BP

Unit - 2 (Concurrent Processes)

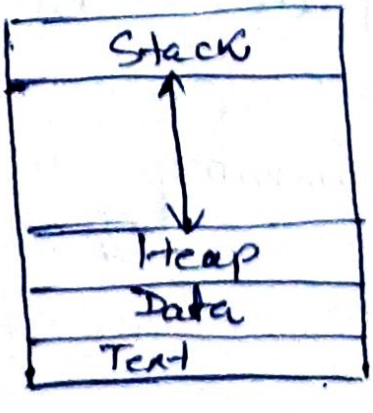
Process → "A Process is basically a Program in execution."

→ The execution of Process must Progress in a sequential fashion.

→ "A Process is defined as an entity which represents the basic unit of work to be implemented in the system."

⇒* To understand it in simple terms, we write our computer Programs in a text file and when we execute this Program, it becomes a Process which Performs all the tasks mentioned in Program.

*→ When a Program is loaded into the memory and it becomes a Process, it can be divided into four sections — stack, heap, text and data.



Stack- The Process Stack Contains the temporary data such as method/function Parameters, return address, and local variables.

Heap- This is a dynamically allocated memory to a Process during its runtime.

Text - This includes the current activity represented by the value of Program Counter and the Content of Processor registers.

Data - This section Contains the global and Static variables

Concurrent Process — Two Processes are Concurrent when they are executed in a way that their execution intervals overlap.

There is Concurrency

There is No Concurrency

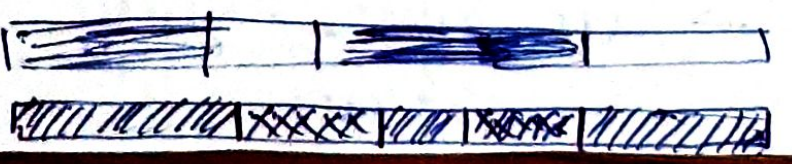
Kinds of Concurrency :-

① Apparent Concurrency — More Processes than Processors

- Processes are multiplexed in time
- Pseudoparallelism.

1 CPU

2 CPU

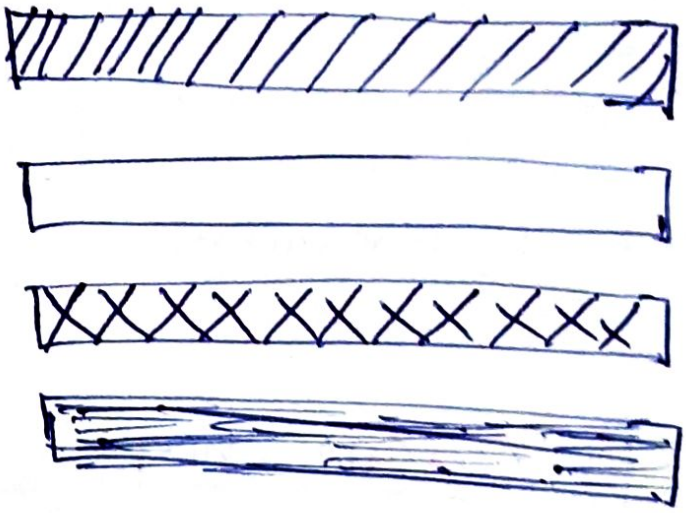


② Real Concurrency:- Each Process runs in a Processor

— A Parallel execution happens

— Also known as real Parallelism

4 CPUs



Concurrent Programming Models

① Multiprogramming with Single Proc

— OS in charge of dividing time among Processes

② Multiprocessor → Combines real Parall -ism and Pseudoparallelism

→ Usually more Processes than CPU

③ Distributed System → Computers
— Several Connected through a Network.

Advantages of Concurrent Execution

① Eases Programming →
- Several tasks can be structured in separate Processes
web Server: — A Process in charge of handling each request

② Accelerates Computation Execution →
- Division of Computations in Processes run in Parallel
ex— Simulations, Image Processing, financial Computations.

③ Improves Application Interactivity
- Ability to separate Processing tasks from user attention tasks.
ex— Print and edit.

