Give an solution to critical section problem ?

- **Mutual Exclusion** : If process P_i is executing in its critical section, then no other processes can be executing in their critical sections **(before entering the critical section make lock in it and after finish working the critical section make unlock)**

- **Progress** : If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

- **Bounded Waiting** : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

What is the atomic instruction ?

It is the instruction that is, cannot be interrupted.

Explain Peterson's Solution ?

- **The two processes share two variables:**
  - **int turn** : indicates whose turn it is to enter the critical section.
  - **Boolean flag[2]** : array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
              critical section
        flag[i] = FALSE;
              remainder section
  } while (TRUE);
```

What is the solution of synchronization of hardware ?

- Uniprocessors – could disable interrupts

- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
- Operating systems using this not broadly scalable

What is the ways of atomic instruction ?

- test memory word and set value
- Or swap contents of two memory words

Explain the solution using lock ?

```
do {
acquire lock
critical section
release lock
remainder section
} while (TRUE);
```

Explain the solution using test and set ?

```
boolean TestAndSet (boolean *target)
        {
            boolean rv = *target;
            *target = TRUE;
            return rv:
        }
```

Shared boolean variable lock., initialized to false.

Solution:

```
                do {
                while ( TestAndSet (&lock ))
                        ;   // do nothing
                        //    critical section
                lock = FALSE;
                        //      remainder section
            } while (TRUE);
```

Explain the swap solution ?

```
void Swap (boolean *a, boolean *b)
        {
              boolean temp = *a;
              *a = *b;
        *b = temp:
    }
```

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
do {
        key = TRUE;
                while ( key == TRUE)
                Swap (&lock, &key );

            //   critical section
        lock = FALSE;
                //     remainder section
            } while (TRUE);
```

Explain Bounded-waiting Mutual Exclusion with TestandSet() solution ?

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
   } while (TRUE);
```

Explain semaphore ?

- It is Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
        wait (S) {
                while S <= 0
                        ; // no-op
S--;
}
        signal (S) {
S++;
}
```

What is type semaphore ?

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement , Also known as mutex locks

How Can implement a counting semaphore S as a binary semaphore ?

```
Semaphore mutex;    //  initialized to 1
do {
wait (mutex);
       // Critical Section
    signal (mutex);
// remainder section
} while (TRUE);
```

How can implement semaphore with out waiting ?

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.

- wakeup – remove one of processes in the waiting queue and place it in the ready queue.

How will be the implement of wait() and signal() of the previous solution ?

- **Implementation of wait:**

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- **Implementation of signal:**

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Gave example of software solution and hardware solution ?

❖ **Software solution :** semaphore
❖ **Hardware solution :** test and set - swap

What is the deadlock ?

two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

What is starvation ?

indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

What is Priority Inversion ?

Scheduling problem when lower-priority process holds a lock needed by higher-priority process