality where a Process acquires a lock when entering the Critical section and releases the lock after leaving it.

So when another process is trying to enter the critical section, it will not be able to enter as it is locked. It can only do so if it is free by acquiring the lock itself.

### Mutex Locks

Synchronization hardware not simple method to implement for everyone, so strict software method known as Mutex Locks was also introduced.

In this approach, in the entry section of code, a LOCK is obtained over the critical resources used inside the critical section. In the exit section that lock is released.

**TestAndSet**  
TestAndSet is a hardware solution to the synchronization problem. In TestAndSet, we have a shared lock variable which can take either of the two values, 0 or 1.

Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting till it becomes free and if it is not locked, it takes the lock and executes the critical section.

In TestAndSet, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

Entry Region :

tsl reg, flag

cmp reg, #0

jnz entry region

ret

Leaving Region:

move flag, #0

ret

### Semaphores:

Semaphore is simply a variable that is non-negative and shared between threads. It is another algorithm or solution to the critical section problem. It is a signaling mechanism and a thread that is waiting on a semaphore, which can be signaled by another thread.

It uses two atomic operations, wait, and signal for the process synchronization.

WAIT ( S ):

while ( S <= 0 );

S = S - 1;

SIGNAL ( S ):

S = S + 1;

There are two types of semaphores : Binary Semaphores and Counting Semaphores

* + Binary Semaphores : They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.
  + Counting Semaphores: They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.