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
int count=5;

P1:-	P2:-

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

<u>Any solution to the critical section problem must satisfy three requirements:</u>

- **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.

<u>On the basis of synchronization, processes are categorized into two types:</u>

- **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.

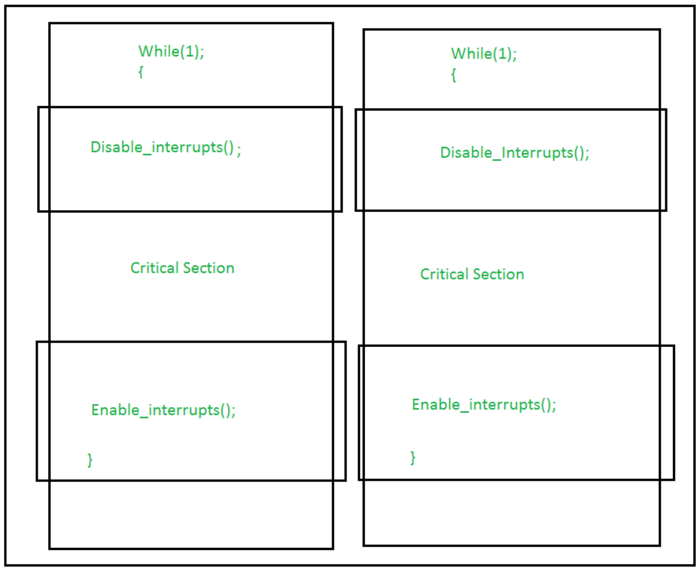
Peterson’s Solution is best for Synchronization. It uses two variables in the Entry Section so as to maintain consistency, like Flag (boolean variable) and Turn variable(storing the process states).

