UNIT NO: - 5

PROCESS SYNCHRONIZATION

INTRODUCTION:-

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.

The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.

Race Condition

A Race Condition typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.

Critical Section

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

The Critical Section Problem

Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.

The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

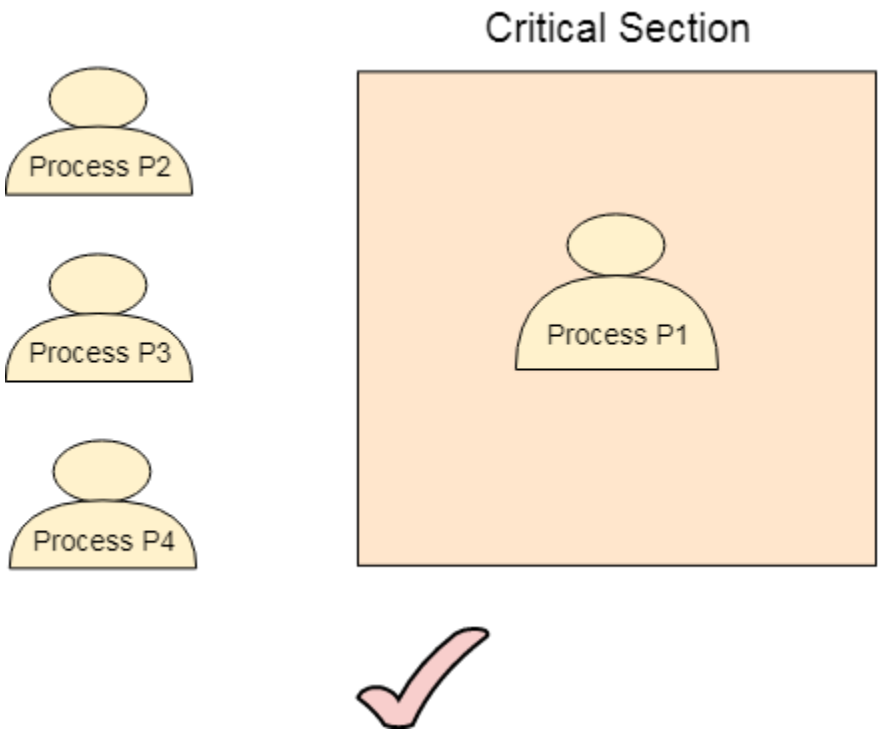
In order to synchronize the cooperative processes, our main task is to solve the critical section problem. We need to provide a solution in such a way that the following conditions can be satisfied.

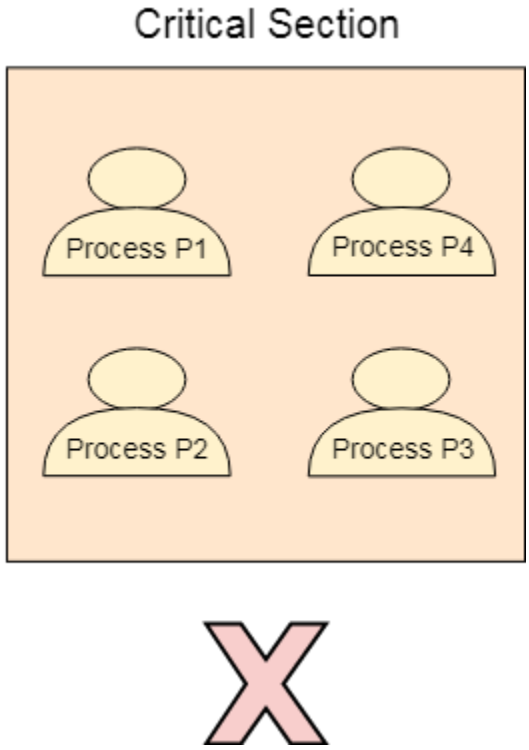
Requirements of Synchronization mechanisms

Primary

1. **Mutual Exclusion**

   Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.





2. **Progress**

   Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Secondary

1. **Bounded Waiting**

   We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

2. **Architectural Neutrality**

   Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Lock Variable

This is the simplest synchronization mechanism. This is a Software Mechanism implemented in User mode. This is a busy waiting solution which can be used for more than two processes.

In this mechanism, a Lock variable **lock**is used. Two values of lock can be possible, either 0 or 1. Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.

A process which wants to get into the critical section first checks the value of the lock variable. If it is 0 then it sets the value of lock as 1 and enters into the critical section, otherwise it waits.