④ Improve CPU utilization →

– I/O phases from one applicatio
are used for Processing from
other applications

Kinds of Concurrent Processes →

① Independent –

– Processes run concurrently
but without interaction

→ No need for Communicatio
→ No need for Synchronizatio

Ex– Two shells for two user
in two different terminals.

② Cooperating →

– Processes run concurrently with
some interaction

→ May Communication each other
→ May be Synchronized

Ex– Transaction Server organized
in receiver Process and request handl
Processes

Interactions among Processes :-

(1) Access to Shared Resources:

- Processes sharing a resource
- Processes competing for a resource

Ex- Request server with different Processes writing to a log file.

(2) Communication :-

- Processes exchanging information

Ex- Requests receiver must Pass information to a request handling process.

(3) Synchronization →

- A Process must wait for an Event in another Process.

Ex- A user interface Process must wait until all Computation Process have finished

So, we can say that

"

The Concurrent Processes executing in the O.S may be either indep Processes or cooperating Processes.

A Process is independent if it can't affect or be affected by the other Processes executing in the system. Clearly, any Process that does no share any data (temporary or Persi with any other Process is indepe - t. On the other hand, a Process i cooperating if it can affect or be affected by the other Process executing in the system. Clearly, any Processes that shares data with other Processes is a cooperating Process.

We want Process Cooperation for several reasons:

(1) Information Sharing: Since several users may be interested in the same Piece of information (for instance, a shar file), we must Provide an environm to allow concurrent access to these type of resou

Computation Speedup → If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPU or I/O channels)

Modularity → We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

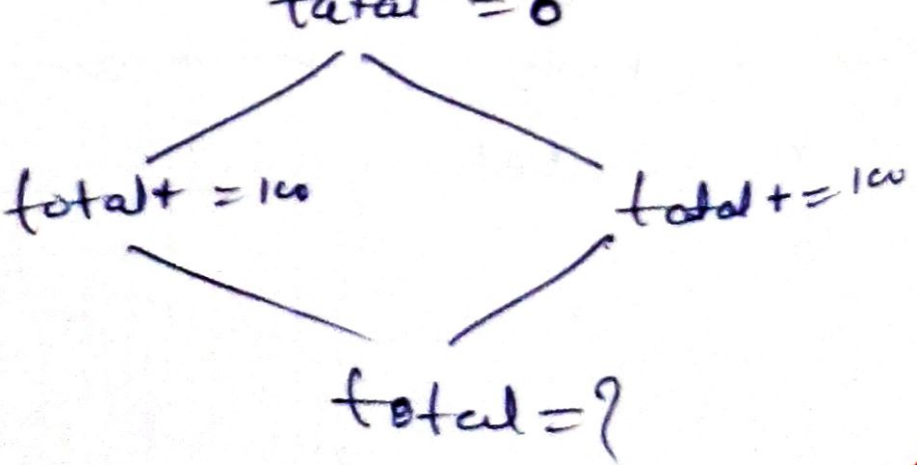
Convenience → Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Process Synchronization → It means:

Coordinating the execution of Processes such that no two Processes access the same shared resource and data and hence its a technique to overcome the Problem that can result in data inconsistency.

What is race condition? → "A situation where several Processes access and manipulate the same data concurrently and outcome of the execution depends on the particular order in which the access takes Place, is called a race condition"

ex →



So, functioning of a process and its results must be independent from its relative execution with respect to other processes.

— Necessary to guarantee that order of execution does not affect result.

Solution → To achieve that a set of instructions can be executed atomically

Mutual Exclusion

Critical Section Problem → Consider a system consisting of n processes ($P_0, P_1, \ldots P_{n-1}$) each has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on.

→ The important feature of the sys
is that when the Process is execut
in its critical section, no other
Process is to be allowed to execut
in its critical section.

→ The execution of critical section by the Proces
is a mutually exclusive.

→ The critical section Problem is
to design a Protocol that th
Process can use to cooperate
each Process must request Permissi
to enter its critical section.