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 :-

```
  1.  Entry section
{
  2.  while(LOCK!=0);
  3.  LOCK=1
}
//Critical section
4. 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.Turn Variable/Strict Alternation method :-
->software mechanism implemented at user mode.
->busy waiting solution.
->2-process solution.

int turn=0;

For process p0:-
while(turn!=0); //entry section
 	
  //critical section
 	
turn=1; //exit section
	For process p1:-
	while(turn!=1); //entry section
	
	  //critical section
	
	turn=0; //exit section

Mutual exclusion:- true
Progress:- false
Bounded waiting:-true
Architecture neutrality:-true

3.Intrested Variable:-

Interested[0]=True -P0
Interested[1]=True -P1

For process p0:-
Int[0]=T
while(Int[1]==T);
  //entry section
 	
  //critical section
 	
Int[0]=F; //exit section
	For process p1:-
	Int[1]=T
	while(Int[0]==T);
	  //entry section
	
	  //critical section
	
	Int[1]=F; //exit section

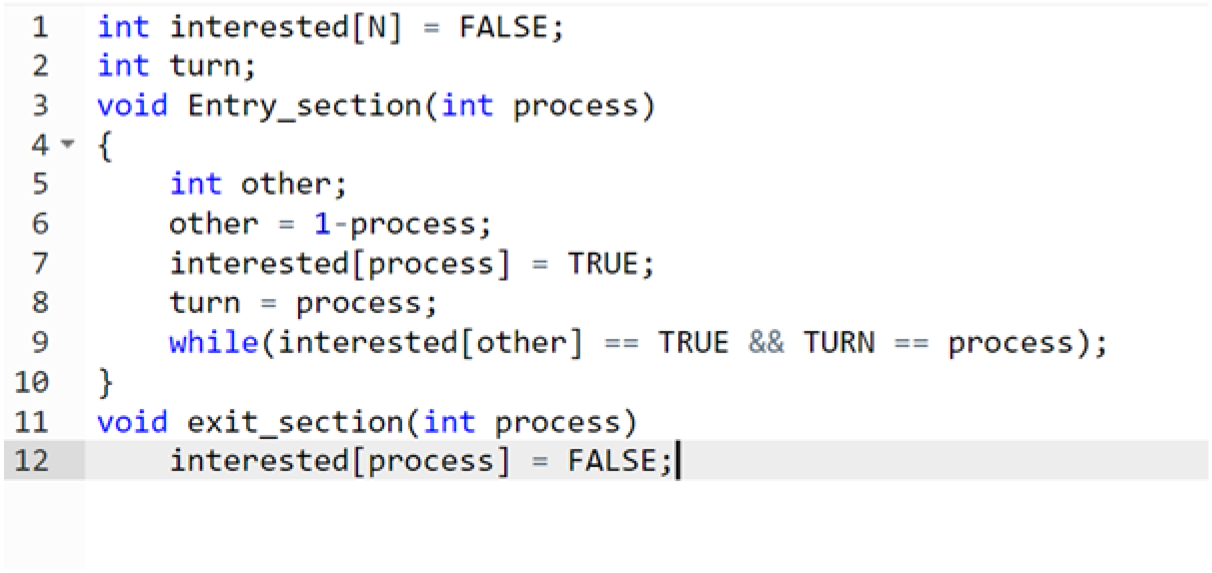
Mutual exclusion:- true
Progress:- true
Bounded waiting:- false
Architecture neutrality:-true

4.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.

->user mode
->busy waiting solution
->2 process solution

