RACE CONDITION :- When two or more programs are trying to read/write some shared buffer or shared memory, the outcome of the program varies with the sequence of execution of program such conditions are known as RACE CONDITION.

To overcome the problem of Race condition is Mutual Exclusion :- Access to shared variable / buffer should be mutually exclusive i.e. only one process will be able to read/write the information from shared buffer / variable.

Critical Section :- Each process has a segment of code, called critical section in which the process may be changing common variables, updating a table, writing a file and soon. The important feature of the 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.

* Consider a simple system of only two process that share a common resource with in a critical section

Critical Section Problem :- the critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

A solution to the critical section problem must satisfy the following three requirements :

(i) **Mutual Exclusion :-** If process P_i is executing in its critical section then no other processes can be executing in their critical section.

(ii) **Progress :-** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next.

(iii) **Bounded waiting :-** There exists a bound, or limit 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 & before that request is granted.

The First Algorithm :-

```
type
    who = (proc1 , proc2);
var
    turn : who;
process p1;
 begin
    while true do
      begin
      while turn = proc2 do {keep testing};
        critical section;
        turn := proc2;
        other - p1 - processing
      end {while}
    end; {P1}
process p2;
 begin
    while True do
    begin
      while turn = proc1 do {keep testing};
        critical section;
        turn := proc1;
        other - p2 - processing
      end {while}
    end; {p2}
```

```
{parent process}
  begin {mutex}
    turn := ....;
    initiate p1, p2
  end {mutex}
```

Mutual exclusion Algo

A Global variable Turn is used to control access to an unspecified shared resource, which may be a data structure, a piece of code, or a physical device.

The first important question is whether this solution ensures mutual exclusion between the two processes. Informally the control variable TURN has only one value at any time. Consequently only one process at a time is allowed access. If the other process wishes to access the shared resource at that time, it is kept waiting in the busy loop until the current occupant completes its critical section and updates TURN.

Problem :-

1) Lets startup with TURN set to Proc1, the two processes can only execute in alternate sequence!

P1, P2, P1, P2, P1, P2 .....

Turn taking actually forces the two processes to work at the collective speed of the slower one.

2) Let us assume that process P1 crashes somewhere in its non critical code, other-P1-Processing. After process P2 to enter its critical section, process P2 sets turn back to Proc 1, as usual. But P1 is no longer active, so it can never give the turn back to P2.

The Second Algorithm :- To avoid strict turn-taking and over-dependence on other processes. It introduces two flags : P1using and P2using. Each process updates its own flag to indicate its current activity or intentions regarding use of the shared resource, and reads the flag of the other process only when necessary for synchronization.

This Algo is much safer than the Algo1 because each process updates only its own flag. As a result, a process crashing outside its critical section, after having updated the related flag accordingly

does not pose a problem.

```
var
    p1using, p2using : boolean;
process P1;
begin
    while true do
        begin
            while p2using do {keep testing};
                p1using := true;
                critical_section;
                p1using := false;
                other-p1-processing
            end {while}
        end {P1}
process P2;
  begin
      while true do
        begin
          while p1using do {keep testing};
              p2using := true;
              critical-section;
              p2using := false;
              other-p2-processing
          end {while}
        end: {P2}
{parent process}
  begin {mutex}
      p1using := false
```

```
p2using := false;
Initiate p1, p2
end {mutex}
```

Mutual enclusion : Algorithm 2

Consider a scenario where both processes are outside of their critical sections, and both flags indicate that the resource is free. Suppose that both processes decide to enter their respective critical sections at about the same time. Process p1 inspects p2using, find it is false, sets the flag p1using and enters the critical section. (Vice versa) with two processes in the critical section at the same time, fails to satisfy the mutual exclusion requirement.

