# SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107
Accredited by NAAC-UGC with 'A' Grade
Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

## Department Of Artificial Intelligence and Data Science

## Course Code & Name –23ITB203 & Operating Systems

## II Year / IV Semester

## Unit 2 - PROCESS SYNCHRONIZATION - THE CRITICAL-SECTION PROBLEM -SYNCHRONIZATION HARDWARE

# Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

**Cooperating processes can either**

- Directly share a logical address space
- Allowed to share data only through files or messages

Concurrent access to shared data may result in data inconsistency!

The orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained

Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Counter variable = 0

Counter is incremented every time we add a new item to the buffer counter++

Counter is decremented every time we remove a new item from the buffer counter- -

Example:

- Suppose that the value of the variable counter is currently 5
- The producer and consumer processes execute the statements “counter++” and “counter- -” concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4,5,6
- The only correct result, though, is counter ==5, which is generated correctly if the producer and consumer execute separately.

“counter++” may be implemented in machine language (on a typical machine) as:

```
register1  = counter
register1  = register1 +1
counter    = register1
```

“counter—” may be implemented in machine language (on a typical machine) as

```
register2  = counter
register2  = register1 -1
counter    = register2
```

| | | | | |
|---|---|---|---|---|
| T0 | producer | execute | $register_1 = counter$ | $\{register_1 = 5\}$ |
| T1<br>T2 | producer<br>consumer | execute<br>execute | $register_1 = register_1 + 1$<br>$register_2 = counter$ | $\{register_1 = 6\}$<br>$\{register_2 = 5\}$ |
| T3<br>T4 | consumer<br>producer | execute<br>execute | $register_2 = register_2 - 1$<br>$counter = register_1$ | $\{register_2 = 4\}$<br>$\{counter = 6\}$ |
| T5 | Consumer | Execute | $counter = register_2$ | $\{counter = 4\}$ |

We would arrive at the this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

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

Clearly we want the resulting changes not to interfere with one another. Hence we need **process synchronization**

# The Critical – Section Problem

Consider a system consisting of n processes { P0 , P1, …, Pn}.

Each process has a segment of code, called a

**Critical Section**

In which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

**The critical –section problem is to design a protocol that the processes can use to cooperate.**

- Each process must request permission to enter its critical section
- The section of code implementing this request is the entry section
- The critical section may be followed by an exit section
- The remaining code is the remainder section.