Algorithm :-



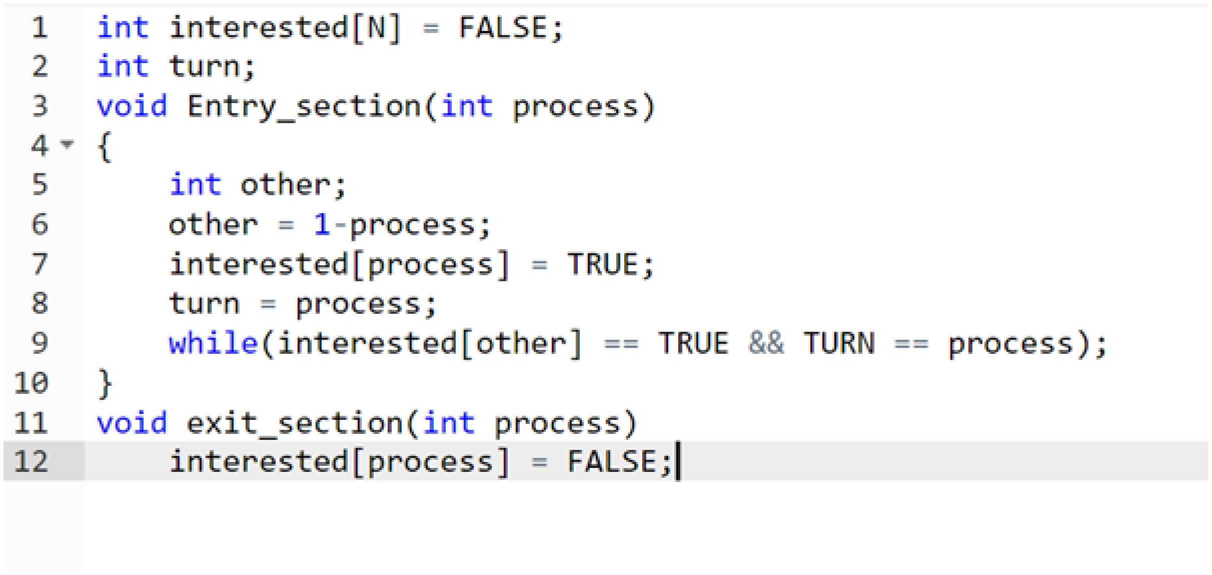
```c
int interested[N] = FALSE;
int turn;
void Entry_section(int process)
{
    int other;
    other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while(interested[other] == TRUE && TURN == process);
}
void exit_section(int process)
    interested[process] = FALSE;
```

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

<u>Tracing peterson solution :-</u>

Suppose we have two processes P0 and P1

<u>Code:-</u>



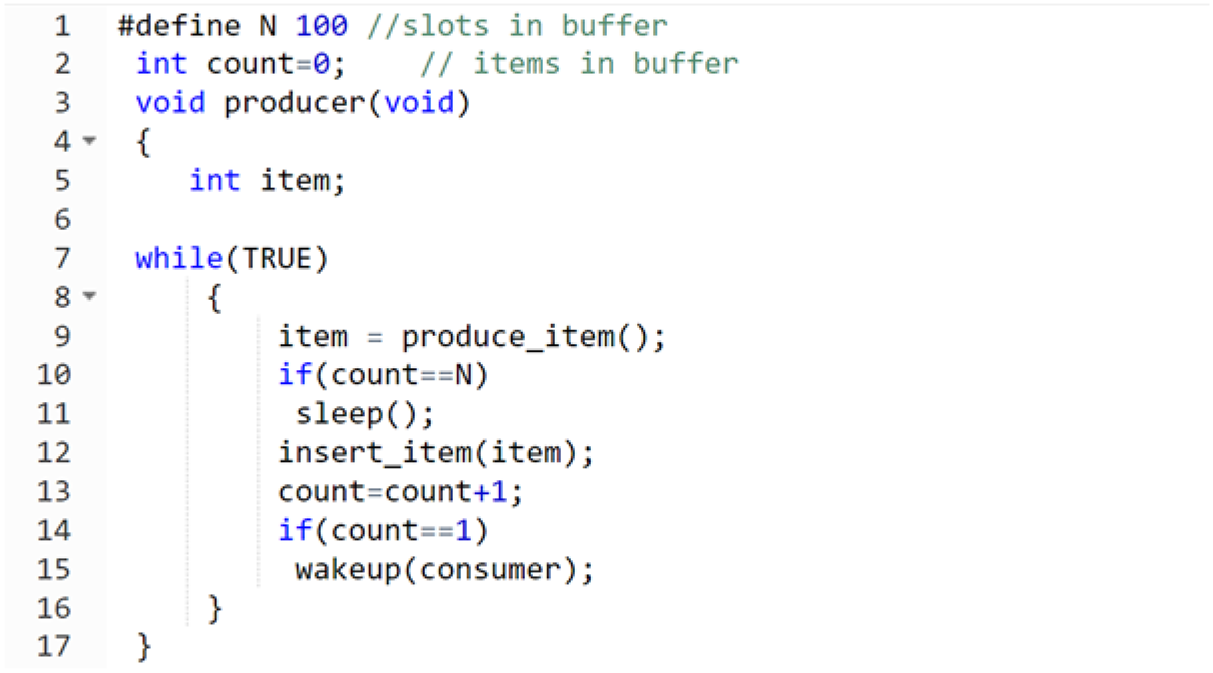
```c
int interested[N] = FALSE;
int turn;
void Entry_section(int process)
{
    int other;
    other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while(interested[other] == TRUE && TURN == process);
}
void exit_section(int process)
    interested[process] = FALSE;
```

5.Sleep and Wakeup :-

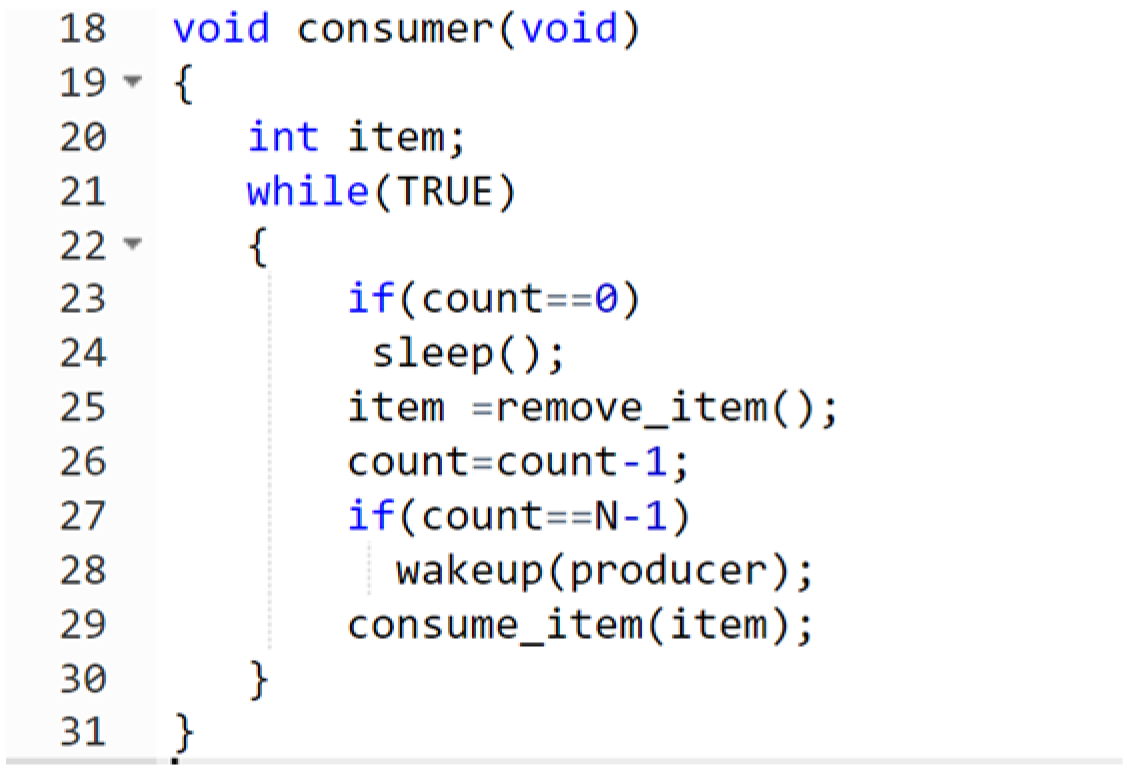
The concept of sleep and wake is very simple. If the critical section is not empty then the process will go and sleep. It will be waked up by the other process which is currently executing inside the critical section so that the process can get inside the critical section.

Ex :-

Producer consumer problem:



```c
#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.

6.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.

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.

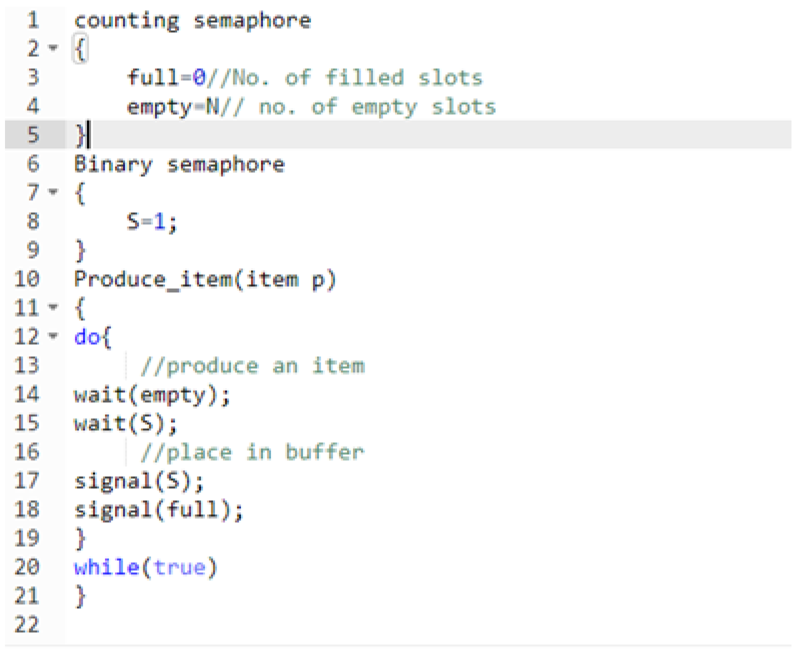
2.Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.

<u>Some other famous problems:</u>

1.Producer Consumer Problem

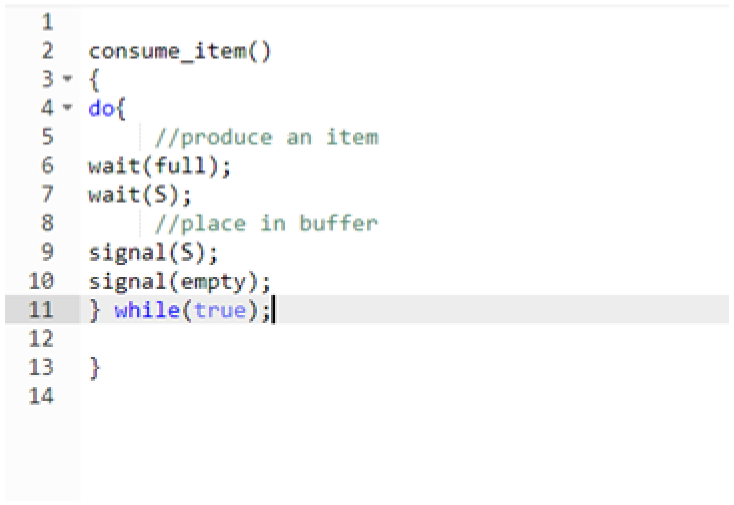
Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this 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 –



```
counting semaphore
{
    full=0//No. of filled slots
    empty=N// no. of empty slots
}
Binary semaphore
{
    S=1;
}
Produce_item(item p)
{
do{
      //produce an item
wait(empty);
wait(S);
      //place in buffer
signal(S);
signal(full);
}
while(true)
}
```