The pseudo code of the mechanism looks like following.

1. Entry Section →
2. While (lock! = 0);
3. Lock = 1;
4. //Critical Section
5. Exit Section →
6. Lock =0;

If we look at the Pseudo Code, we find that there are three sections in the code. Entry Section, Critical Section and the exit section.

Initially the value of **lock variable** is **0**. The process which needs to get into the **critical section**, enters into the entry section and checks the condition provided in the while loop.

The process will wait infinitely until the value of **lock** is 1 (that is implied by while loop). Since, at the very first time critical section is vacant hence the process will enter the critical section by setting the lock variable as 1.

When the process exits from the critical section, then in the exit section, it reassigns the value of **lock** as 0.

Every Synchronization mechanism is judged on the basis of four conditions.

1. Mutual Exclusion
2. Progress
3. Bounded Waiting
4. Portability

Out of the four parameters, Mutual Exclusion and Progress must be provided by any solution. Let?s analyze this mechanism on the basis of the above mentioned conditions.

MUTUAL EXCLUSION:-

The lock variable mechanism doesn't provide Mutual Exclusion in some of the cases. This can be better described by looking at the pseudo code by the Operating System point of view I.E. Assembly code of the program. Let's convert the Code into the assembly language.

1. *Load Lock, R0*
2. *CMP R0, #0*
3. *JNZ Step 1*
4. *Store #1, Lock*
5. *Store #0, Lock*

Let us consider that we have two processes P1 and P2. The process P1 wants to execute its critical section. P1 gets into the entry section. Since the value of lock is 0 hence P1 changes its value from 0 to 1 and enters into the critical section.

Meanwhile, P1 is preempted by the CPU and P2 gets scheduled. Now there is no other process in the critical section and the value of lock variable is 0. P2 also wants to execute its critical section. It enters into the critical section by setting the lock variable to 1.

Now, CPU changes P1's state from waiting to running. P1 is yet to finish its critical section. P1 has already checked the value of lock variable and remembers that its value was 0 when it previously checked it. Hence, it also enters into the critical section without checking the updated value of lock variable.

Now, we got two processes in the critical section. According to the condition of mutual exclusion, morethan one process in the critical section must not be present at the same time. Hence, the lock variable mechanism doesn't guarantee the mutual exclusion.

The problem with the lock variable mechanism is that, at the same time, more than one process can see the vacant tag and more than one process can enter in the critical section. Hence, the lock variable doesn't provide the mutual exclusion that's why it cannot be used in general.

Since, this method is failed at the basic step; hence, there is no need to talk about the other conditions to be fulfilled.

Test Set Lock Mechanism

Modification in the assembly code

In lock variable mechanism, Sometimes Process reads the old value of lock variable and enters the critical section. Due to this reason, more than one process might get into critical section. However, the code shown in the part one of the following section can be replaced with the code shown in the part two. This doesn't affect the algorithm but, by doing this, we can manage to provide the mutual exclusion to some extent but not completely.

In the updated version of code, the value of Lock is loaded into the local register R0 and then value of lock is set to 1.

However, in step 3, the previous value of lock (that is now stored into R0) is compared with 0. if this is 0 then the process will simply enter into the critical section otherwise will wait by executing continuously in the loop.

The benefit of setting the lock immediately to 1 by the process itself is that, now the process which enters into the critical section carries the updated value of lock variable that is 1.

In the case when it gets preempted and scheduled again then also it will not enter the critical section regardless of the current value of the lock variable as it already knows what the updated value of lock variable is.

Section 1	Section 2
1. Load Lock, R0 2. CMP R0, #0 3. JNZ step1 4. store #1, Lock	1. Load Lock, R0 2. Store #1, Lock 3. CMP R0, #0 4. JNZ step 1

TSL Instruction

However, the solution provided in the above segment provides mutual exclusion to some extent but it doesn't make sure that the mutual exclusion will always be there. There is a possibility of having more than one process in the critical section.

What if the process gets preempted just after executing the first instruction of the assembly code written in section 2? In that case, it will carry the old value of lock variable with it and it will enter into the critical section regardless of knowing the current value of lock variable. This may make the two processes present in the critical section at the same time.

To get rid of this problem, we have to make sure that the preemption must not take place just after loading the previous value of lock variable and before setting it to 1. The problem can be solved if we can be able to merge the first two instructions.

