UNIT-III

5.0 Introduction

- This chapter introduces the concepts of process synchronization.
- It introduces the critical section problem, S/W and H/W solutions to the critical section problem.
- This chapter describes the problems encountered in the inter process communication by taking an example of producer consumer algorithms.
- This chapter introduces the concept of semaphore, critical region.
- This chapter also focuses on classical problem of synchronization, like bounded buffer, reader-writer and Dinning philosopher.
- At the last we will discuss the concept of monitor and their uses.

5.1 Background

- Multiprogramming created an environment for concurrent classical processes. It also made it possible for programmer to create group of cooperating processes to work concurrently on the single problem.
- The resulting computation could even exhibit true parallelisms if some of the processor were performing I/O operation while other uses a CPU.
- In practice, several processes are wanted to communicate with each other's simultaneously. These require proper synchronization because it shared data residing in shared memory location.
- **As we know there are two main types of processes one is independent and another is co-operative.**
- The independent processes are those which cannot be affected by other processes and it can not affect to other processes, where as the co-operative processes are those processes that can be affected by other processes and can be affects to other processes.
- Generally cooperative process in which each process shares some shared location, variable or any resource at a time.
- So In this scenario, two or more concurrent processes are trying to modify the same common shared variable that time data in-consisting may occur.
- That **data inconsistent** may end up with wrong results.
- So in order to maintain the data consistency we need to provide the process sequence that leads to proper execution of all processes.
- Simply it needs to serialize all the activity of all cooperating process operations.

– Eg. Bank transaction, shared variable updation.

5.2 Critical section

Q. Define critical section.

Q. Write a shout note on CS?

Q. What do you understand by CS problem? What requirement should be meet by the CS solution.

Q. Explain the condition that a CS problem must satisfy.]

Q. What is CS problem?

- In transportation, intersection are part of a street, but they are a unique part of the street because the intersection portion is a shared between two different streets.
- In below cartoon, a bus proceeds from one side and car from other side.
- If the bus and the car get to the intersection at the same time.
- There will be a collision. We say that the intersection is a critical section of each street.

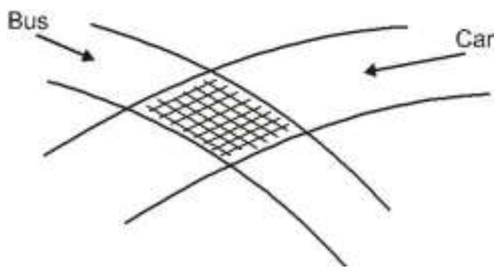


Fig. 5.1 Simple traffic intersection.

- Critical section occurs in concurrent s/w where two process/threads access a common shared variable. Like bus and car, there may be certain parts of the two processes that should not be executed concurrently.
- Such parts of the code are the S/W critical section, for example, suppose that two process P1 & P2 execute concurrently to access a common integer variable, balance.
- For example, thread P1 might handle credits to an account while P2 handles the debits. Both need access to the account balance variable at different times. This operation works as subtractions and addition.

code:- shared double balance
// shared variable

```
code for P1                     code for P2
-----                             -----
balance = balance + amount    balance = balance – amount.
-----                             -----
```

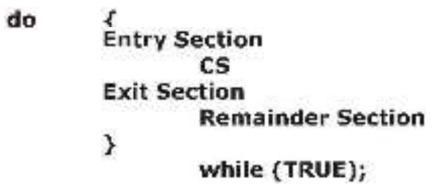
- The portion in any program which accesses by a shared resource is called critical section.
- In concurred programming a CS is a piece of code (data structure or device) that must not be concurrently accessed by more than one thread of execution at a time.
- A CS will usually terminate in fixed time and a thread, task or process will have to wait for a fixed time to enter in it.
- But in multiprogramming environment more process wants to execute their code in critical section, so those times which will get the chance of critical section to execute the process. (**Because of race condition)**
- Such operations are needs to be serialized, to avoid a race condition.
- 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 changes common variable, updates a table writes a file and so on.
- The important truth about system is that when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- Thus the execution of critical section by the process in mutually exclusive in time.

Note*:-The uniprocessor has single processor and it allows executing single process at a time so why there is a requirement to serialize the activity the answer is yes because of context switching among different burst (CPU & I/O) of processes.*

Critical Section problem

- In OS, there is **a segment of code**, which can be shared by many processes.
- But when two processes share the code and if one are updating the code while other is reading it, and then **conflicts may occur**. This is called **C.S. problem**.
- To avoid this problem a **lock i**s created and that block of code which establishes that new process can access that block until the first process release it, after its usage.

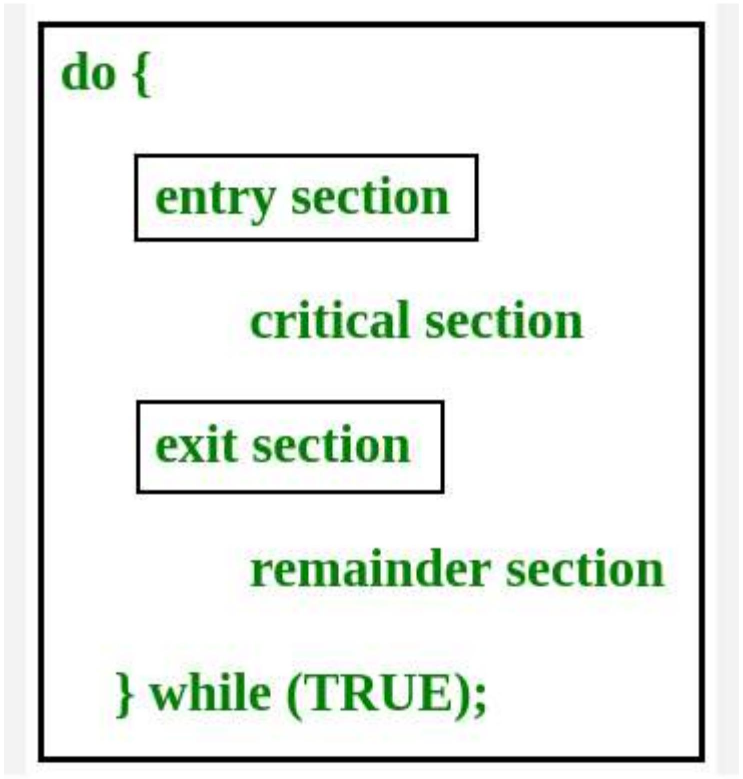
  Standard code structure of critical section problem.



```
do      {
        Entry Section
                CS
        Exit Section
                Remainder Section
        }
                while (TRUE);
```

Description

– consider a system consisting of processes {P_0, P_1 ,, P_n}. Each process has a segment of code, called critical section (CS), in which the process may be changes common variable, updates a table, writes a file etc.