→ the section of code implementing
this request is the entry sectio

→ The critical section is followed by
exit section

→ The remaining code is the remain
section

Example:

```
while (1)
{
    Entry Section;
        critical Section;
    Exit Section;
        Remainder Section;
}
```

A solution to the critical section problem must satisfy the following conditions

1- Mutual Exclusions:- If Process P_I is executing in its critical section then no any other Process can be executing in their critical section

2- Progress:- If no Process is executing in its critical section and some Process wish to enter their critical section then only those

Process that are not executing in their remainder section can ente its critical section next

3.- Bounded Waiting → There exists a bound on the number of tim that other Processes are allowe to enter their critical Section after a Process has made a requ

---

Semaphores: —

→ For the solution to the critica section Problem one synchroniz tool is used which is known as semaphores.

→ A Semaphore 'S' is an integ variable which is accessed thro two standard operations such as wait and Signal.

→ these operations were originally termed 'P' (for wait means to test and 'V' (for [redacted] Signal means to incr

The classical definition of wait is

```
wait(S)
{
    while(S<=0)
    {
        ; // no-op
    }
    S--;
}
```

The classical definition of the signal is

```
Signal(S)
{
    S++;
}
```

repeat
wait(mutex) // Entry Section
critical Section
signal(mutex) // Exit Section
remainder Section
until false.

In case of wait the testing integer value (S<=0) and the decrement (S--) is executed without interruption.

* The semaphore construct described previously is known as counting semaphore, since its integer value can range over an unrestricted domain.

Binary Semaphore: A binary semaphore is a semaphore with an integer value which can range between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture.

Let S be a counting semaphore. To implement it in terms of a binary semaphore we need the following structure of data

else
Signal(S1);

P() , Down , wait
V() , up , Signal

type semaphore = record

value : integer

L : List of Process

end

The semaphore operations can be defined as

wait(S) : S.value = S.value − 1

if S.value < 0

then begin

add this Process to S.L

Block/Sleep()

End

Signal(S): S.value = S.value + 1

if S.value ≤ 0

then begin

remove a Process P from

wakeup(P)

end.

To get into ready Queue wake up() means become active based on FIFO

Now, Definition of Binary Semaphore

Type Binary_semaphore = record

value (0,1)

Queue: List of Process

end

Var S: binary_semaphore.

Wait B(S)

if S.value == 1

then S.value = 0

else begin

Place the Process to S Queue

Block this Process

end

NB- Value=1 (... is ...
Value= there the ... means (rear of occupi Queue

Proc
and
re
w
code

Signal B(S)

If S. Queue is empty

then S. value = 1

else

remove a Process from S. Queue

Place Process P on ready list

End.

Process Synchronization using Producer and Consumer Problems

repeat

- - -

Produce an item in next P

(next P: item produced by Producer)

- - -

while Counter = n do no-operation

buffer[in] = next P

in = (in + 1) mod n

(in is the rear of Queue)

Counter = Counter + 1

until False

Code for Producer

register 1 = Counter

register 1 = register 1 + 1

Counter = register

repeat

while Counter = 0 do no-operation

next C = buffer[out]