In order to address the problem, the operating system provides a special instruction called **Test Set Lock (TSL)** instruction which simply loads the value of lock variable into the local register R0 and sets it to 1 simultaneously

The process which executes the TSL first will enter into the critical section and no other process after that can enter until the first process comes out. No process can execute the critical section even in the case of preemption of the first process.

The assembly code of the solution will look like following.

1. *TSL Lock, R0*
2. *CMP R0, #0*
3. *JNZ step 1*

Let's examine TSL on the basis of the four conditions.

- **Mutual Exclusion**

  Mutual Exclusion is guaranteed in TSL mechanism since a process can never be preempted just before setting the lock variable. Only one process can see the lock variable as 0 at a particular time and that's why, the mutual exclusion is guaranteed.

- **Progress**

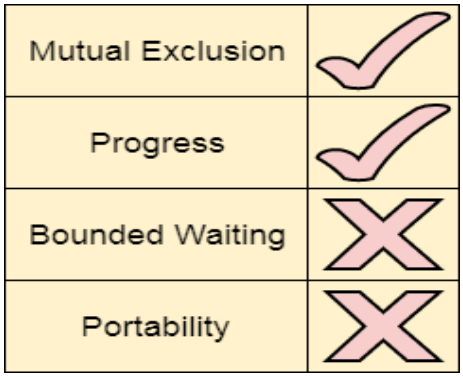
  According to the definition of the progress, a process which doesn't want to enter in the critical section should not stop other processes to get into it. In TSL mechanism, a process will execute the TSL instruction only when it wants to get into the critical section. The value of the lock will always be 0 if no process doesn't want to enter into the critical section hence the progress is always guaranteed in TSL.

- **Bounded Waiting**

  Bounded Waiting is not guaranteed in TSL. Some process might not get a chance for so long. We cannot predict for a process that it will definitely get a chance to enter in critical section after a certain time.

- **Architectural Neutrality**

  TSL doesn't provide Architectural Neutrality. It depends on the hardware platform. The TSL instruction is provided by the operating system. Some platforms might not provide that. Hence it is not Architectural natural.



PRIORITY INVERSION:-

In TSL mechanism, there can be a problem of priority inversion. Let?s say that there are two cooperative processes, P1 and P2.

The priority of P1 is 2 while that of P2 is 1. P1 arrives earlier and got scheduled by the CPU. Since it is a cooperative process and wants to execute in the critical section hence it will enter in the critical section by setting the lock variable to 1.

Now, P2 arrives in the ready queue. The priority of P2 is higher than P1 hence according to priority scheduling, P2 is scheduled and P1 got preempted. P2 is also a cooperative process and wants to execute inside the critical section.

Although, P1 got preempted but it the value of lock variable will be shown as 1 since P1 is not completed and it is yet to finish its critical section.

P1 needs to finish the critical section but according to the scheduling algorithm, CPU is with P2. P2 wants to execute in the critical section, but according to the synchronization mechanism, critical section is with P1.

This is a kind of lock where each of the process neither executes nor completes. Such kind of lock is called **Spin Lock**.

This is different from deadlock since they are not in blocked state. One is in ready state and the other is in running state, but neither of the two is being executed.

Turn Variable or Strict Alternation Approach

Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock.

This approach can only be used for only two processes. In general, let the two processes be Pi and Pj. They share a variable called turn variable. The pseudo code of the program can be given as following.

For Process Pi

1. Non - CS
2. while (turn ! = i);
3. Critical Section
4. turn = j;
5. Non - CS

For Process Pj

1. Non - CS
2. while (turn ! = j);
3. Critical Section
4. turn = i ;
5. Non - CS

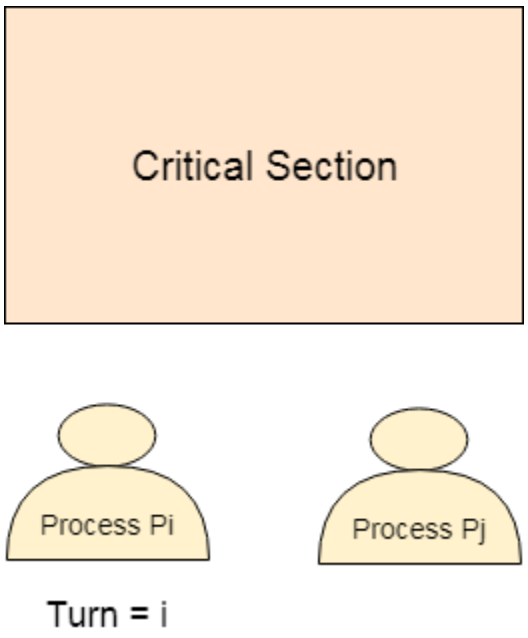
The actual problem of the lock variable approach was the fact that the process was entering in the critical section only when the lock variable is 1. More than one process could see the lock variable as 1 at the same time hence the mutual exclusion was not guaranteed there.