Solution for Consumer –



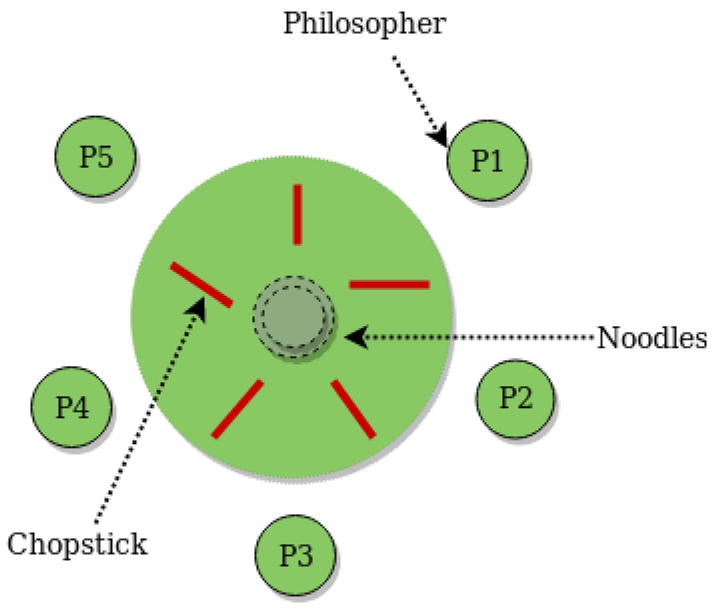
```

consume_item()
{
do{
      //produce an item
wait(full);
wait(S);
      //place in buffer
signal(S);
signal(empty);
} 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.

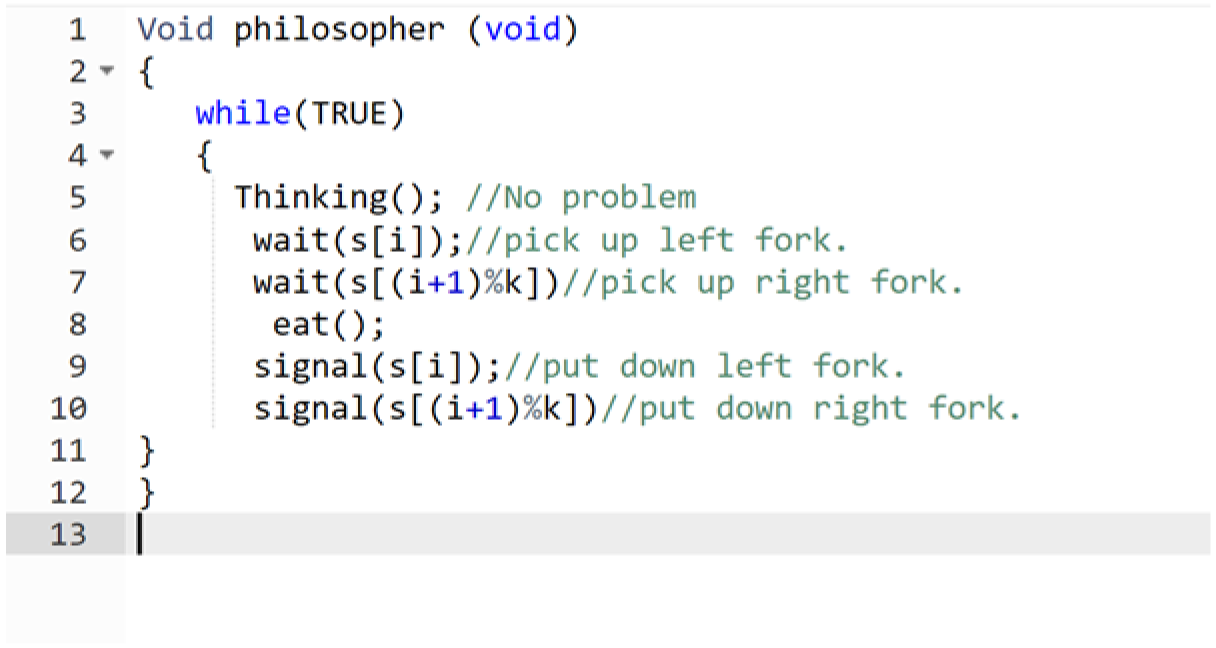
2.Dining Philosopher Problem Using Semaphores

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
      wait(s[i]);//pick up left fork.
      wait(s[(i+1)%k])//pick up right fork.
       eat();