out = (out + 1) mod n

Counter = Counter − 1

— — —

Consume the item in next C

until false

Code for Consumer –

register 2 = Counter

register 2 = register 2 − 1

Counter = register 2

To understand

let Counter = 5

T_0 : Producer execute $register_1$ = Counter

T_1 : " " " = $register_1$ + 1

T_2 : Consumer execute $register_2$ = Counter

T_3 : Consumer executes register$_2$ = register$_2$ − 1

{register$_2$ = 4}

T_4 : Producer execute Counter = register$_1$

{Counter = 6}

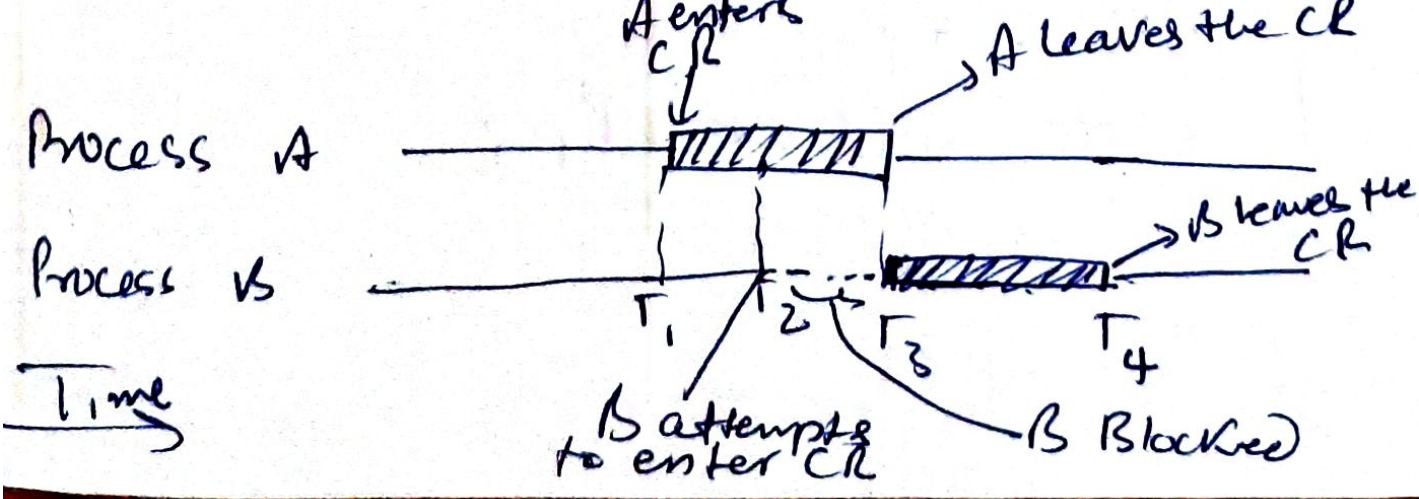
T_5 : Consumer executes Counter = register$_2$

{Counter = 4}

Notice that we have arrived at the incorrect state "Counter = 4", indicating that four buffers are full, when infact, five buffers are full.

** We arrive at this incorrect state because we allowed both Process to manipulate the variable counter concurrently.

Mutual Exclusion using Critical Region



// Producer.

```
int Count = 0
Void Producer (void)
{
    int itemp;
    while (true)
    {
        Produce_item(itemp)
        while (Count == n); // Buffer
        Buffer[in] = itemp
        in = (in+1) mod n;
        Count = Count + 1; // Count is shared by both P and
    }
}
```

Micro instructions generated by

```
{ Load Rp, m[Count]   (memory to Rp)
  INCR Rp
  Store m[Count], Rp;
```

// Consumer

```
Void Consumer (Void)
{
    int itemc;
    while (true)
    {
        while (count == 0);   // Buffer empty
        itemc = Buffer(out)
        out = (out + 1) mod n;
        count = count - 1;
        Process_item(itemc);
    }
}
```

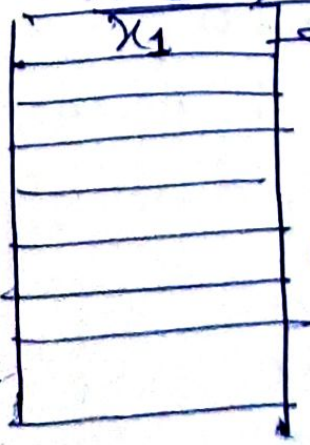
memory

load R_c, m[count]
DECR R_c
Store m[count], R_e

Case-2 ① executed I_1 I_2/Preempt, I_3
② I_1 I_2/Preempt executed

n=8, Buffer[0 ... n-1]

(no bubble)
R_e - I - X_1
(0+1) mod 8
= 1
{(0+1) mod 8
1 mod 8
= 1



out [0] / 1

Consumed Successful
Case 1

in [0]

Count [1] / 0

// is Shared by Both Producer and Consumer

Peterson's Solution (1981)

→ A Classic Software-based Solution to the critical-Section Problem

→ May not work on modern Comp Architecture.

→ However, it Provides a good algorithmic description of Solving the critical Section Problem and illustrates Som of the Complexities involved in designing Software that addresses the requirem of mutual exclusion, Progress and bound waiting requirements.

⇒ Peterson's Solution is restricted to two Processes that alternate exec between their critical Section and rema Section.

Let's call the Processes P_i and P_j

⇒ Peterson's Solution requires two data ite to be shared between the two Processes

⇒ int turn
indicates whose turn it is to enter its critical Section

boolean flag[2]
used to indicate if a Process is ready to enter its critical Section.