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

Fig.: Structure of CS

- The important feature of the system is that when one process is executing is its CS, no other process is to be allowed to execute in its C.S. That is, two processes cannot execute in their CS at the same time.

Sections: ----critical section structure comprise of four section

1. **Entry section**:- Each process must request permission to enter in its CS. The section of code implements this request is the “entry section”. Many processes can request at a time but only one successes
2. **Critical section**:- Only one process can execute at a time.
3. **Exit section**:- once CS done process come out of that called exit section, means Execution of process has been completed
4. **The remaining section**: - if any portion of process remains to get executed in first iteration.

Conditions of good solution or requirement to CS problem.

A solution to the CS problems must satisfy the following requirement.

1. **Mutual exclusion**:

- If the process Pi is executing in its CS, then no other processes can be executing in their CS.
- It means that only one process can enter in its critical section at a time.

Implications

- Critical section better be focused and short.
- Better not get an infinite loop in there CS.

If a process is somehow it halts/waits in its CS. it must not interleaved with other processor.

2. **Progress**: - if no process is executing in its critical section and there exist some processes that wish to enter into their CS, then the section of the process that will enter the CS next cannot **be postponed indefinitely**. And remainder section process cannot take part into race condition

Implications

- If only one process wants to enter, it should be able to.
- If two or more are want to enter, one of them should succeed.

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

Implications

- Assume that each process executes at a non zero speed.
- No assumption concerning relative speed of the n process.

5.3 Various solutions to critical section problem?

Q. What are the various solutions to CS problem?

– The solution for CS problems are generally categorized into some categories

(A) Software based solution

(B) Hardware based solution

(C) OS Based solution

(D) Programming Language based solution.

– As per the syllabus this categorized into three categories that are as follows:

(A) S/W based solution

“Algorithm who’s correctness does not rely on any other assumption”.

i) Algorithm 1

ii) Algorithm 2

iii) Algorithm 3 (Peterson)

iv) Bakery Algorithm

(B) Hardware based solution

“It relies on some special machine instruction”.

i) Lock

ii) Test and set

iii) Swap.

(C) OS Based solution

"Provide some function and data structure to the program through System Library calls"

i) Semaphore

ii) Monitor

(A) Software Based solution

– Software based solution are applicable to solve two process synchronization problem at a time.

i) **Algorithm** 1

– This solution for CS is based on turn value. Once turn is set to pi processes, it will get access to CS.

Code:
```
shared variable
            int id turn;
            initialized value = 0
    do {
            while (turn ! = i)
            CS
            turn = j;
                    Remainder section
            }  while (1);
```

- In above solution there are two processes Pi and Pj.
- When the value of turn = = 1 then process Pi will eligible to execute in their CS, where as Pj have to wait till the execution of Pi.
- In other hand if turn of other process then it have to wait. (if pi turn pj must will wait)
- Now, we will check whether. Algorithm 1 is a good solution to two process communication or not.

1. **Mutual exclusion:**
    - ME is satisfied because at a time those process (either Pi or Pj) having turn = true that
    - Process can execute in their CS, that time other process have to wait.
    - This solution provides alternative chance to each process.

2. **Progress**

- When one process gets hold of CS execution, then it can keep CS till the entire execution.
- Because there is no any provision to switch from one process Pi execution to Pj in finite time. Hence this (progress) condition is not satisfied (there is no fixed time to switch).
- Consider the example of game:-*There are two brothers who wanted to play a game*

on computer but there is only one computer is available, and mother of these two child has provided a solution to play game in alternative manner but she not declared or said how many time both should play the game, so if elder brother got a chance to play then he decided to play the game infinite time, which means he is not allowing his younger brother to play game in finite time hence small child will not get a chance to play a game in within a time.

Note: This solution is not a good solution because it cannot holds all the conditions of good solution.

ii) Algorithms 2

As we explained in Algorithm 1 that, it not provides a good solution to two process CS problem

– Code shared variables

```
    Boolean flag [2] ;// set to false
    Flag [i] = true : // pi ready to enter in its
CS do {
        flag [i] = true
        while (flag [j])
                :
        CS
        flag [i] = false
        RS
    }  while (1)
```

- This Algorithm is based on flag value.
- Boolean flag of 2 array elements are initialized to two processes.
- As per the code, those processes are having flag value is true then it will get CS access.
- Alter the execution process will set flag to false, it means it allows executing to other process.
- Still this solution is not a good solution (will check all condition).

1. ME
   - ME is satisfied because those process having flag value is true it will execute in their CS.
   - And if another process wants to enter into CS so it will spin in while loop till the execution of first process.
2. Progress
   - Consider if both processes came at a time and set their flag true, so only one can enter into their CS
   - And other has to spin in while loop, this spin may take too (infinite) much time to get chance.
   - Due to this problem this solution is not a good solution.
3. **Algorithms 3 (Peterson solution)**

- As we have seen in above both solution's that they both are not good solution to solve the CS problem.
- Peterson algorithm is combines the features of above both algorithms.
- Peterson algorithm is the simple and elegant solution for CS problem.
- When n process CS problem arise that time Peterson solution can be used.