This problem is addressed in the turn variable approach. Now, A process can enter in the critical section only in the case when the value of the turn variable equal to the PID of the process.

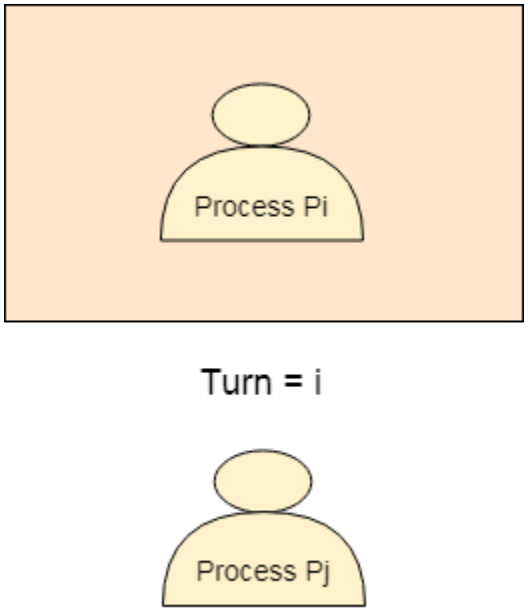
There are only two values possible for turn variable, i or j. if its value is not i then it will definitely be j or vice versa.

In the entry section, in general, the process Pi will not enter in the critical section until its value is j or the process Pj will not enter in the critical section until its value is i.

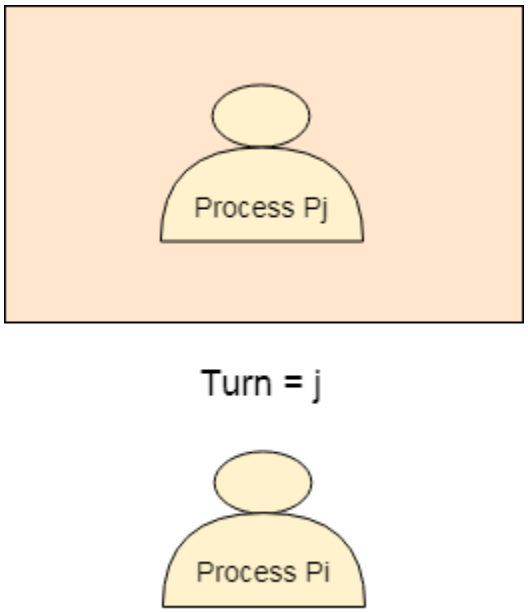
Initially, two processes Pi and Pj are available and want to execute into critical section.



The turn variable is equal to i hence Pi will get the chance to enter into the critical section. The value of Pi remains I until Pi finishes critical section.



Pi finishes its critical section and assigns j to turn variable. Pj will get the chance to enter into the critical section. The value of turn remains j until Pj finishes its critical section.



Analysis of Strict Alternation approach

Let's analyze Strict Alternation approach on the basis of four requirements.

Mutual Exclusion

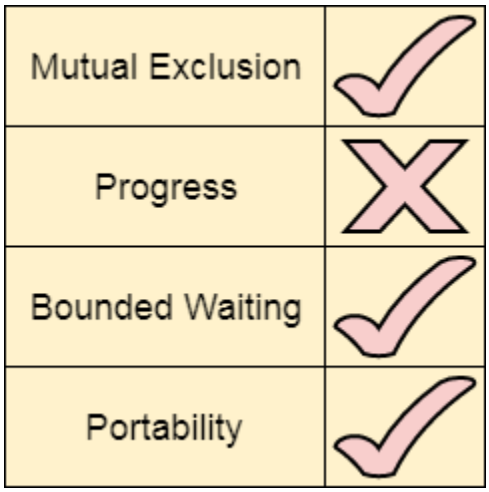
The strict alternation approach provides mutual exclusion in every case. This procedure works only for two processes. The pseudo code is different for both of the processes. The process will only enter when it sees that the turn variable is equal to its Process ID otherwise not Hence No process can enter in the critical section regardless of its turn.

Progress

Progress is not guaranteed in this mechanism. If Pi doesn't want to get enter into the critical section on its turn then Pj got blocked for infinite time. Pj has to wait for so long for its turn since the turn variable will remain 0 until Pi assigns it to j.

