Process Synchronization

It is the task of coordinating the execution of processes in a way such that no two processes can have access to the same shared data and resources.

Need for the Process Synchronization

Suppose there are 2 processes

P1:-
{
Statements;
Count++;
Statements;
}
 	
In assembly language:-
1. Move Count,R0
2. Inc. R0
3. Move R0,count
	P2:-
	{
	Statements;
	Count--;
	Statements;
	}
	
	In assembly language:-
	1.Move Count,R0
	2.Dec. R0
	3.Move R0,count

If:-

Gantt chart

P1:1,2,3	P2:1,2,3

Then:-
Count=5

If:-

Gantt chart

P1:1,2	P2:1,2,3	P1:3

Then:-
Count=6

If:-

Gantt chart

P1:1,2	P2:1,2	P1:3	P2:3

Then:-
Count=4

So the value of count is depending on the order of the Process Scheduling.

Now this is the condition where several processes try to access the resources and modify the shared data concurrently and outcome of the process depends on the particular order of execution that leads to data inconsistency, known as **Race Condition**. And here comes the need of Process Synchronization, in which we allow only one process to enter and manipulate the shared data in the Critical Section.

Four essential elements of the critical section:

- **Entry Section:** A part of the process which decides the entry of a particular process.
- **Critical Section:** This part allows one process to enter and modify the shared variable.
- **Exit Section:** It allows the other process that are waiting in the Entry Section, to enter into the Critical Sections and also checks that a process that finished its execution should be removed through this section.
- **Remainder Section:** All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

The entry to the critical section is handled by the wait() function, and it is represented as P().

The exit from a critical section is controlled by the signal() function, represented as V().

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion :** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress :** This solution is used when no one is in the critical section, and someone wants in. Then those processes not in their reminder section should decide who should go in, in a finite time.
- **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.

On the basis of synchronization, processes are categorized into two types:

- **Independent Process** : Execution of one process does not affect the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Process Synchronization are handled by two approaches:

Software Approach:-
Some specific Algorithm approach is used to maintain synchronization of the data.

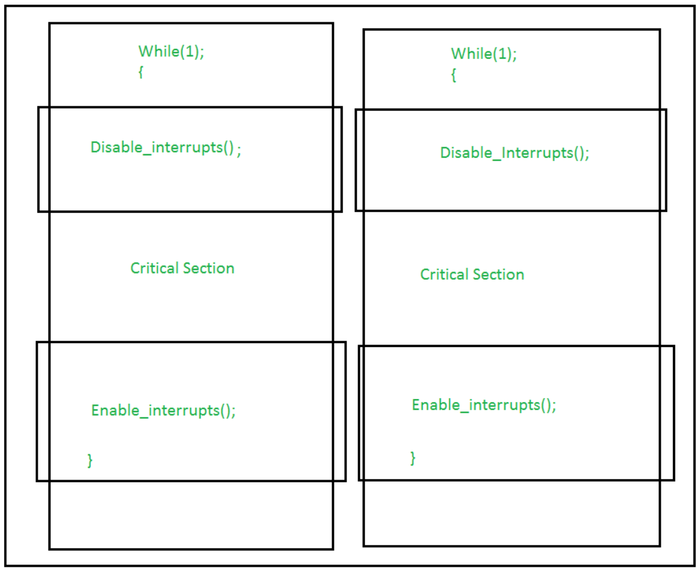
Peterson's Solution is best for Synchronization. It uses two variables in the Entry Section so as to maintain consistency, like Flag (boolean variable) and Turn variable(storing the process states).

It satisfy all the requirement of critical section.

Hardware Approach :-
The Hardware Approach of synchronization can be done through Lock & Unlock technique.Locking part is done in the Entry Section, so that only one process is allowed to enter into the Critical Section, after it complete its execution, the process is moved to the Exit Section, where Unlock Operation is done so that another process in the Lock Section can repeat this process of Execution.

Using Interrupts :-

These are easy to implement.When Interrupt are disabled then no other process is allowed to perform Context Switch operation that would allow only one process to enter into the Critical State.