Code

```
do {
            flag [i]: TRUE
             turn = j;
                  while (flag [j] && turn = =j);
                  CS
             flag [i] = FALSE
                              ; RS
            }
            while (TRUE) ;
```

- In above algorithm every processes sets its flag to true and turn to other process
- When the processes having flagged is true and turn also set to process that process will enter into CS.
- After executing in CS the process will set flag to false.
- Here, Mutual exclusion is possible, at a time one process can execute into their CS.
- Progress condition is also achieved in this code/algorithm because it will take decision in finite time. (Due to flag and turn value is to other process).
- Bounded waiting condition is also satisfied because it allow fixed no's of time to every processes for execution.
- Hence, Peterson solution is good solution because it provides good solution to process synchronization.

Drawback:-

- But it has some drawback likes position of flag and turn may conflicts.
- As we can see the code makes the process more tricky and risky.

4. Bakery algorithm

- Bakery algorithm is developed by **LESLIE LAMPORT.**
- This algorithm is specially used in **distributed environment** (OS) where arrival of task happens in random order & no one is master workstation.
- This also provide solution to n process CS problem
- When there are n process want to Enter in CS that time each process required a ticket no's.
- Those process having **small token/ticket value** that will get a chance to enter into CS.
- If same ticket is generate that **time conflict occurs** but this decision will taken care with the consideration of small process pid (system PID) no's.
- Those the having small process value it will enter into its CS.

Note: "Bakery algorithm is not a part of our simple OS Syllabus".

"This algorithm is specially design for Advance OS concept, like distributed

OS".

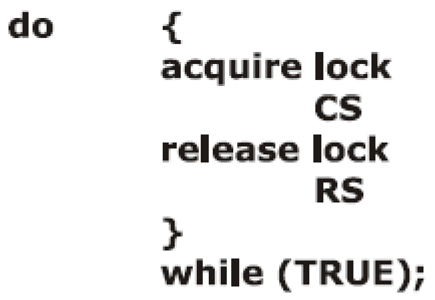
(B) Hardware Based Solution:

- Nowadays computer system architecture functions and complications are increase, and software based solutions are not suitable for today's hardware.
- Software based solutions are possible to solve problem in uniprocessor environment by using interrupt, but it's not good in multiprocessing because it may cause processing delay.
- Generally accesses to a memory location exclude other access to that same location.
- The designer has proposed machine instruction that performs different operations automatically on the memory location at same time (R/W/M) i.e. reading, writing, modify.
- This machine instructions are enough to provide mutual exclusion but some extract mechanism are needed to satisfy the other two requirements (progress & Bounded buffer) of good solution to CS.
- Hardware instruction are device dependent, to use lock and release the devices.

i) Lock

- In order to achieve process synchronization we need to provide individual access to each process.
- For that we need to serialize the activity, hence we need an explicit lock mechanism which will guarantee to avoid race condition.
- If any process wants to execute in their CS it required to hold lock.
- After executing in CS they should release that for other process.
- Figure shows the simple lock and Release structure to CS problem.
- Those process having lock that can be able to execute in their CS.
- After work/execution it needs to release the lock

Code:



```
do      {
        acquire lock
                CS
        release lock
                RS
        }
        while (TRUE);
```

Fig. lock structure

ii) Test And Set

- Test And set is an instruction which support to solve CS problem in multiprocessing environment.
- Test And set instruction can be defined as follows

```
boolean Test And set (boolean * Target)
{
boolean rv = * target
* target = TRUE;
return rv
}
```

- As the researchers said this instruction should support all (R/W/M) the operation, and Test and set instruction supports that.
- Because this instruction is executed automatically, and also support read, write and modify operation.
- Consider the Boolean rv = * target this statement support read operation and targeted value is stored in rv variable.
- Second * target = true statement which modify the flag value.
- Finally by return statement it also supports to write operation.
- N-process CS problem solution using Test and Set instruction.

```
do {
    while (Test And Set (& lock) )
        //  do nothing
        //  CS
    Lock = FALSE
        //  Remainder
    section } while (TRUE)
```

- Consider above code segment, In order to execute in CS every process sets the “Test and Set” and “lock” should set to true then only process can enter into their CS.
- Those process succeed to set above both condition, they will get execution in their CS, and keep lock value to true till the program execution.
- If another process want to enter in CS but it cannot because it will spin in while loop till the lock set to false.
- Even after this solution is not a good solution because it only holds ME and progress condition but not support bounded waiting.

iii) Swap

- Swap instruction uses the key as a local variable and set it to false initially.
- Whereas lock is a global variable and it also set to false.
- The swap function can able to interchange the values.
- Consider if only one process wants to enter in CS then it should set the value of key = false and lock = true then it will check whether key is true, if the answer is no then it can enter in CS.
- Second case if more than one process (two) so, one of the process can get chance to enter in CS using swap function.

- Although, this solution is not a good solution because it does not satisfy all three condition. It holds ME and progress only but not hold bounded waiting.

  Code

```
do {  Key = TRUE
        While (Key = = TRUE)
        Swap (&lock, &key)
                //  CS
        Lock = False
                //  RS
    } while (TRUE);
```

Good solution using Test and set method of

```
do
{
    waiting [i] = TRUE
     KEY = TRUE
    While (waiting [i] & & Key)
    Key = Test And set (& lock);
waiting [1] : false
    cs
j = (i + 1) % n
while ((j = i) & & ! waiting [j]
 j = (j + 1) %n
        if (j = i) Lock = False;
        else
waiting [j] = false;
// RS
} while (TRUE)
```

Description

- It uses three variables one is key it sets to local and lock and waiting is set to global
- If we go through all code it's possible to satisfy all three conditions

(C)OS Based Solutions

Q. How do semaphore provide better solution as compared to the other s/w solution for the CS problem? Explain in default.

Q. Explain semaphore and what the limitations of semaphore are.

Q. What is the purpose of semaphore in CS problem?

Q. What is P & V operation in process synchronization?

Q. What are the limitations of semaphore?

Q. What is semaphore? What is the difference between binary and counting semaphore? Explain how improper implementation of a semaphore can lead to a deadlock.

1. Semaphore

- In software, a semaphore is an OS abstract data type that performs operations similar to the traffic light.
- The semaphore allows one process to control the shared resource while the other processes have to waits for the resource to be released.