The Third Algorithm :-

```
var
    p1using, p2using : boolean;
process p1;
  begin
    while true do
      begin
        p1using := true;
```

```
while p2using do {keeptesting};
        critical-section;
        p1using := false;
        other-p1-processing
    end {while}
end; {p1}

Process p2;
  begin
      while true do
          begin
            p2using := true;
            while p1using do {keeptesting};
              critical-section;
              p2using := false
              other-p2-processing
          end {while}
      end; {p2}
{parent process}
  begin {mutex3}
        p1using := false;
        p2using := false;
        initiate p1, p2
    end {mutex 3}
```

problem :- consider a scenario where process P1 wishes to enter the critical section and sets P1using to indicate the fact. If process P2 wishes to enter the critical section at the same time and preempts process P1 just before P1 sets P2using, process P2 may set P2using and start looping awaiting P1using to become false. When control is eventually returned to process P1, it finds P2using set and starts looping awaiting P2using to become false.

ALGO 4

```
Var
    p1using , p2using : boolean;

process p1;
begin
    while true do
    begin
        p1using := true
        while p2using do
        begin
            p1using = false;
            {give p2 a chance}
            p1using := true
        end ; {while}
```

```
        critical_section;
        p1using := false;
        other-p1-processing
    end; {p1}

process p2;
 begin
    while true do
    begin
        p2using := true
        while p1using do
        begin
            p2using := false;
            {give p1 a chance}
            p2using := true
        end; {while}
        critical_section
        p2using := false
        other-p2-processing
    end {while}
  end; {p2}

{parent process}
 begin
    p1using := false
    p2using := false
    initiate p1, p2
  end
```

Dekker's solution to mutual exclusion

```
type
    who = (proc1, proc2)
var
    turn: who;
    p1using, p2using: boolean

process p1
begin while true do
    begin
        p1using := true
        while p2using do
            if turn = proc2
            then
                begin
                    p1using = false
                    while turn=proc2 do
                                {keeps testing}
                    p1using = true
                end {if}
        critical-section
        turn := proc2
        p1using := false
        other-p1-processing
    end {while}
end {p1}

process p2
begin
    while true do
    begin
        p2using := true
        while p1using do
```

```
        if turn = proc1
        then
            begin
                p2using := false
                while turn = proc1 do {keeptesting}
                p2using := true
            end {if}
        critical-section
        turn := proc1
        p2using := false
        other-p2-processing
    end {while}
end {p2}

{parent process}
begin
    p1using := false;
    p2using := false;
    turn := proc1
    initiate p1, p2
end
```

x — x — x — x — x — x —

Lamport's Bakery Algorithm :-

```
var {for all processes}
    choosing: array [1..n] of integer;
    number: array [1..n] of integer;
```

```
{individual processes}
Process i; {process on processor i}
var
    j : integer;
    begin
        while true do
        begin
            choosing[i] := 1
            number[i] := 1 + maximum(number[1],-
                                ..... number[n]);
            choosing[i] := 0;
        for j := 1 to n do
        begin
            while choosing[j] <> 0 do {keep testing};
            while number[j] <> 0 and
            (number[j], j < number[i], i) do
                                {keep testing};
        end; {for}
        critical_section;
        number[i] := 0;
        other-i-processing
    end {while}
end; {process i}
```

Semaphore :- Dijkstra's proposal of a mechanism for mutual exclusion among an arbitrary number of processes, called a semaphore. A semaphore mechanism basically consists of the two primitive operations SIGNAL and WAIT which operate on a special type of semaphore variable s. The semaphore variable can assume integer values and except possibly for initialization may be accessed and manipulated only by means of the signal and wait operations.

wait(s): decrements the value of its argument semaphore, s, as soon as it would become non negative.

signal(s): increments the value of its argument semaphore, s, as an indivisible operation.

A semaphore whose variable is allowed to take on only the values of 0 (busy) and 1 (free) is called a binary semaphore.

The general semaphore make take any integer value. The logic of WAIT and SIGNAL operations is applicable to both binary and general semaphores.