Portability

The solution provides portability. It is a pure software mechanism implemented at user mode and doesn't need any special instruction from the Operating System.



Interested Variable Mechanism

We have to make sure that the progress must be provided by our synchronization mechanism. In the turn variable mechanism, progress was not provided due to the fact that the process which doesn't want to enter in the critical section does not consider the other interested process as well.

The other process will also have to wait regardless of the fact that there is no one inside the critical section. If the operating system can make use of an extra variable along with the turn

variable then this problem can be solved and our problem can provide progress to most of the extent.

Interested variable mechanism makes use of an extra Boolean variable to make sure that the progress is provided.

For Process Pi

1. Non CS
2. Int[i] = T ;
3. while (Int[j] == T) ;
4. Critical Section
5. Int[i] = F ;

For Process Pj

1. Non CS
2. Int [1] = T ;
3. while (Int[i] == T) ;
4. Critical Section
5. Int[j]=F ;

In this mechanism, an extra variable **interested** is used. This is a Boolean variable used to store the interest of the processes to get enter inside the critical section.

A process which wants to enter in the critical section first checks in the entry section whether the other process is interested to get inside. The process will wait for the time until the other process is interested.

In exit section, the process makes the value of its interest variable false so that the other process can get into the critical section.

The table shows the possible values of interest variable of both the processes and the process which get the chance in the scenario.

Interest [Pi]	Interest [Pj]	Process which get the chance
True	True	The process which first shows interest.
True	False	Pi
False	True	Pj

False	False	X

Let's analyze the mechanism on the basis of the requirements.

Mutual Exclusion

In interested variable mechanism, if one process is interested in getting into the CPU then the other process will wait until it becomes uninterested. Therefore, more than one process can never be present in the critical section at the same time hence the mechanism guarantees mutual exclusion.

Progress

In this mechanism, if a process is not interested in getting into the critical section then it will not stop the other process from getting into the critical section. Therefore the progress will definitely be provided by this method.

Bounded Waiting

To analyze bounded waiting, let us consider two processes Pi and Pj, are the cooperative processes wants to execute in the critical section. The instructions executed by the processes are shown below in relative manner.

Process Pi	**Process Pj**	**Process Pi**	**Process Pj**
1. Int [Pi] = True 2. while (Int [Pj] == True); 3.Critical Section	1. Int [Pj] = True 2.while(Int[Pi]==True);	1. Int [Pi] = False 2. Int [Pi] = True 3. while (Int [Pj] == True); //waiting for Pj	1. While (Int [Pi] == True); //waiting for Pj

Initially, the interest variable of both the processes is **false**. The process Pi shows the interest to get inside the critical section.

It sets its Interest Variable to true and check whether the Pj is also interested or not. Since the other process's interest variable is false hence Pi will get enter into the critical section.

Meanwhile, the process Pi is preempted and Pj is scheduled. Pj is a cooperative process and therefore, it also wants to enter in the critical section. It shows its interest by setting the interest variable to true.

It also checks whether the other process is also interested or not. We should notice that Pi is preempted but its interested variable is true that means it needs to further execute in the critical section. Therefore Pj will not get the chance and gets stuck in the while loop.

Meanwhile, CPU changes Pi's state from blocked to running. Pi is yet to finish its critical section hence it finishes the critical section and makes an exit by setting the interest variable to False.

Now, a case can be possible when Pi again wants to enter in the critical section and set its interested variable to true and checks whether the interested variable of Pj is true. Here, Pj's interest variable is True hence Pi will get stuck in the while loop and waits for Pj become uninterested.

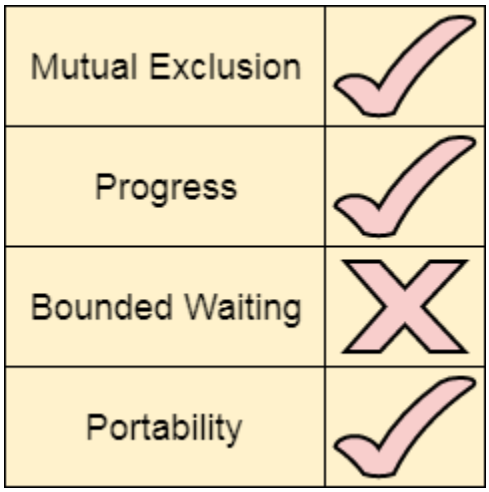
Since, Pj still stuck in the while loop waiting for the Pi' interested variable to become false. Therefore, both the processes are waiting for each other and none of them is getting into the critical section.

