UNIT - III PART-II

SYNCHRONIZATION: TOOLS

Cooperating processes can either directly share a logical address space or be allowed to share data only through files / messages.

"Concurrent access to shared data may result in data inconsistency."

Synchronization

Synchronization is the process of coordinating the activities of multiple cooperating processes to ensure data consistency and prevent race condition.

Example:

Suppose value of "counter" variable is exactly 5. and the producer & consumer currently execute "counter++" and "counter--" statements. Though the correct result is 5 if both are executed seperately

Producer's Counter++

```
register1 = Counter
register1 = register1 + 1
Counter = register1
```

Consumer's Counter--

```
register2 = Counter
register2 = register2 - 1
Counter = register2
```

where register1 & register2 are local CPU registers.

But the concurrent execution of ~~one~~ "counter++" and "counter--" is equivalent to sequential execution of interleaved ~~execution~~ instructions in some arbitrary order.

One such interleaving is as follows:

		Counter
T_0 : Producer	execute register1 = counter	5
T_1 : Producer	execute register1 = register1 + 1	6
T_2 : Consumer	execute register2 = counter	5
T_3 : Consumer	execute register2 = register2 - 1	4
T_4 : Producer	execute counter = register1.	6
T_5 : Consumer	execute counter = register2	4

We have arrived at incorrect state counter = = 4. If T_4 & T_5 are interchanged, then counter = = 6.

** RACE CONDITION:

A situation where several processes access & manipulate the same data concurrently and the outcome of the execution depends on the particular order in which access takes place, is called race condition.

4.) CRITICAL SECTION PROBLEM:-

Consider a system consisting of n processes $P_0, P_1, \ldots P_{n-1}$. Each process has a segment of code called critical section in which the process may be changing common variables, updating a table, writing a file & so on.

The critical section problem is to design a protocol that the processes can use to cooperate such that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

General Structure of a typical process P_i

```
do
{
    entry section
        critical section
    exit section
        remainder section
} while(true);
```

Entry Section:- Each process must request permission to enter its critical section. The section of code implementing this request is entry section. The critical section is followed by an exit section & remaining code is in remainder section.

Solution to critical section problem must satisfy 3 conditions :-

(i) **Mutual Exclusion:-**
If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

(ii) **Progress:-**
If no process is executing in its critical section & some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, & this selection cannot be postponed indefinitely.

(iii) Bounded waiting:-

There exists a bound/limit on no. of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that request is granted.

APPROACHES TO HANDLE CRITICAL SECTION PROBLEM IN OPERATING SYSTEM:-

(i) PREEMPTIVE KERNEL:-

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

(ii) NON PREEMPTIVE KERNEL:-

A non-preemptive kernel doesnot allow a process running in kernel mode to be preempted.

(2) PETERSON'S SOLUTION:-

Peterson's solution is a classic software based solution to the critical section problem that addresses the requirements of mutual-exclusion, progress & bounded waiting.

Structure of Process P_i in Peterson's Solution.

```
do
{
    flag[i]=true;
    turn=j;
    while(flag[j] && turn==j);

        critical section

    flag[i]=false;

} while(TRUE);
```

Peterson's algorithm ensures 2 processes donot enter into critical section at the same time.
It has 2 components :-

* turn) variable that indicates whose turn is it to enter critical section.
* flag array contains boolean values of each process, indicating whether a process wants to enter critical section.

```
int turn;
boolean flag[2];
```

→ Peterson's solution is restricted to 2 process P_0, P_1 that alternate execution between their critical & remainder sections.

Algorithm for P_i process:

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

Algorithm for P_j process

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

* Initially both processes set their respective flag values to false, meaning no one wants to enter critical section.

To enter the critical section, Process P_i first sets flag[i] to be true &

then sets turn value to j, thereby stating if other process wishes to enter critical section, it can do so.

If both processes wish to enter at same time, turn will be set to both i & j (0 & 1) at same time but only 1 value lasts overwriting the other. So, eventual turn determines, which process enters critical section.

mutual exclusion preserved:-

P_i enters critical section only if either flag[j] = false or turn = i. Process P_0 & P_1 cannot enter critical section at same times as turn can be either 0 or 1 (i or j) but not both at same time.