(S<=0)

wait(S): while not (S>0) do keep testing

S := S-1

signal(S): S := S+1

Time	Process status/ Activity			Mutex 1 = Free 0 = Busy	Processes in critical section attempting to enter
	P1	P2	P3		
M1	-	-	-	1	-;-
M2	wait(mutex)	wait(mutex)	wait(mutex)	0	-; P1, P2, P3
M3	critical-section	waiting	waiting	0	P1; P2, P3
M4	signal(mutex)	waiting	waiting	1	-; P2, P3
M5	other-P1-process	critical-section	waiting	0	P2; P3
M6	wait(mutex)	critical-section	waiting	0	P2; P3, P1
M7	waiting	signal(mutex)	waiting	1	-; P3, P1
M8	critical-section	other-P2-process	waiting	0	P1; P3

A scenario of execution of Semaphore

Mutual exclusion with Semaphores :-

```
Var mutex : semaphore ; {binary}

Process P1 ;
  begin
    while true do
      begin
        wait (mutex);
        critical-section;
        signal (mutex);
        other-p1-processing
      end {while}
  end ; {p1}

Process P2 ;
  begin
    while true do
      begin
        wait (mutex);
        critical-section;
        signal (mutex);
        other-p2-processing
      end {while}
  end ; {p2}

Process P3 ;
  begin
    while true do
      begin
        wait (mutex);
        critical-section;
        signal (mutex);
        other-p3-processing
      end {while}
  end ; {p3}
```

```
{Parent Process}
  begin {smutex}
    mutex := 1 ; {free}
    initiate p1, p2, p3
  end {smutex}
```

X — X — X — X — X — X — X

Hardware Support for Mutual Exclusion

Semaphores solve the mutual-exclusion problem in a simple and natural way, from the user's point of view, and also boast a no. of nice properties.

In terms of pessimistic updating a shared global variable or a semaphore variable, a typical pessimistic solution may act as follows:

(i) Block everything that could possibly interfere, so that nothing will interfere.

(ii) Update the global variable

(iii) UnBlock the part of the system blocked in step 1.

In contrast, optimistic approaches are based on the assumption that no or few conflicts are likely to be experienced by any particular user of the shared resource. For example, a typical optimistic solution may be structured as follows:

(i) Read the value of the global variable and prepare the tentative local update based on that value.

(ii) Compare the current value of the global variable to the value used to prepare the tentative update. If the value of the global variable is unchanged, apply the tentative local update to the global variable. otherwise, discard the tentative update and repeat from step 1.

Critical Section for H/w :- Two ways

→ By Disabling the interrupt

→ By using the h/w instruction.

Disable / Enable Interrupts :- The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. The basic idea is quite simple, and it follows the principle that the shared resource must be exclusively obtained by the process wishing to enter its critical section and subsequently released for the use of others. This may be accomplished by the following sequence

```
DI                  ; disable interrupts
critical-section    ; use guarded resource
EI                  ; enable interrupts
```

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling and enabling of interrupts for multiprocessor systems with shared memory, because the DI and EI instructions affect only the issuing processor and cannot prevent interference from processes running on other processor.

By using the h/w instructions :-

(i) Test-and-set Instruction :- The test-and-set instruction is intended for direct hardware support of mutual exclusion. The basic idea is to set the global control variable to FREE when the guarded shared resource is available. Each process wishing to access the resource must obtain a permit to do so by executing the TS instruction with the related control variable as the operand. In principle, TS takes one operand, the address of the control variable or a register that may act as a semaphore, and works as follows :

TS operand ; test and set operand

(i) Compare the value of the operand to BUSY, and set the condition codes to reflect the outcome.

(ii) set the operand to BUSY.