      signal(s[i]);//put down left fork.
      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

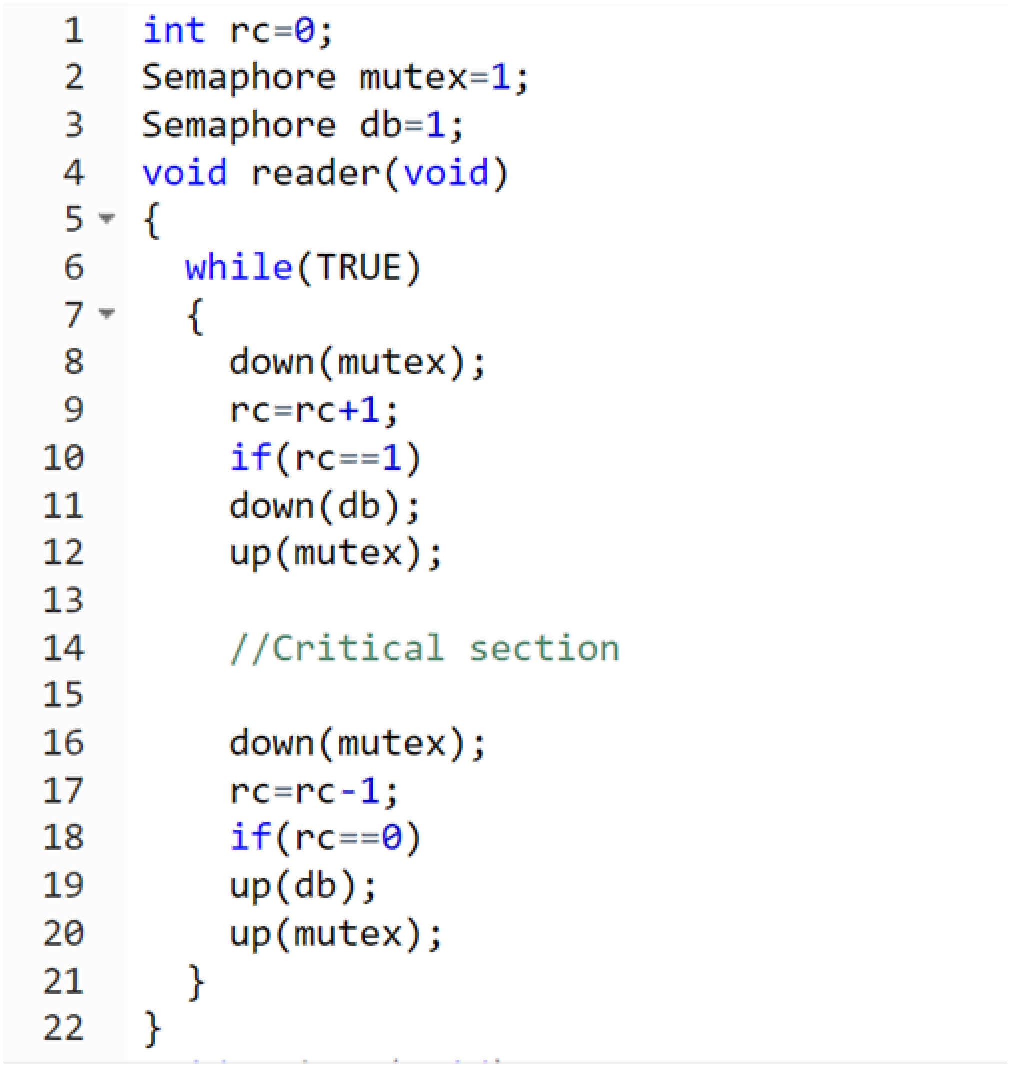
Consider a situation where we have a file shared between many people.

- 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.

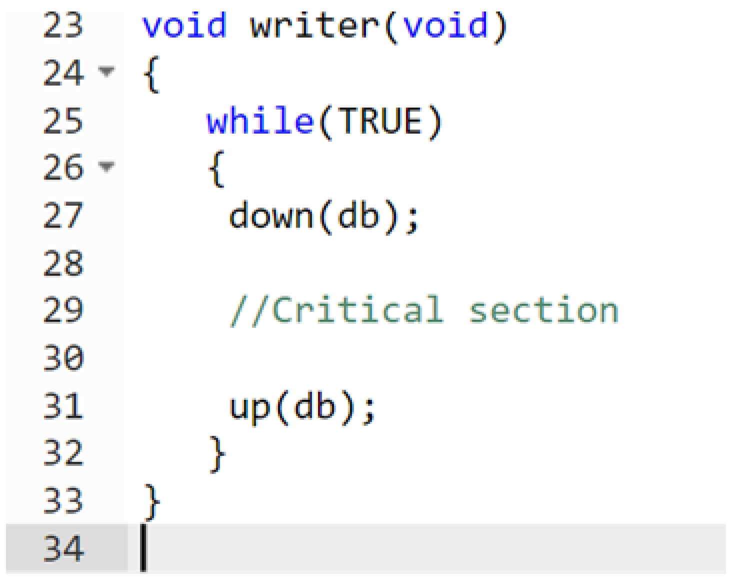
Precisely in OS we call this situation as the readers-writers problem
Problem parameters:

- 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);
   }
}
```