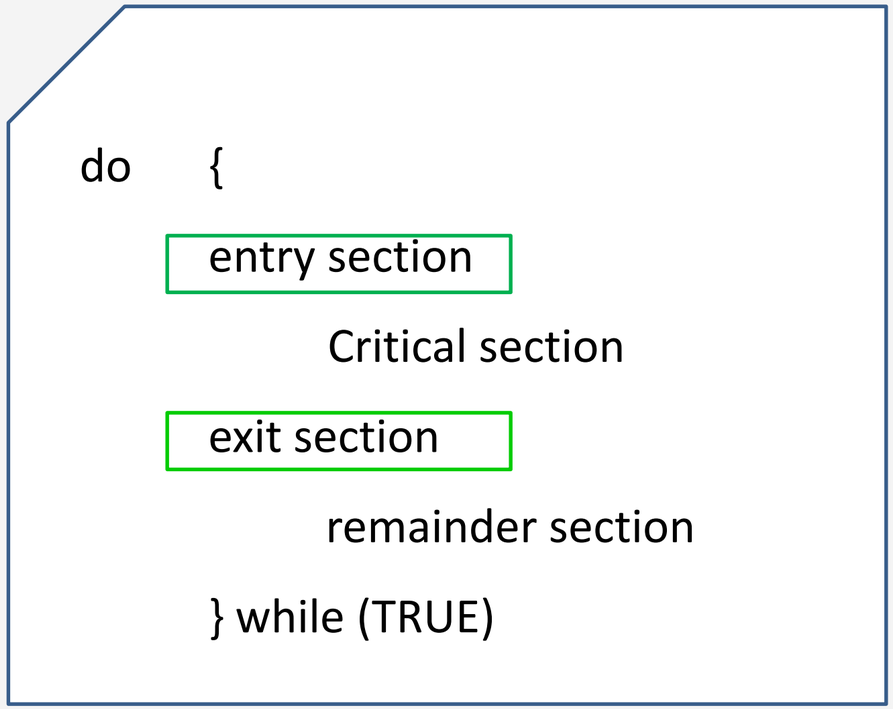


**Figure: General structure of a typical process.**

- Each process must request permission to enter its critical section
- The section of code implementing this request is the entry section
- The critical section may be followed by an exit section
- The remaining code is the remainder section.

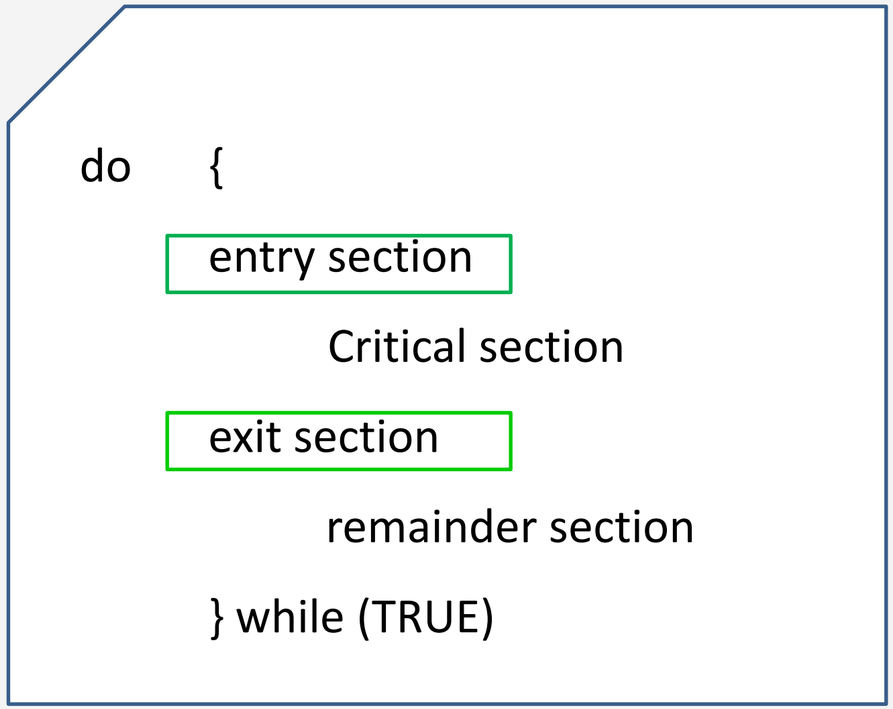


**Figure: General structure of a typical process.**

# Peterson's Solution

- A classic software – based solution to the critical – section problem
- May not work correctly on modern computer architectures
- However, if provides a good algorithmic description of solving the critical- section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Lets call the processes $P_i$ and $P_j$

Peterson's solution requires two data items to be shared between the two processes

int turn

Indicates whose turn it is to enter its critical section

boolean flag [2]

Used to indicate if a process is ready to enter its critical section

**Structure of a process Pi in Peterson's solution**

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

    Critical section

    flag[i[ = false;

    remainder section

} while (TRUE);
```

**Structure of a process Pj Peterson's solution**

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

    Critical section

    flag[j[ = false;

    remainder section

} while (TRUE);
```

## Test and Set Lock

- A hardware solution to the synchronization problem.
- There is 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
- If it is not locked, it takes the lock and executes the critical problem

```
boolean TestandSet (Boolean *target){
boolean rv = *target;
*target = TRUE;
Return rv;
}
```

ATOMIC OPERATIONS

**The definition of the TestandSet() instruction**

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

Process 1

Process 2

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

Satisfies mutual exclusion.

Doesn't satisfy bounded-waiting