These steps are performed as a single, indivisible operation. The WAIT operation on the semaphore variable S may be implemented as the following procedure when TS is available in the instruction set of the supporting hardware.

```
WAIT :
        TS    S        ; request enclusive
                         aucss
        BNF   WAIT     ; repeat until granted
        RETURN         ; proceed to critical
                         section
```

The TS, with proper implementation, may also be used in multiprocessor systems. To understand the need for the specific h/w support for TS, consider a multiprocessor systems with a certain amount of memory shared by and accessible to several processors.

```
WAIT:
        MOV   AL,1      ; set local flag to BUSY
LOCK    XCHG  AL,S      ; exchange global & local flags
        CMP   AL,0      ; was global = FREE?
        JNZ   WAIT      ; if not, keep trying until FREE
        RETURN          ; return control, resource is FREE
```

A multiprocessor WAIT for 8086 family.

x - x - x - x - x - x - x - x

(ii) Compare-and-swap Instruction :- Compare-and-swap (CS) is a hardware instruction introduced by IBM in their 370 series. The CS instruction follows an optimistic approach to the mutual-exclusion problem. The CS instruction is very convenient and efficient for simple updates of shared variables.

CS takes three operands :- (i)

=> a register that contains the value of the global variable on which the tentative update is based (OLDREG)

=> a register that contains the tentative update (NEWREG)

=> the address of the global variable in question (GLOBVAR)

CS consists of the following sequence of steps, which are executed as a single, indivisible operation.

```
CS  OLDREG, NEWREG, GLOBVAR

    COMPARE  OLDREG, GLOBVAR
    SET CONDITION CODES
    IF (OLDREG = GLOBVAR)
                THEN  GLOBVAR ← NEWREG
                ELSE  OLDREG ← GLOBREG
```

X - X - X - X - X - X - X - X - X - X

Classical Problems In Concurrent Programming:-

The Producers / consumers Problem :- In general, the producers/consumers problem may be stated as follows

⇒ Given a set of co-operating processes, some of which "produce" data items to be "consumed" by others (consumers).

Devise a synchronization protocol that allows both producers and consumers to operate at their respective service rates in such a way that produced items are consumed in the exact order in which they are produced (FIFO)

Producers and Consumer with an Unbounded buffer :-

In the first attempt to solve the producers/consumers problem, we assume that a buffer of unbounded capacity is set aside to smooth the speed differences between producers and consumers. Given the unbounded buffer, producers may run at any time without restrictions. The buffer itself may be implemented as an array, a linked list, or any other collection of those data items.

```
var produced : semaphore; {general}
Process producer;
begin while true do
      begin
            produce;
            place-in-buffer;
            signal (produced);
            other-producer-processing
      end {while}
end; {producer}

Process consumer
begin while true do
      begin
            wait (produced);
            take-from-buffer;
```

```
        consume;
        other_consumer_processing
    end {while}
end; {consumer}

{parent process}
begin {producer-consumer}
    {produced := 0;}
    initiate producer, consumer
end {producer-consumer}
```

Producers and consumers with a bounded buffer

The unbounded buffer assumption simplifies analysis of the producers/consumers problem by allowing virtually unrestricted execution of producers. However, this assumption is unrealistic, and proposed solutions based on it may not be directly implementable in real computer systems, which have finite memory capacity. In this section we present solutions to the producers/consumers problem assuming a finite capacity of the shared buffer.

Let icount be the number of items produced but not yet consumed, that is,

$$icount = produced - consumed$$

Given a finite capacity of the buffer the following must hold:

$$0 \leq icount \leq capacity$$

where capacity is the capacity of the global buffer. Since producers may run only when there are some empty slots in the buffer,

$$mayproduce : icount < capacity$$

consumers, on the other hand, can execute only when there is at least one item produced but not yet consumed, that is

$$mayconsume : icount > 0$$

The two conditions, may produce and may consume can be control execution of producer and consumer processes based on the current state of the buffer.

```
const
    capacity = .....;
type
    item = ....;
var
    buffer : array [1..capacity] of item;
```

```
mayproduce, mayconsume : semaphore ; {prod
pmutex, cmutex : semaphore ; {buffer
in, out : (1.. capacity) ;

Process producer X;
  var pitem : item;
  begin
      while True do
      begin
          wait (may produce);
          pitem := produce;
          wait (pmutex);
          buffer[in] := pitem;
          in := (in mod capacity) + 1;
          signal (pmutex);
          signal (mayconsume);
          other - X - processing
      end {while}
  end; {producer X}

Process consumer Z;
  var citem : item;
  begin while True do
      begin
          wait (mayconsume);
          wait (cmutex);
          citem := buffer[out];
```

```
        out := (out mod capacity) + 1;
        signal (cmutex);
        signal (mayproduce);
        consume (citem);
        other-2-processing
    end {while}
end {consumer2}

{Parent Process}
begin {BB-producers-consumers}
    in := 1;
    out := 1;
    signal (pmutex);
    signal (cmutex);
    {mayconsume := 0;}
for i := 1 to capacity do signal (mayproduce);
initiate producers, consumers
end {BB-producers-consumers}
```