Test_and_Set Operation :-
This allows boolean value (True/False) as a hardware Synchronization, which is atomic in nature i.e no other interrupt is allowed to access.In this process, a variable is allowed to accessed in Critical Section while its lock operation is ON.Till then, the other process is in Busy Waiting State. Hence Critical Section Requirements are achieved.

<u>Various synchronization mechanism for processes</u>

1.Lock Variable Synchronization Mechanism:-

It is a software mechanism implemented in user mode, i.e. no support required from the Operating System.
It is a busy waiting solution (keeps the CPU busy even when its technically waiting).

When Lock = 0 implies critical section is vacant (initial value) and Lock = 1 implies critical section occupied.

Pseudocode :-

```
Entry section
{
    while(LOCK!=0)
    LOCK=1
}
//Critical section
Exit section - LOCK=0
```

Pseudo-code in the form of an assembly language code :-

```
1. Load Lock, R0 ; (Store the value of Lock in Register R0.)
2. CMP R0, #0 ; (Compare the value of register R0 with 0.)
3. JNZ Step 1 ; (Jump to step 1 if value of R0 is not 0.)
4. Store #1, Lock ; (Set new value of Lock as 1.)
Enter critical section
5. Store #0, Lock ; (Set the value of lock as 0 again.)
```

The Lock Variable doesn’t provide mutual exclusion in some cases.

Ex :-

Suppose two processes P1 and P2

Gantt chart:-

P1:1	P2:1,2,3,4	P1:2,3,4

As we can see two process are accessing critical section at the same time.

2.Peterson's Algorithm :-

is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it.
When a process wants to execute it's critical section, it sets it's flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section.
After this the current process enters it's critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indication it does not wish to execute anymore.

Algorithm :-

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

while (true);
```

The algorithm satisfies all the three essential criteria(mutual exclusion,progress,bounded waiting) to solve the critical section problem

Tracing peterson solution :-

Suppose we have two processes P0 and P1

Code:-

```
  Entry section
  {
1.     flag[i] = true;
2.     turn = j;
3.     while (flag[j] && turn == j);
4.     /* critical section */
5.     flag[i] = false;
6.     /* remainder section */
  }
```

Gantt chart:-

P0:1	P1:1,2,3(waiting)	P0:2,3(waiting)	P1:3,CS,5	P0:3,CS,5

3.Semaphores :-

Semaphore is simply a variable that is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Difference between semaphores and mutex

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096-byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. The objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of a single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is a binary semaphore. *But it is not*! The purpose of mutex and semaphore are different. Maybe, due to similarity in their implementation a mutex would be referred to as a binary semaphore.

Semaphores are of two types:

1.Binary Semaphore –

This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

2.Counting Semaphore –

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Two main operations in semaphores:

```
 P(semaphore s)
 {
    while(s==0);/*wait until s=0*/
   s=s-1;
 }
V(semaphore s)
{
   s=s+1;
}
```

P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
Critical section is surrounded by these two operations
P(s);
//Critical section
V(s);

EX:-
Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved.
Limitations :-

1.One of the biggest limitations of semaphore is priority inversion.
2.Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.

4.Sleep and Wakeup :-
The concept of sleep and wake is very simple. If the critical section is not empty then the process will go and sleep. It will be waked up by the other process which

is currently executing inside the critical section so that the process can get inside the critical section.

Ex :-

```
 Producer consumer problem:
 #define N 100 //slots in buffer
 int count=0;    // items in buffer
 void producer(void)
 {
    int item;

 while(TRUE)
     {
         item = produce_item();
         if(count==N)
          sleep();
         insert_item(item);
         count=count+1;
         if(count==1)
          wakeup(consumer);
     }
 }
void consumer(void)
{
   int item;
   while(TRUE)
   {
       if(count==0)
        sleep();
       item =remove_item();
       count=count-1;
       if(count==N-1)
```