This is a condition of deadlock and bounded waiting can never be provided in the case of deadlock.

Therefore, we can say that the interested variable mechanism doesn't guarantee deadlock.

Architectural Neutrality

The mechanism is a complete software mechanism executed in the user mode therefore it guarantees portability or architectural neutrality.



Paterson Solution

This is a software mechanism implemented at user mode. It is a busy waiting solution can be implemented for only two processes. It uses two variables that are turn variable and interested variable.

The Code of the solution is given below

```
1.  # define N 2
2.  # define TRUE 1
3.  # define FALSE 0
4.  int interested[N] = FALSE;
5.  int turn;
6.  voidEntry_Section (int process)
7.  {
8.      int other;
9.      other = 1-process;
10.     interested[process] = TRUE;
11.     turn = process;
12.     while (interested [other] =True && TURN=process);
13. }
14. voidExit_Section (int process)
15. {
16.     interested [process] = FALSE;
17. }
```

Till now, each of our solution is affected by one or the other problem. However, the Peterson solution provides you all the necessary requirements such as Mutual Exclusion, Progress, Bounded Waiting and Portability.

Analysis of Peterson Solution

```
1.  voidEntry_Section (int process)
2.  {
3.      1. int other;
4.      2. other = 1-process;
5.      3. interested[process] = TRUE;
6.      4. turn = process;
7.      5. while (interested [other] =True && TURN=process);
8.  }
9.
10. Critical Section
11.
12. voidExit_Section (int process)
13. {
14.     6. interested [process] = FALSE;
15. }
```

This is a two process solution. Let us consider two cooperative processes P1 and P2. The entry section and exit section are shown below. Initially, the value of interested variables and turn variable is 0.

Initially process P1 arrives and wants to enter into the critical section. It sets its interested variable to True (instruction line 3) and also sets turn to 1 (line number 4). Since the condition given in line number 5 is completely satisfied by P1 therefore it will enter in the critical section.

1. P1 → 1 2 3 4 5 CS

Meanwhile, Process P1 got preempted and process P2 got scheduled. P2 also wants to enter in the critical section and executes instructions 1, 2, 3 and 4 of entry section. On instruction 5, it got stuck since it doesn't satisfy the condition (value of other interested variable is still true). Therefore it gets into the busy waiting.

1. P2 → 1 2 3 4 5

P1 again got scheduled and finish the critical section by executing the instruction no. 6 (setting interested variable to false). Now if P2 checks then it are going to satisfy the condition since other process's interested variable becomes false. P2 will also get enter the critical section.

1. P1 → 6
2. P2 → 5 CS

Any of the process may enter in the critical section for multiple numbers of times. Hence the procedure occurs in the cyclic order.

Mutual Exclusion

The method provides mutual exclusion for sure. In entry section, the while condition involves the criteria for two variables therefore a process cannot enter in the critical section until the other process is interested and the process is the last one to update turn variable.

Progress

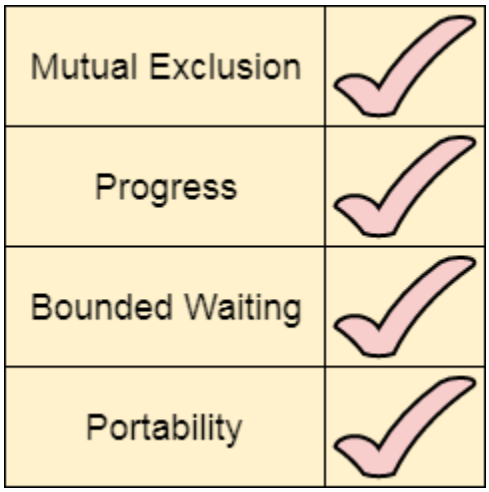
An uninterested process will never stop the other interested process from entering in the critical section. If the other process is also interested then the process will wait.

Bounded waiting

The interested variable mechanism failed because it was not providing bounded waiting. However, in Peterson solution, A deadlock can never happen because the process which first sets the turn variable will enter in the critical section for sure. Therefore, if a process is preempted after executing line number 4 of the entry section then it will definitely get into the critical section in its next chance.

Portability

This is the complete software solution and therefore it is portable on every hardware.