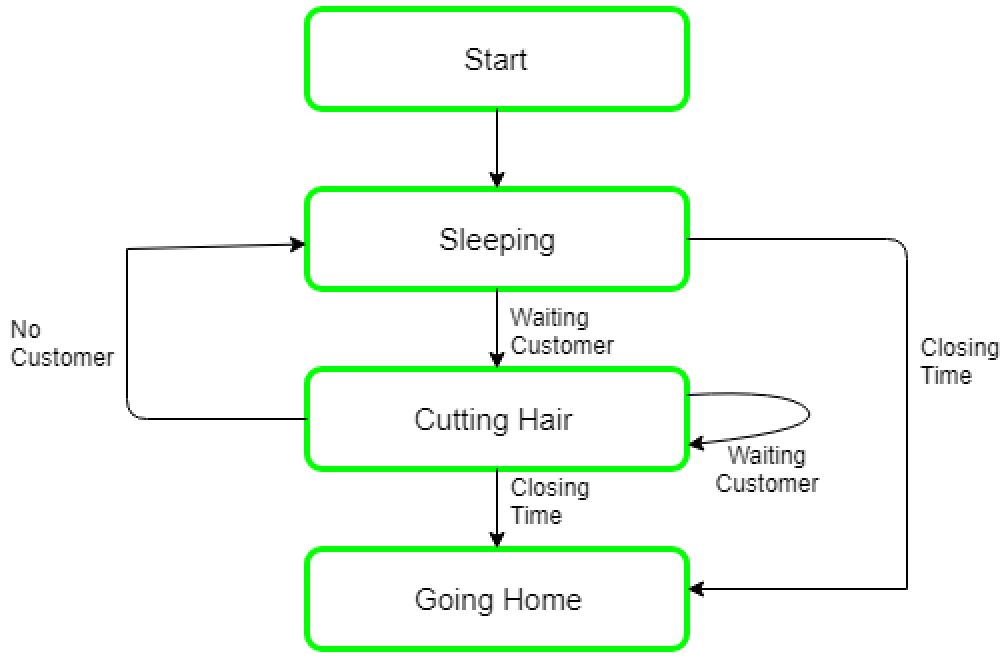
Hence,the semaphore db restricts both (reader and writer)/(writer and writer)/(writer and reader) to enter the critical section(data) at the same time.

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