```
         wakeup(producer);
       consume_item(item);
   }
}
```

Disadvantage:-

Ex:-
Suppose initially count=0 and consumer process comes and check for count(which is 0)and got preempted then producer process comes and started producing item and after some time when buffer becomes full goes to sleep.now the previous consumer process get scheduled since previously it checked the count is zero so it also goes to sleep now both the processes are sleeping waiting for each other to wake them up(so they will sleep forever) hence deadlock.

5. Dekker’s algorithm

To obtain such a mutual exclusion, bounded waiting, and progress there have been several algorithms implemented, one of which is Dekker’s Algorithm.

First Version of Dekker’s Solution – The idea is to use common or shared thread number between processes and stop the other process from entering its critical section if the shared thread indicates the former one already running.

Pseudocode –

```
int thread_number=1;
Thread1()
{
    do
    {

        while (thread_number == 2);

        // critical section
        thread_number = 2;

        // remainder section
    }
    while (completed==false)
}

Thread2()
{

    do
    {
        while (thread_number == 1);


        // critical section

        thread_number = 1;

        // remainder section

    }
    while (completed == false)
}
```

The problem in the above implementation is lockstep synchronization, i.e each thread depends on the other for its execution. If one of the processes completes, then the second process runs, gives access to the completed one and waits for its turn, however, the former process is already completed and would never run to return the access back to the latter one. Hence, the second process waits infinitely then.

Second Version of Dekker's Solution – To remove lockstep synchronization, it uses two flags to indicate its current status and updates them accordingly at the entry and exit section.

Pseudocode :-

```
}


Bool thread1=thread2=FALSE;

Thread1()
{

    do {

         while (thread2 == true);


         thread1 = true;

        // critical section

        thread1 = false;

        // remainder section

    } while (completed == false)
}

Thread2()
{

    do {

         while (thread1 == true);

        thread2 = true;

        // critical section

        thread2 = false;

        // remainder section

    } while (completed == false)
}
```

The problem in the above version is mutual exclusion itself. If threads are preempted (stopped) during flag updation (i.e during current_thread = true) then, both the threads enter their critical section once the preempted thread is restarted, also the same can be observed at the start itself, when both the flags are false.

Third Version of Dekker's Solution – To re-ensure mutual exclusion, it sets the flags before entry section itself.

Pseudocode :-

```
bool thread1wantstoenter = false;
bool thread2wantstoenter = false;

Thread1()
{

    do {

        thread1wantstoenter = true;

        while (thread2wantstoenter == true);


        // critical section

        thread1wantstoenter = false;

        // remainder section

    } while (completed == false)
}

Thread2()
{

    do {
        thread2wantstoenter = true;

         while (thread1wantstoenter == true);

         // critical section

         thread2wantstoenter = false;

        // remainder section

    } while (completed == false)
}
```

The problem with this version is deadlock possibility. Both threads could set their flag as true simultaneously and both will wait infinitely later on.

Fourth Version of Dekker's Solution – Uses small time interval to recheck the condition, eliminates deadlock and ensures mutual exclusion as well.

Pseudocode :-

```
boolean thread1wantstoenter = false;
boolean thread2wantstoenter = false;

Thread1()
{

    do {

        thread1wantstoenter = true;

        while (thread2wantstoenter == true) {
             thread1wantstoenter = false;

             thread1wantstoenter = true;
        }

        // critical section
```

```
         thread1wantstoenter = false;

        // remainder section

    } while (completed == false)
}

Thread2()
{

    do {

        thread2wantstoenter = true;

        while ( thread1wantstoenter == true) {

            thread2wantstoenter = false;

            thread2wantstoenter = true;
        }

        // critical section

        thread2wantstoenter = false;

        // remainder section

    } while (completed == false)
}
```

The problem with this version is the indefinite postponement. Also, random amount of time is erratic depending upon the situation in which the algorithm is being implemented, hence not an acceptable solution in business critical systems.

Final and completed Solution – -Idea is to use favoured thread notion to determine entry to the critical section. Favoured thread alternates between the thread providing mutual exclusion and avoiding deadlock, indefinite postponement or lockstep synchronization.

Pseudocode :-

```
int favoured_thread = 1;

boolean thread1_wants_to_enter = false;
boolean thread2_wants_to_enter = false;