Structure of Process P_i in Peterson's Solution

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

Structure of Process P_j in Peterson's Solution

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

DeKKer's Algorithm

(Dutch Mathematicians)

DeKKer's algorithm is the first known algorithm that solves the mutual exclusion problems in concurrent programming. It is credited to Th. J. DeKKer

The DeKKer's algo assumes that there are two processes P_{first} and P_{second}

```
{   int first=0;
    int second= 1 - first;
```

Two process share two variables:

```
boolean flag[2];
int turn;
```

Here initially, flag[first] = flag[second]; and no initial value for turn is needed but variable turn can either 0 or 1.

P_{first}

```
// Entry section begins
flag[first] = true;
while(flag[second])
{
    if (turn == second)
    {   flag[first]=false;
        while(turn = second)
        {
        // do no op as Psecond
        } is executing in its CS;
    }   flag[first]=true;
}
```

P_{second}

```
// Entry section begins
flag[second]=1
while(flag[first])
{
    if (turn==first)
    {  flag[second] =false;
       while (turn==fi
       {
       // do no-op as Pfi
       } is executing in
    }  flag[second] = True
}
```

```
// Entry Section ends
{Critical Section}
// Exit. Section begins
turn = Second;
Flag [first] = false;
// Exit Section Ends
{remainder Section}
```

```
// Entry Section ends
{Critical Section}
// Exit Section begins
turn = first;
flag [second] = false;
// Exit Section ends
{ remainder Section}
}
```

int itemp, itemc

Solution of P-C Problem

Counting Semaphore → full = 0 = No of filled slots

→ Empty = N = No of empty slots

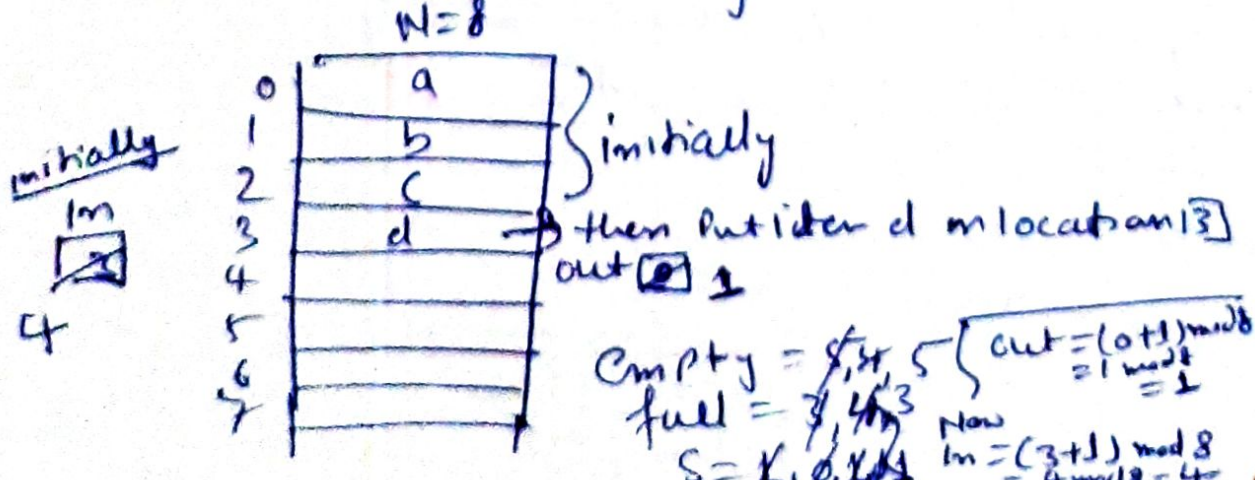
Binary Semaphore S = 1

// Producer

```
Produce_item(itemp)
1- down(empty);
2- down(S);
   Buffer[in] = itemp;
   in = (in+1) mod n;
3- UP(S);
4- UP(full);
```