Main Points

- Semaphore is a synchronization tool, which is used to solve the n-process CS problem.
- Semaphore is OS based solution and it is present in OS itself.
- Semaphore is an integer variable that can only be accessed via two individual (atomic) operation i.e. wait () and signal ().
- Actually semaphore "S" is an Abstract data type and it use to control access by multiple processes to common shared resource.
- Consider the example of a road, and vehicles, here road is a resource it called as a semaphore which is shared by both the vehicles, and operation is passing the vehicles on the road.

Semaphore operations

There are two atomic operation of semaphore

1. wait ( )
2. signal ( )

1. Wait ( )

```
wait (s)
{
    while s < = 0
    // No-operation
S – –
}
```

2. Signal ( )

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

1. *In wait function the value of semaphore will be decremented and once it reaches to zero the process can enter into their critical section.(Wait is use to take a hold)*

2. *In signal function the value of semaphore will be increased by 1 means it allows to other (signal is use to release the hold to others)*

Description: - Let the initial value of a semaphore be 1, and say there are two concurrent processes P0 and P1 which are not supposed to perform operations of their critical section simultaneously.

Now consider P0 is in its critical section, so the Semaphore S must have value 0, now consider P1 wants to enter its critical section that time it executes wait (), and in wait () it continuously gets spin, now to exit from the loop the semaphore value must be incremented, but it may not be possible because according to the wait () it is an atomic operation and it can't be interrupted and thus the process P0 can't call signal () in a single processor system.

Note: The wait ( ) is also called as P operation and signal ( ) is also called V operation.

# usage or Implementation of Semaphore

– Semaphore can be used to (n) process CS problem.

– Consider the code.

```
# shared data
Semaphore mutex = 1;

# Structure of Pi
do {
wait (mutex)
CS
signal (mutex);
RS
} white (1);
```

Description

- Consider the above code structure in which the shared data (semaphore) is mutex & it's initialized as a 1.
- The process Pi wants to enter into their CS. & the structure for process Pi consists of both wait and signal operation.
- As per the definition of wait ( ) operation the initialized value of mutex (i.e. 1) is decremented to 0 by wait() s -- statement then process can enter into CS.
- Once the process completes their execution in CS, then by using signal ( ) operation it will increment the value of mutex and it release the CS to another process.

Now check whether this solution is good solution or not.

1. **ME**: This solution satisfies the ME condition because only one process (Pi) can enter in their CS at a time.
2. **Progres**s: It also holds progress condition, as per the condition of progress, the process in remainder section cannot participate and we cannot postpone it to infinite time. It takes single execution of wait instruction. Means it does provide a chance to other process in finite (specific) time.
3. **Bounded waiting:**

- This property cannot hold. As per the bounded waiting condition we should assign a fixed chance to all process.
- Consider if process Pi executes in CS and quickly bumps the value of semaphore and again it can go to entry section.

- By this scenario we cannot give exact limit that how many times each process can allow to enter into their CS.
- Because any process can bumps the value of semaphore and will enter into the entry section.
- *Note: Above solution does not provide a good solution because it not satisfied all three conditions. But we have said that semaphore is used to solve the n-process CS problem for that we need to change the semaphore definition by adding some new code into semaphore definition to achieve good solution for CS problem.*

Actually wait ( ) and signal ( ) operations are not a machine instruction; they are piece of code this helps the mutex to control the activity.

- **Busy waiting**: When a process Pi in its critical section and other process wants to enter into their CS that time it check the loop frequently (CPU cycle are wastes) but it did not get succeed to enter into CS.
- As shown in the examples above, processes waiting on a semaphore the other process need to check frequently to see if the semaphore is zero or not.
- This continual looping is clearly a problem in a real multiprogramming system (where often a single CPU is shared among multiple processes).
- This (spinning in loop) is called busy waiting and it wastes CPU cycles. When a semaphore does this, it is called a spinlock.
- Busy waiting is wastage of CPU cycle that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" in while loop and waiting for the lock.
- **Spinlock** is useful in multiprocessor system. The advantages of spinlocks are that no context switch is required. A context switch may take considerable amount of time.
- To overcome the above problem of busy waiting. It's better to modify the definition of wait and signal semaphore operation.
- The solution, is instead of busy waiting, the process can block itself, and should wait in queue which is associated with the semaphore.
- The implementation of semaphore definition to above solution we have one structure i.e.

```
type def struct
{
    int value ;
    struct process * L;
} semaphore;
```

- Each semaphore has an integer value and a list of processes.
- When process waits on a semaphore it is simply added to the list of processes (At the time of attempt to acquire CS).
- Then signal operations removes one process from the list of waiting processes and awake that process.
- For this purpose two methods are used, one's block ( ) and another is wakeup ( )
- Where block ( ) function will suspends the process and wakeup ( ) function will resume the execution of a blocked process.

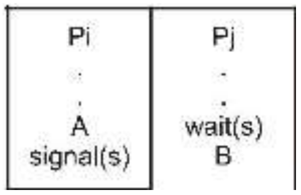
- Busy waiting leads to wastage of CPU cycle, so instead of busy wait processes should wait only in queue.
- This mechanism can save the CPU cycle while attempting the CS because we have created a queue in which processes can reside till the response.
- Note:-Busy waiting version is better when CS is small and queue waiting version is better for long CS.

# Use of semaphore to process synchronization

1. Show how semaphore solve two process CS problem

   Problem statement: - Execute a statement B in Pj only after the A has been executed in process Pi.

   Solution:



Pi	Pj
.	.
.	.
A	wait(s)
signal(s)	B

Where Pi = process pi

Pj = process Pj

A = statement of process Pi

B = Statement of process Pj

-All Are Initialized to zero.

- Now, as per the problem statement, statement A in Pi should execute before the B statement in Pj
- Here, As a solution wait and signal operation is used to solve this problem, in which the statement A is executed 1st then by signal operation the CS release to other process. Where Process Pj is waiting to get hold of semaphore but that is hold by Pi, now Pj will get hold of S and the statement B can execute after the A.