Synchronization Mechanism without busy waiting

All the solutions we have seen till now were intended to provide mutual exclusion with busy waiting. However, busy waiting is not the optimal allocation of resources because it keeps CPU busy all the time in checking the while loops condition continuously although the process is waiting for the critical section to become available.

All the synchronization mechanism with busy waiting are also suffering from the priority inversion problem that is there is always a possibility of spin lock whenever there is a process with the higher priority has to wait outside the critical section since the mechanism intends to execute the lower priority process in the critical section.

However these problems need a proper solution without busy waiting and priority inversion.

SLEEP AND WAKE

(PRODUCER CONSUMER PROBLEM)

Let's examine the basic model that is sleep and wake. Assume that we have two system calls as **sleep** and **wake**. The process which calls sleep will get blocked while the process which calls will get waked up.

There is a popular example called **producer consumer problem** which is the most popular problem simulating **sleep and wake** mechanism.

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.

In producer consumer problem, let us say there are two processes, one process writes something while the other process reads that. The process which is writing something is called **producer** while the process which is reading is called **consumer**.

In order to read and write, both of them are usinga buffer. The code that simulates the sleep and wake mechanism in terms of providing the solution to producer consumer problem is shown below.

```
1.  #define N 100 //maximum slots in buffer
2.  #define count=0 //items in the buffer
3.  void producer (void)
4.  {
5.      int item;
6.      while(True)
7.      {
8.          item = produce_item(); //producer produces an item
9.          if(count == N) //if the buffer is full then the producer will sleep
10.             Sleep();
11.         insert_item (item); //the item is inserted into buffer
12.         countcount=count+1;
13.         if(count==1) //The producer will wake up the
14.         //consumer if there is at least 1 item in the buffer
15.         wake-up(consumer);
16.     }
17. }
18.
19. void consumer (void)
20. {
21.     int item;
22.     while(True)
23.     {
24.         {
25.             if(count == 0) //The consumer will sleep if the buffer is empty.
26.             sleep();
27.             item = remove_item();
28.             countcount = count - 1;
```

```
29.             if(count == N-1) //if there is at least one slot available in the buffer
30.             //then the consumer will wake up producer
31.             wake-up(producer);
32.             consume_item(item); //the item is read by consumer.
33.         }
34.     }
35. }
```

The producer produces the item and inserts it into the buffer. The value of the global variable count got increased at each insertion. If the buffer is filled completely and no slot is available then the producer will sleep, otherwise it keep inserting.

On the consumer's end, the value of count got decreased by 1 at each consumption. If the buffer is empty at any point of time then the consumer will sleep otherwise, it keeps consuming the items and decreasing the value of count by 1.

The consumer will be waked up by the producer if there is at least 1 item available in the buffer which is to be consumed. The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that.

Well, the problem arises in the case when the consumer got preempted just before it was about to sleep. Now the consumer is neither sleeping nor consuming. Since the producer is not aware of the fact that consumer is not actually sleeping therefore it keep waking the consumer while the consumer is not responding since it is not sleeping.

This leads to the wastage of system calls. When the consumer get scheduled again, it will sleep because it was about to sleep when it was preempted.

The producer keep writing in the buffer and it got filled after some time. The producer will also sleep at that time keeping in the mind that the consumer will wake him up when there is a slot available in the buffer.

The consumer is also sleeping and not aware with the fact that the producer will wake him up.

This is a kind of deadlock where neither producer nor consumer is active and waiting for each other to wake them up. This is a serious problem which needs to be addressed.

Using a flag bit to get rid of this problem

A flag bit can be used in order to get rid of this problem. The producer can set the bit when it calls wake-up on the first time. When the consumer got scheduled, it checks the bit.

The consumer will now get to know that the producer tried to wake him and therefore it will not sleep and get into the ready state to consume whatever produced by the producer.

This solution works for only one pair of producer and consumer, what if there are n producers and n consumers. In that case, there is a need to maintain an integer which can record how many wake-up calls have been made and how many consumers need not sleep. This integer variable is called semaphore. We will discuss more about semaphore later in detail.

Introduction to semaphore

To get rid of the problem of wasting the wake-up signals, Dijkstra proposed an approach which involves storing all the wake-up calls. Dijkstra states that, instead of giving the wake-up calls directly to the consumer, producer can store the wake-up call in a variable. Any of the consumers can read it whenever it needs to do so.

Semaphore is the variables which storesthe entire wake up calls that are being transferred from producer to consumer. It is a variable on which read, modify and update happens automatically in kernel mode.