Progress preserved:-

This solution says, though it wishes to enter critical section by making flag as true, it gives turn to other process

Bounded waiting preserved:-

As soon as P_i completes execution of critical section, it sets its flag to false allowing P_j to enter its critical section immediately. So, bounded wait is also achieved.

(3) MUTEX LOCKS :- Mutex - Mutual Exclusion -

Mutex locks is a simple higher-level software tool to solve critical section problem and prevent race condition.

A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

A mutex lock has a boolean variable available whose value indicates whether the lock is available or not.

```
while(true)
{
    acquire lock
        critical section
    release lock
        remainder section
}
```

SOLUTION TO CRITICAL SECTION PROBLEM USING MUTEX LOCKS

* The acquire() function acquires the lock. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.

⇒

```
acquire()
{
    while(!available);
        // busy wait.
    available = false;
}
```

* The release() function releases the lock

⇒

```
release()
{
    available = true;
}
```

Busy waiting:- (disadvantage)

While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). This is called busy waiting

Such type of mutex locks is called spinlock because the process spins while waiting for the lock to become available.

4. SEMAPHORES:-

* Its simple non-negative integer value shared between concurrent processes to solve critical section problem & to achieve process synchronization
* A semaphore S is an integer variable, that apart from initialization is accessed only by 2 standard atomic operations:-
  (i) wait() → denoted by P
  (ii) signal() → denoted by V.

(i) wait (P operation):-

This operation checks the semaphore's value. If the value is greater than 0, the process is allowed to continue, & value is decremented by 1.

If value is 0, the process is blocked (waits) until the semaphore value becomes greater than 0.

```
wait(S)
{
    while(S <= 0);
        // busy wait.
    S--;
}
```

(ii) Signal (V operation):-

After a process is done using shared resource, it performs signal operation. This increment the semaphore value by 1, potentially unblocking other waiting processes & allowing them to access the resources

```
signal(S)
{
    S++;
}
```

4.1. Semaphore Usage:-

Semaphores are of 2 types:-

(i) Binary Semaphore:-

This is also known as a mutex lock as they are locks that provide mutual exclusion. It can only have 2 values 0 & 1. Its value is initialized to 1.

(ii) Counting Semaphore:-

Counting semaphores can be used to control access to a given resource consisting of finite no. of ~~res~~ instances. The semaphore is initialized to the no. of resources available.

Each process that wishes to use a resource performs wait() operation on semaphore (decrements count)

When a process releases a resource, it performs a signal() operation (increments count)

When count for semaphore for semaphore goes to 0, all resources are being used.

After that, processes that wish to use a resource will block until the count becomes greater than 0.

Example:-

* Consider 2 process P_1 & P_2 with statements S_1 and S_2. P_1 & P_2 are concurrent processes & we require that S_2 be executed only after S_1 has completed.

We can implement this by letting P_1 & P_2 share a common semaphore synch initialized to 0.

```
S1;                          wait(synch);
signal(synch);               S2;
```

Process 1 Process 2

P_2 will executed S_2 only after P_1 invokes signal(synch) which is after statement S_1 has been executed.

* The same scenario can be used where 3 processes are P_1, P_2, P_3 are trying to share 2 instances of a resource. Here, semaphore can be initialized to zero. While a process is requesting a resource, wait() operation is used to check its availability & signal() operation is used when resource is to be released by the process

4.2. Semaphore implementation:-

The disadvange of wait() & signal() is busy waiting. To overcome this we need to modify the definition of wait() & signal().

When a semaphore value is not positive, a process is engaged in busy waiting. Instead, suspend the process placing it into waiting queue associated with semaphore, & state of the process is switched to the waiting state. Then control is transferred to CPU scheduler, which selects another process to execute.

Semaphore definition.

```
typedef struct
{
    int value;
    struct process *list;
} semaphore;
```

When a process executes the wait() operation & finds the semaphore value is not positive, it must wait.

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

A signal() operation removes one process from the list of waiting processes and awakens that process using wakeup.

```
signal (Semaphore *S)
{
    S->value++;
    if (S->value <= 0)
    {
        Remove a process from S->list;
        Wakeup(P)
    }
}
```