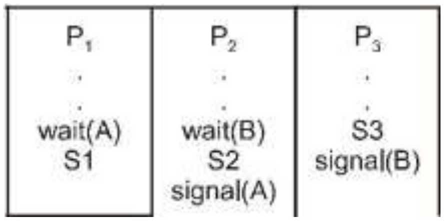
2. **Show how semaphore solve 3 process CS problem**

   Problem statement: The statement SI in process P_1 executes only after the statement S2 in P2 has executed and statement S2 in P2 should execute only after the statement S3 in P3 has executed.

   Solution: where P_1, P_2, P_3 are process

   S_1, S_2, S_3 are statement and

   A and B are the semaphore



P_1	P_2	P_3
.	.	.
.	.	.
wait(A)	wait(B)	S3
S1	S2	signal(B)
	signal(A)	

- In above solution is a proper sequence of atomic operation to solve 3 process CS problem
- All are waiting for respective semaphore the S3 is executed by P3 and P2 is waiting for the B semaphore which is hold by P_3 that will release and get by P_2, then P_2 can

execute statement S_2.

Finally P_1 can execute statement S_1.

Note:-Programmer can suggest/use any/many sequence of wait and signal operation to solve CS problem.

# Bad used of semaphore or limitation of semaphore

- There is no doubt that semaphore is not provides a good solution.
- It is semaphores quality that it gives solution to n process CS problem.
- As we all know about semaphore operations as i.e. wait () and signal ().
- But bad use (sequence) of semaphore operation will leads to some serious problems

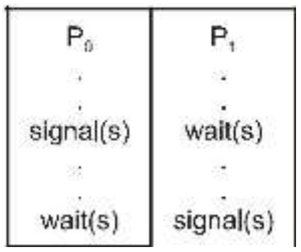
These problems are:-

(a) violation of ME

(b) Indefinite blocking/Starvation

(c) Dead lock

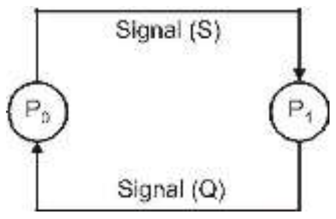
1. Violation of mutual exclusion



Consider the side scenario situation of P_0, and P_1 processes which used both operations.

It has been given that both P_0 and P_1 can enter into CS at a time. if both enter into CS so there is obvious violation of ME.

2. Deadlock



– Improper use of semaphore will lead to deadlock problem.

– Consider the general scenario where the one process is waiting to take hold of some resource and required by other process but no one will get hold of resource/semaphore. (As per standard Definition of Deadlock).

– In this scenario Po process waiting for signal operation on Q semaphore and P_1 is waiting for signal operation on S semaphore.

– Now see how semaphore leads to dead lock

Where'; – P_0 and P_1 is process

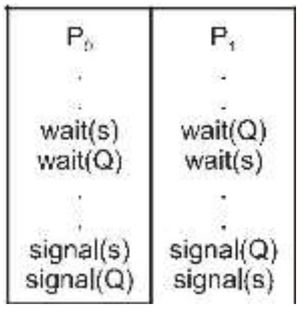
– S and Q are the semaphore

– Both are initialized to 1.

– Both P_0 and P_1 process wanted to used Q and S.

– P_0 is using S semaphore and waiting for Q which but it is already hold by the process P_1 and P_1 is using a Q semaphore and waiting for S which is hold by the P_0.

– It means both are depending on each other, and no one going to release the semaphore hence deadlock is possible.



P_0	P_1
.	.
.	.
wait(s)	wait(Q)
wait(Q)	wait(s)
.	.
.	.
signal(s)	signal(Q)
signal(Q)	signal(s)

3. **Indefinite blocking/starvation:**

- Starvation is a situation where every process wanted to enter into CS but no one will get hold of CS because that already holds by other and will not releasing to others.
- Consider the scenario where process P0 and P1 both are wanted to enter into CS.
- Let's consider, if 1st process P0 (or p1) will able to enter into CS due to wait ( ) but to release that CS P0 has to used signal ( ) operation but by mistake programmer has used wait ( ) operation.
- In second case P1 is waiting to enter in CS but semaphore S yet not released by process P0. So, P1 cannot enter into CS and it leads to indefinitely blocking (Starvation) or vice versa.

Note: - Programmer must use atomic operation in proper sequence to avoid misbehavior

# **What is Difference between Binary and counting semaphore**

Binary Semaphore	**Counting Semaphore**
1. It has two states. i.e. Taken and not taken	1. It is having N internal state. i.e. +ve, –ve and zero.
2. This semaphore has ownership.	2. This semaphore don't have ownership attribute and can be signaled by any thread or interrupt handler regardless of who Performed the last wait operation.
3. This semaphore having no * 1 priority Inheritance.	3. This also have no * 1 priority inheritance.
4. Queue organization is FIFO or priority.	4. Same as binary semaphore.
5. Simply Integer value can range in 0 & 1 Only.	5. The integer value can range over an Unrestricted integer domain.
6. It having two method associated with it	6. NA

(Up/down, lock/unlock).

Note: Both semaphores are used to solve n-process synchronization problem.

5.4 Classical Problem of Process synchronization

Q. 1. Explain semaphore solution and Reader-writer problem.

Q. 2. Explain Dinning philosopher problem.

Q. 3. Explain at least two classic problem of synchronization.

Q. 4. What is semaphore? State and solve the reader writer problem with the help of Semaphore.

- As we all know that semaphore is a way to solve process synchronization problem.
- In order to check whether semaphore gives a good solution or not we need to apply semaphore solution on standard or real time problems then we can say semaphore is having a capacity to solve synchronization problems.

There are three classical problems i.e.

(A) Bounded Buffer

(B) Reader–Writer

(C) Dinning philosopher

(A) Bounded Buffer

- Consider producer and consumer problem, where each having different capacity to produce and consume an item. (Explained on last page)
- So the rate of producing data may be high or low and the rate of consumer may be high or low (we can not predict).
- If this happened then we need to control the activity of each process means at which rate producer is producing, the same way consumer should consume an item and vice versa.
- If it is not possible to control the rate of each process so, we need provide intermediate facility (like buffer) to control these activities.
- I.e. we need to provide buffer to put items produced by producer so that consumer will consume as per their rate.
- Bounded buffer is a mechanism in which these problems can tackle properly & semaphore can used to solve this problem.