Semaphore cannot be implemented in the user mode because race condition may always arise when two or more processes try to access the variable simultaneously. It always needs support from the operating system to be implemented.

According to the demand of the situation, Semaphore can be divided into two categories.

1. Counting Semaphore
2. Binary Semaphore or Mutex

We will discuss each one in detail.

Counting Semaphore

There are the scenarios in which more than one processes need to execute in critical section simultaneously. However, counting semaphore can be used when we need to have more than one process in the critical section at the same time.

The programming code of semaphore implementation is shown below which includes the structure of semaphore and the logic using which the entry and the exit can be performed in the critical section.

```
1. struct Semaphore
2. {
3.     int value; // processes that can enter in the critical section simultaneously.
4.     queue type L; // L contains set of processes which get blocked
```

```
5.  }
6.  Down (Semaphore S)
7.  {
8.      SS.value = S.value - 1; //semaphore's value will get decreased when a new
9.      //process enter in the critical section
10.     if (S.value< 0)
11.     {
12.         put_process(PCB) in L; //if the value is negative then
13.         //the process will get into the blocked state.
14.         Sleep();
15.     }
16.     else
17.         return;
18. }
19. up (Semaphore s)
20. {
21.     SS.value = S.value+1; //semaphore value will get increased when
22.     //it makes an exit from the critical section.
23.     if(S.value<=0)
24.     {
25.         select a process from L; //if the value of semaphore is positive
26.         //then wake one of the processes in the blocked queue.
27.         wake-up();
28.     }
29.     }
30. }
```

In this mechanism, the entry and exit in the critical section are performed on the basis of the value of counting semaphore. The value of counting semaphore at any point of time indicates the maximum number of processes that can enter in the critical section at the same time.

A process which wants to enter in the critical section first decrease the semaphore value by 1 and then check whether it gets negative or not. If it gets negative then the process is pushed in the list of blocked processes (i.e. q) otherwise it gets enter in the critical section.

When a process exits from the critical section, it increases the counting semaphore by 1 and then checks whether it is negative or zero. If it is negative then that means that at least one process is waiting in the blocked state hence, to ensure bounded waiting, the first process among the list of blocked processes will wake up and gets enter in the critical section.

The processes in the blocked list will get waked in the order in which they slept. If the value of counting semaphore is negative then it states the number of processes in the blocked state while if it is positive then it states the number of slots available in the critical section.

Problem on Counting Semaphore

The questions are being asked on counting semaphore in GATE. Generally the questions are very simple that contains only subtraction and addition.

1. Wait → Decre → Down → P
2. Signal → Inc → Up → V

The following type questions can be asked in GATE.

A Counting Semaphore was initialized to 12. then 10P (wait) and 4V (Signal) operations were computed on this semaphore. What is the result?

1. S = 12 (initial)
2. 10 p (wait) :
3. SS = S -10 = 12 - 10 = 2
4. then 4 V :
5. SS = S + 4 =2 + 4 = 6

Hence, the final value of counting semaphore is 6.

Binary Semaphore or Mutex

In counting semaphore, Mutual exclusion was not provided because we has the set of processes which required to execute in the critical section simultaneously.

However, Binary Semaphore strictly provides mutual exclusion. Here, instead of having more than 1 slots available in the critical section, we can only have at most 1 process in the critical section. The semaphore can have only two values, 0 or 1.

Let's see the programming implementation of Binary Semaphore.

```
1. StructBsemaphore
2. {
3.     enum Value(0,1); //value is enumerated data type which can only have two values 0 or 1.
4.     Queue type L;
5. }
6. /* L contains all PCBs corresponding to process
7. Blocked while processing down operation unsuccessfully.
```

```
8.  */
9.  Down (Bsemaphore S)
10. {
11.     if (s.value == 1) // if a slot is available in the
12.     //critical section then let the process enter in the queue.
13.     {
14.         S.value = 0; // initialize the value to 0 so that no other process can read it as 1.
15.     }
16.     else
17.     {
18.         put the process (PCB) in S.L; //if no slot is available
19.         //then let the process wait in the blocked queue.
20.         sleep();
21.     }
22. }
23. Up (Bsemaphore S)
24. {
25.     if (S.L is empty) //an empty blocked processes list implies that no process
26.     //has ever tried to get enter in the critical section.
27.     {
28.         S.Value =1;
29.     }
30.     else
31.     {
32.         Select a process from S.L;
33.         Wakeup(); // if it is not empty then wake the first process of the blocked queue.
34.     }
35. }
```