Thread1()
{
    do {

        thread1_wants_to_enter = true;

        while (thread2_wants_to_enter == true) {

            if (favaoured_thread == 2) {

                thread1_wants_to_enter = false;

                while (favoured_thread == 2);

                thread1_wants_to_enter = true;
            }
        }

        // critical section

        favoured_thread = 2;

        thread1_wants_to_enter = false;

        // remainder section
    } while (completed == false)
}

Thread2()
{

    do {

        thread2_wants_to_enter = true;

        while (thread1_wants_to_enter == true) {

                if (favaoured_thread == 1) {

                thread2wantstoenter = false;

                while (favoured_thread == 1);

                thread2wantstoenter = true;
            }
        }

        // critical section

        favoured_thread = 1;

        thread2_wants_to_enter = false;

        // remainder section

    } while (completed == false)
}
```

This version guarantees a complete solution to the critical solution problem.

6.Bakery Algorithm

It is a critical section solution for N processes and also preserves the first come first serve property.

Before entering its critical section, the process receives a number. Holder of the smallest number enters the critical section.

If processes Pi and Pj receive the same number,

```
  if i < j
    Pi is served first;
   else
     Pj is served first.
```

Firstly the ticket number is compared. If same then the process ID is compared next. Shared data – choosing is an array [0..n – 1] of boolean values; & number is an array [0..n – 1] of integer values. Both are initialized to False & Zero respectively.

Pseudocode –

```
Repeat
choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n -
1])+1;
    choosing[i] := false;
    for j := 0 to n - 1
        do begin
            while choosing[j] do no-op;
            while number[j] != 0
                and (number[j], j) < (number[i], i) do no-op;
        end;
        critical section
number[i] := 0;
 remainder section
until false;
```

Some other famous problems:

1.Producer Consumer Problem

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.
To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Solution for Producer –

```
do{
     //produce an item

wait(empty);

wait(mutex);
     //place in buffer

signal(mutex);

signal(full);

} while(true)
```

Solution for Consumer –

```
do{

wait(full);

wait(mutex);
```

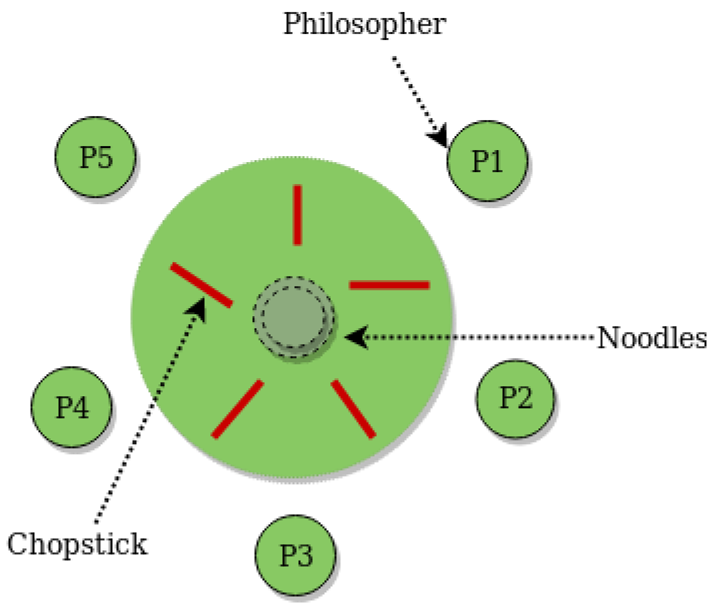
```
     // remove item from buffer
signal(mutex);
signal(empty);
     // consumes item
} while(true).
```

When a producer produces an item, the value of “empty” decreases by 1 as a slot is filled now and the value of mutex is also reduced so that consumers can't access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of the producer has been completed and consumer can access the buffer.

Similarly when consumer consumes the value of full and mutex is reduced to prevent access of buffer to producer and after the consumption value of both empty and mutex increases by one so that producer can access the buffer.

2.Dining Philosopher Problem Using Semaphores

Problem Statement -- The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Pseudocode –

Semaphore s[k]; //each semaphore for each fork/chopsticks.