-**We will see how it helps to this problem.**

– **Solution**

# shared data

Three semaphore

1. Full :- It indicate how many slots are free in buffer
2. Empty: - It indicates number of empty slot.
3. Mutex: - It used for ME either by producer by consumer.

# Initialization

Full = 0; empty = n; mutex = 1

# *Producer code*

```
do{
producer produce an item next
        - - -
wait (empty)
wait (mutex)
- -
add next to buffer
signal (mutex)
signal (full)
} white (1):
```

Description:

- Producer can produce an item as per the empty slot available (maximum) in empty semaphore.
- Wait on empty semaphore is used to decrement the value of empty slot. By n - - .
- After decrement we need a hold on buffer that is done by the wait on mutex, then producer will add produced item in buffer.
- Finally adding value, we need to release that semaphore to other process by signal operation.

# ***Consumer code***

```
do {
wait (full)
wait (mutex)
- -
Remove item
signal (mutex)
signal (empty)
- -
consume the next item
}
white (1) ;
```

- After producing an item by a producer, consumer will able to consume an item.
- In order to consume at item we is need to first check the condition of "full" Semaphore i.e. it is having an items or not, means check if that is > 0 or not, if yes

then its full.

- Wait on Mutex operation will hold the buffer, then consumer can consume an items.
- Signal on mutex it used to release mutex for other producer.
- And finally signal on empty show one empty value in empty buffer.

(B) Reader writer problem

- In multiprogramming and multitasking environment single resource is shared by more than one users or processes at a time.
- Which need to provide exclusive access to each resource?
- Consider an example of any variable, file or database most of the processes and user are wants to share, so inconsistency may occurs.
- Semaphore can be used to restrict access to the database under certain condition.
- Semaphore are used to prevent only writing process from changing information in the database white other processes can read from the database.
- Consider the general (fig) snap shot of any resource where more than one reader can eligible to read the content but only one can allowed to write at a time. (example reading news paper is possible but writing the same is not possible)
- The general solution to this problem can be assigning the priority to each one so that serialization activity can be achieve and conflict can avoided to access the shared resource.

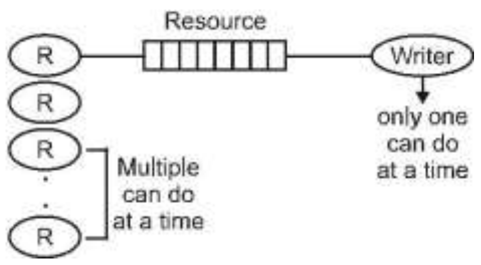


Fig. General Scenario

– There are two algorithms to this classical problem.

1. First Reader and Writer problem

- No reader will be kept waiting unless to writer has already obtained permission to use the shared object.
- If any reader is reading the value and other reader and writer are wanted to do, so allow only reader to read the data.
- Simply in this case reader will have to wait minimum and writer need to wait max (more) time.

2. Second Reader writer problem

- In second reader writer algorithm, if any reader is reading and some reader and writer want to do, so allow writer in very less time and force reader to wait maximum time.

- i.e. writer will wait minimum time and reader have to wait maximum time.

# Solution

# shared data:

semaphore (1) Mutex = This can used for ME

(2) wrt = This is for, it 1st Reader want to enter into CS then block other. means it Ist Reader access wrt so it should hold and block others.

# initialization

mutex = 1 ; wrt = 1

read count = 0; It indicate nos of reader are reading (1, 2, ...n)

code: -

```
wait (wrt)
-------
writing is performed
-------
signal (wrt)
```

Description:

- In order to do some operation on shared variable we need to take hold of wrt semaphore (need to lock wrt).
- Lock on wrt will lead to exclusive access
- Then writing can be possible.
- Now if another process wants to read so that it needs to spin in loop.
- After completion signal on wrt must perform to release hold of wrt.

– Code

```
{ wait (mutex)
read count ++;
if (read count = = 1)
wait (wrt);
signal (mutex);
– –
Reading is performed
– –
wait (mutex)
read count – – ;
if (read count = = 0)
signal (wrt)
signal (mutex) }
```

Description

- When more reader wants to read a shared variable/resource then we need to increment the read court.
- Now if anyone is 1st reader, then have to block other writer by wait on wrt function.
- Then release on mutex will release semaphore.
- If the reader is a last reader so then it need to decrement the value of read count by --.
- At last that needs to apply signal operation on wrt so that reader and writer can hold wrt semaphore.

(c) Dinning philosopher

- This problem is third classical problem which is very interesting.
- Five Chinese great philosophy waste their total life on thinking and eating.
- The great philosopher thinks a lot of time and when they became hungry then they want a lunch in a finite time.
- Another fact is the Chinese people eat lunch by spoon or chopsticks only otherwise they never do their lunch.

# General flow of dinning problem

$$\begin{tikzcd}
\text{Think for a while} \arrow[r] & \text{pick up chopsticks} \arrow[d] \\
\text{Put chopsticks} \arrow[u] & \text{Eat for a white} \arrow[l]
\end{tikzcd}$$

# Problem statement

– Consider a table, plates, philosophers and chopsticks,

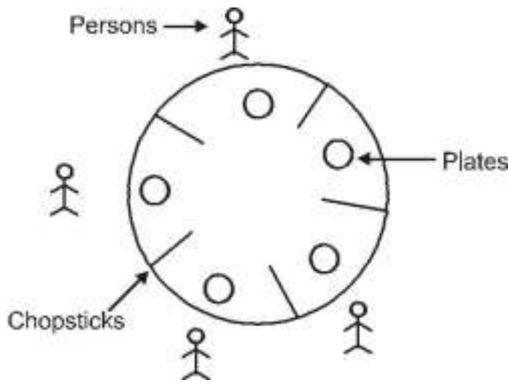


Fig.: General Scenario

- Initially every philosopher think and when they become hungry then they eat something.
- Early day individual philosopher gets hungry and eats food and get back to work.
- But one day (sad day) all 5 philosopher became hungry at a time and everyone wanted to eat a food.
- Hence they sit on table

– But there are some constraints to eat a food i.e.