Readers and writers :- Readers and writers is another classical problem in concurrent programming with numerous applications. The processes are categorized depending on their usage of the resource, as either readers or writers. A reader never modifies the shared data structure, whereas a writer may read it and write into it. A number of readers may use the shared data structure currently, because no matter how they are interleaved, they cannot possibly compromise its consistency. The problem may be somewhat more precisely stated as follows.

- Given a universe of readers that read a common data structure and a universe of writers that modify the same common data structure.
- Devise a synchronization protocol among the readers and writers that ensures consistency of common data while maintaining as high a degree of concurrency as possible.

```
Var
    readercount : integer
    mutex, write : semphore ; {binary}
```

```
Process reader x;
begin
  while true do
  begin
    {obtain permission to enter}
    wait (mutex);
    readercount := readercount + 1;
    if readercount = 1 then wait(write);
    signal (mutex);
    ---
    {reads}
    ---
    wait (mutex);
    readercount := readercount - 1;
    if readercount = 0 then signal(write);
    signal (mutex);
    other-X-processing
  end {while}
end;
Process writer2;
begin
  while true do
  begin
    wait (write);
    ---
    {writes};
    ---
    signal (write);
```

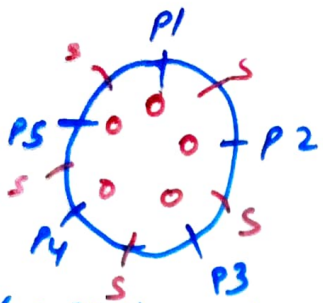
```
    other-2-processing
  end {while}
end ; {writer 2}

{Parent process}
begin
      readercount := 0
      signal (mutex);
      signal (wrte);
      initiate readers, writers
  end
```

Problem in Reader/writer :- Reader & writer can't enter system simultaneously. At any given time any of the reader can enter to the system.

- Basically it includes

* A new reader should not start if there is a writer waiting (Prevents starvation of writers).
* All readers waiting at the end of a write should have priority over the next writer (Prevents starvation of readers).

DINING PHILOSPHER PROBLEM :-



These Philosphers perform only 2 operations that are think and eat. spoon is present b/w philosphers and each philosphar whenever feels hungry pick 2 spoons from his left & right side. He can eat only when he has 2 spoons otherwise not. In this, at one time only 2 philosopher can eat. So, to overcome from this problem are follow the below algo :-

Process = Philosopher
Sources = spoon

```
#define N 5                    // No. of philosopher are 5.
#define left (i-1)%N           // i is any given philosopher
#define right (i+1)%N
#define thinking 0
#define Hungry 1               } can ignore
#define Eating 2               }
int state[N];                  // Keep track of each process or philosopher
semaphore mutex 1;
semaphore s[N];                // Allocating semaphore to each & every philosopheres.
void philosopher (int i)
{
    while (true)
    {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

```
void take_forks (int i)
  {
     down (mutex);          // to pick up the spoons
     state [i] = hungry;    // from left & right so
     test (i);              // lock the shared varia-
     up (mutex);            // bles that are spoons.
     down s[i];
  }

Void put_forks (int i)
  {
     down (mutex);
     state[i] = thinking;
     Test (left);
     Test (right);
     up (mutex);
  }
Void Test (int i)
{
  if state[i] = hungry && state [left] != eating
     && state [right] != eating
     {
        state[i] = eating
        up(s[i]);
     }
}
```