```
Void philosopher (void)
{
  while(TRUE)
  {
    Thinking(); //No problem
1.     wait(s[i]);//pick up left fork.
2.     wait(s[(i+1)%k])//pick up right fork.
3.     eat();
4.     signal(s[i]);//put down left fork.
5.     signal(s[(i+1)%k])//put down right fork.
}
}
```

Disadvantage:-
Dead lock can happen in this solution.
Ex:-
Assume there are five philosophers p0,p1,p2,p3,p4.

p0:1	p1:1	p2:1	p3:1	p4:1(waiting)

As we can see now all the philosophers are waiting(forever) for there neighbors to put down there right fork.hence deadlock.
Solution:-

The only solution is to change the sequence of line 1 and 2 for exactly one process which will block that process from taking it’s left fork so deadlock condition will not appear.

3.Readers-Writers Problem

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the readers-writers problem
Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Pseudocode –

```
int rc=0;
Semaphore mutex=1;
Semaphore db=1;
void reader(void)
{
  while(TRUE)
  {
    down(mutex);
    rc=rc+1;
    if(rc==1)
    down(db);
    up(mutex);

    //Critical section

    down(mutex);
    rc=rc-1;
    if(rc==0)
    up(db);
    up(mutex);
  }
}
void writer(void)
{
   while(TRUE)
   {
    down(db);

    //Critical section

    up(db);
   }
}
```
Hence,the semaphore db restricts both (reader and writer)/(writer and writer)/(writer and reader) to enter the critical section(data) at the same time.

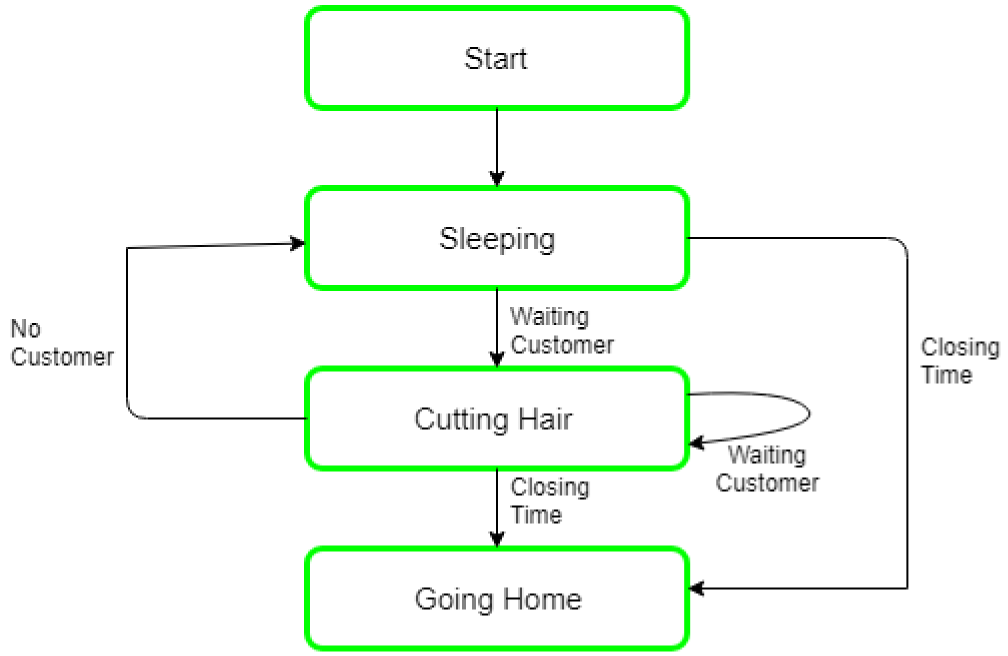
4.Sleeping Barber problem

Problem Statement – There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.

- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

To solve this problem, we need 3 semaphores -- Count for the no of customers present in the waiting room**,** barber(0 or 1) to tell whether the barber is idle or is working **,** and the third mutex to provide the mutual exclusion which is required for the process to execute.


Start
Sleeping
Waiting Customer
No Customer
Closing Time
Cutting Hair
Waiting Customer
Closing Time
Going Home