1. Every philosopher must have two chopsticks
2. Every philosopher must pick up right chopsticks 1st and then left.

– Now problem is that who will get both chopsticks, because if each philosopher picks their right hand chopsticks and wanted to pick left one but left side chopsticks is already picked by the next (left side) philosopher.

– Then deadlock like situation arises and no one will get both chopsticks.

-They wait and wait for long time but no one got chopsticks, this situation constantly remains hence all remain hungry.

– This is called as a dinning philosopher problem.

# **General solution to such a problem**

1. Allow at most four philosophers to sit simultaneously at table.
2. Allow a philosopher to pick up chopsticks only if both chopsticks are available.
3. Apply even and odd option to pick up the chopsticks.
4. Or Allow those philosopher who is most hungry.

Code for philosopher

```
do {
wait (chopsticks [i])
wait (chopsticks [((i + 1) % 5)]
- -
// Eat food
signal (chopsticks [i]);
signal (chopsticks [(i + 1) % 5]; // Think
} while (TRUE)
```

Description:

- One simple solution is that represent each chopsticks with a semaphore.
- A philosopher tries to grab a chopsticks they needs to execute a wait () operation on
- They can release their chopsticks by executing the signal () operation on the appropriate semaphore.
- If we go through the code this solution is also not a good solution because if leads to starvation and deadlock.
- So we need to apply new strategy to solve dinning philosopher problem (i.e. monitors) will discuss in next topic.

5.5 Monitor

Q.1. what is monitor? State its advantages and disadvantages in concurrency control.

Q. 2. What do you mean by monitor? How does it give better solution than semaphore?

Q. 3. Write a short note on monitor.

Q. 4. Give the solution to Dinning philosopher problem using monitor.

-In previous section we have learn one process synchronization tool i.e. semaphore which can provide a solution to process synchronization problem.

– But bad used of semaphore will leads to some serious problems i.e. deadlock, mutual violation, indefinite blocking etc.

- As we have seen in Dinning philosopher problem semaphore is not provides a good solution.
- Monitor is a high level language construct that allow the safe sharing of an abstract data type among co-operating process.
- Monitor uses a concept of class mechanism of object oriented programming, and that class member function invoked many times.
- Only one process can achieve/reside in monitor at a time.
- The representation of a monitor type consist of declaration and variable whose value define the state of an instance of the type, as well as the bodies of procedure or function that implement operation on the type.
- The representation of monitor type cannot be used directly by the various processes.
- This procedure defined within a monitor & can access only those variable declared locally within the monitor and its formal parameter.
- The local variable of monitor can be access by only local procedure.
- The monitor construct ensure that only one process at a time can be active within the monitor.
- Actually the programmer does not need to synchronization the code that will ensure by language itself.

General syntax of monitor

```
Monitor monitor_name
{Shared variable declaration
Procedure P1 (- - - )
{ }
Procedure P2 ( - - )
{ }
Initialization
code { }
}
```

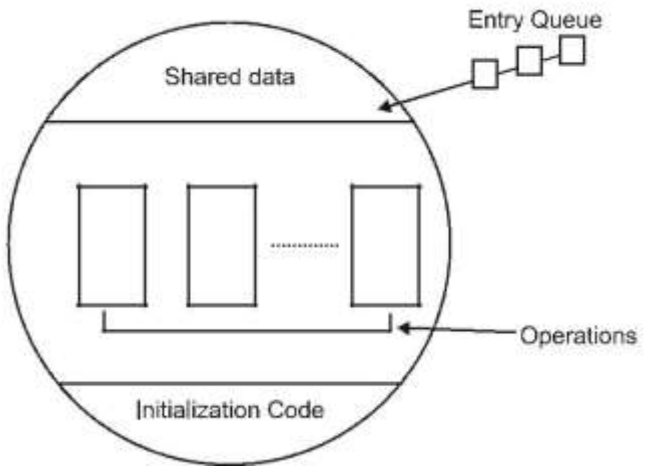


Fig.: General Structure of Monitor

# General points

- Monitor is a thread safe class, object or module that uses wrapped mutual exclusion in order to transparently be used by more than one thread.
- In concurrency programming a monitor is a synchronization construct that allow thread to have both ME and ability to wait (block) for certain condition to become true.
- From above all points and discussion monitor is used to concurrency control.

# Function code

- As per monitor syntax, we need monitor with condition variable.
- Additional synchronization construct are needed to model such synchronization, problem, they can be modeled with conditional variable.

I.e. Condition x, y

- Only two operations can be performed on condition variable i.e. wait and signal.
- Meaning and wait ( ) process is, invoking this operation is suspended unit another process invokes X. signal.
- Meaning of signal ( ) X. signal: Resume exactly one suspended process if no process is suspended this is a null operation i.e. the state of x is unaffected.

Note: The wait and signal function/operation of monitor are differed than semaphore.

# How to solve Dinning philosopher problem

Consider,

Three states: thinking, eating, hungry

Three functions: pickup, put down, test

Five conditional variable: self [5]

Initialization code, data, variable.

Structure

```
    monitor dp
{
enum {thinking, hungry, eating}
 state [5]; condition self [5];
    valid pickup (int i)
    valid put down (int i)
     valid test (int i)
    valid test (int i)
    {
        for (int i = 0 ; i < 5 : i + + )
        state [i] : thinking
} }
```

# pickup function **//philosopher id (int i)**

```
void pickup (int i)
{
state [i]=hungry ;
test [i]
    if (state [i] ! =eating)
self [i]. wait ( )
}
```

– In pickup function () a philosopher [i] can set state as a hungry by calling test function with respective id.

# **test function**

```
void test (int i)
{
    if state [(i + 4) % 5] ! = eating & & (state [i] = hungry)
     & &(state [(i + 1)%5 ! = eating])
{
    state (i) = eating
    self [1] . signal ( )
}
}
```

– Consider n philosophers and out of n philosophers assume 3rd philosopher became hungry then that philosopher need to test, is my right hand side philosopher is hungry or not, if yes then I have to block/wait, or if my right side neighbor is not in eating state, and I am hungry so, set my state to eating and call signal to release from previous state.

