Process Synchronization(OS)

Process Synchronization is a technique which is used to coordinate the process that use shared Data. There are two types of Processes in an Operating Systems:-

1. **Independent Process –**
   The process that does not affect or is affected by the other process while its execution then the process is called Independent Process. Example The process that does not share any shared variable, database, files, etc.
2. **Cooperating Process –**
   The process that affect or is affected by the other process while execution, is called a Cooperating Process. Example The process

that share file, variable, database, etc are the Cooperating Process.

Process Synchronization is mainly used for Cooperating Process that shares the resources.

Objectives

1.To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.

2.To present both software and hardware solutions of the critical-section problem.

3.To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

Critical Section Problem

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

A diagram that demonstrates the critical section is as follows:

```
do {

        Entry Section

        Critical Section

        Exit Section

        Remainder Section

} while (TRUE);
```

OR

```
repeat
    entry section

    critical section

    exit section

    remainder section
```

until FALSE

Any synchronization process must satisfy the below given three conditions.

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many

other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

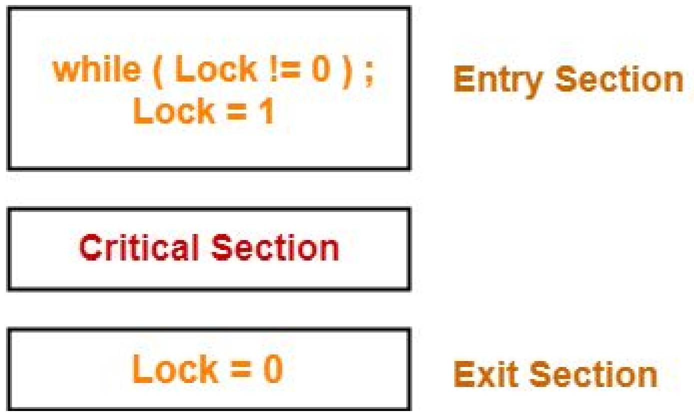
Mechanism to counter critical section problem--

1.Lock Variable-

- **Lock variable is a synchronization mechanism.**
- **It uses a lock variable to provide the synchronization among the processes executing concurrently.**

- **However, it completely fails to provide the synchronization.**

It is implemented as follows --



Initially, lock value is set to 0.

- **Lock value = 0 means the critical section is currently vacant and no process is present inside it.**
- **Lock Value=1 and a process is present inside it.**

- **Value = 1 means the critical section is currently occupied.**

But this process is not completely able to solve critical section problem as only one process can be present in critical section but in this mechanism if a process halts in entry section just before setting the Value =1 a different process may enter resulting any number of processes in critical section.

2.Peterson’s Solution

1.Two process solution.

2. Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted

3.The two processes share two variables:

int ;

Boolean flag[2]

4.The variable <u>turn</u> indicates whose turn it is to enter the critical section

5.The flag 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!

Algorithm For Peterson's Solution--

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

It satisfies the requirements for solution to critical section problem as we can see..

1. Mutual exclusion is preserved.
2. Progress requirement is satisfied.
3. Bounded-waiting requirement is met.