# putdown function

```
void putdown (int i)
{
state [i] = thinking : // set state to
 think // Test left and right neighbor
test ((i+4) %5) ;
test ((i+1) %5) ;
}
```

- In this function () state need to set as thinking.
- Then check whether right and left neighbor are hungry or not.
- If hungry (right) and their right neighbor are not eating then set as a eating.
- If not hungry and left neighbor is not eating then set state to eating?

# Disadvantages of Monitor

1. Since only one process can be active within a monitor at a time, the absence of concurrency is the major weakness in monitor and this leads to weakling of encapsulation.

– For example in the reader writer problem the protected resource (file or database) is separated from the monitor and hence, there is possibility that some malicious reader or writer could simply access the database directly without getting a permission from the monitor to do so.

2. Another problem is the possibility of deadlock in the case of the nested monitor call. E.g. consider a process calling a monitor that in turn calls another lower level monitor procedure. If a wait is executed in the last monitor called. The ME will be relinquished by the process. However ME will not be relinquished by process in monitors from which nested call have been made. Process that attempt to invoke procedure in these monitor will became blocked. If those blocked processes are the only cause signaling to occur in the lower level monitor, then deadlock can occurs.

Q.: What is the difference between semaphore and monitor?

	Semaphore	**Monitor**
1.	Semaphore is synchronization tool that does not required busy waiting.	Monitor is a high level abstraction that provides a suitable and affective mechanism to process synchronization.
2.	Semaphore is useful to solve mutual exclusion problem.	Monitor is best support for solving dinning philosopher process
3.	It does not acts as resource allocates.	It acts as single resource allocator.
4.	Can be used anywhere in a program that should not be used a monitor.	Can only be used in monitor.
5.	Wait ( ) does not always block the caller (i.e. when the semaphore counter is greater than zero).	Wait ( ) always block the caller.
6.	Signal ( ) either release a blocked thread, if there is one, or increase the semaphore counter.	Signal ( ) either release a blocked thread, if there is one, or the signal is lost as if it never happens.
7.	It signal ( ) release a blocked thread the caller and the release thread both continue.	If signal ( ) release a blocked thread the caller yield the monitor or continue only one of the caller or the release thread can continue but not both.

Que.: What is the difference between critical section and critical Region.

	CS (Critical Section)	CR (Critical Region)
1.	A process has a segment in which process may change variable, update variable and so, on that segment is called as critical section.	No process can share and update variable in critical region area.
2.	The problem occur when other process also comes into CS.	CR is used to implement the mutual exclusion and arbitrary synchronization problem.
3.	It needs various algorithms to solve the problem.	It requires less algorithm support.
4.	CS does not construct the guard certain simple error, which made by programmer.	CR construct the guard against certain simple error, which made by programmer.
5.	In some system CS is a simple data structure (i.e. critical sect) used a built critical region.	CR is a code region that enjoys mutual exclusion.

Q.: State procedure Of Consumer problem. Give the solution for the same using bounded buffer.

Q.: Explain process synchronization using producer consumer problem.

Ans.: A pipe or other finite queue (buffer) is an example of the bounded buffer problem.

- This problem is also called as producer and consumer problem. In which a finite supply of container is available.
- Producer takes an empty container and fills it with an item.
- Consumer takes a full container, consume the product (produced item) and leave empty container. The main complexity of this problem is that we must maintain the count for both number of empty and full container are available.

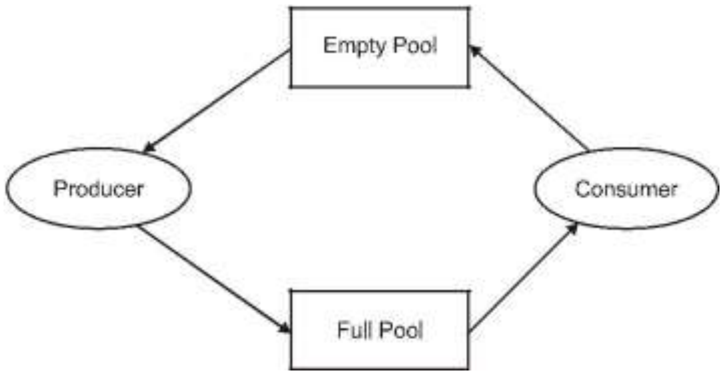


Fig.: General Scenario

- Producer produces a product and consumer consumers the product but both uses one of the container each time.

- Actually this problem arises when the rate of producer and consumer is differing so we need to keep or provide buffer pool to stored produced value.
- After that each producer or consumer can used as per their availability / capacity.
- In order to provide process synchronization this solution required.

# shared data

```
          #define Buffer-size
          typedef struct {
                           } item
item buffer [buffer-size]
          int in = 0 ; out = 0;
int counter = 0;
```

Description: Here simple declaration of all possible variables is done. Buffer size indicates maximum size of buffer, ***in*** variable points to next point to produced data, ***out*** variable points to next consumer data to consumer and ***counter*** variable is a shared variable which can be used by both process i.e. producer & consumer.

Producer code

```
          item next produced
          while (1)
          next produce = getitem ( )
          while (counter = buffer-size)
                    buffer [in] = next produce
                    in = (in + 1) % Buffer -
                    size counter ++
          }
```

Description: In producer code producer produces an item each time and store it in ***in*** variable and increment value of shared variable counter by one (i.e. ++).

The value produce by the producer can be consumed by the consumer as per rate of consumer process.

Consumer code

```
          item next consumed
          while (1)
{
          while (counter = 0)
                    out [out + 1]%
                    size counter - - - ;
}
```

Description:

Consumer will consume an item each time, by referring ***out*** variable. When the value of **out** variable decreases then we can say that one item is consumed by the consumer.

General problem may occurs in producer consumer process

1. If consumer is slow or busy with other process.
2. If producer process is slow or busy with other process
3. Buffer is neither full nor empty.

❍ ❍ ❍