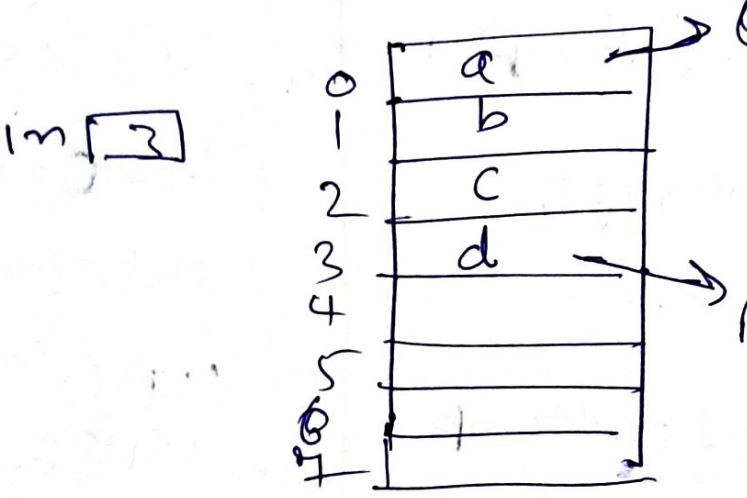
// Consumer

```
1- down(full);
2- down(S);
   itemc = Buffer[out];
   out = (out+1) mod n
3- UP(S)
4- UP(Empty)
```



So, if 1 Producer and one Consumer is Comming then there is no Problem.

Again lets see the situation when Concurrent Process ~~ex~~ situation ...—



Empty = ~~4~~ 4

full = ~~3~~ 1,2,3

S = ~~1~~ 0, ~~1~~, 0, 1

Producer Preempt after executing instruction 1 i.e down(Empty) (Preempt)

~~But~~ this Point Consu arrives

⇒ Now consumer is at line out i.e Processing out instru Now Preempted

⇒ at this Point agai Consumer comes, 1st Ins already executed (1) of Producer. it can' perform Down(S) it will Blocked

Now after Performing all (5) operations of Consumer Code, the control arrives at Produce down(S) → successful (i.e - 1 to 0)

in = d → item Produced

in = 4 - updated

Critical Section Solution using Lock

d {
acquire lock
CS
release lock
}

Entry code:
1. while (Lock==1);
2. Lock = 1
3. critical Section
4. Lock = 0 — Exit code

* Execute in user mode
* Multi process Solution
* No Mutual Exclusion Solution Gurantee

{Lock = 0 → Vacant} CS (mean)
1 → full

Case-1 —
P_1, P_2

- instruction 1 2 3 executed and P_1 is in critical Section
- Now after executing critical Section, P_1 will change the value of lock to '0'

Since the value of Lock = 1 at this time because P_1 has changed the value of Lock from 0 to 1 to enter in critical Section, So P_2 can't enter

= Now after P_1 has been executed in its critical Section P_2 can enter into in its C.S because at this time Lock = 0.

Critical Section Solution using Test and Set Instructions →

Case-2 To Prove that there is no guarantee of mutual Exclusion.

L=0 initially

P_1	P_2
- L1 executed ⇒ as soon as P_1 Completed the L1 execution ie before L2 execution it is get Preempted	- at the same time when P_1 get Preempted, P_2 enters and get same L=0 because it is not updated yet ie L1 executed L2 // and P_2 is in C.S
Now again P_1 comes back at this time to execute ⊕ L2 and executes it successfully and enters into its C.S (L3 executed)	

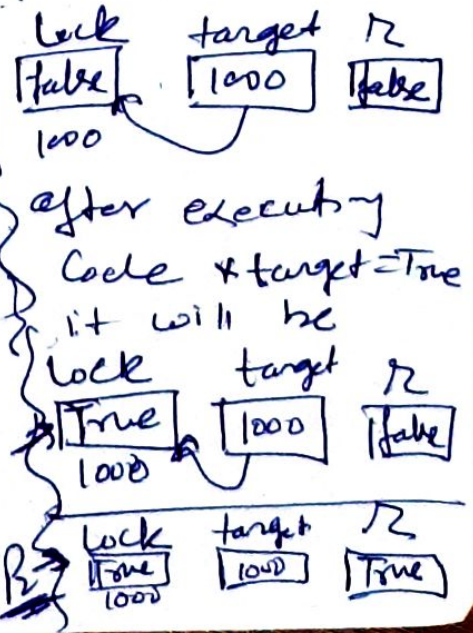
NB— It means there is no guarantee of mutual Exclusion as explained in Case-No-2

So, Now the code is changed in Solution using Test_and_Set instruction:

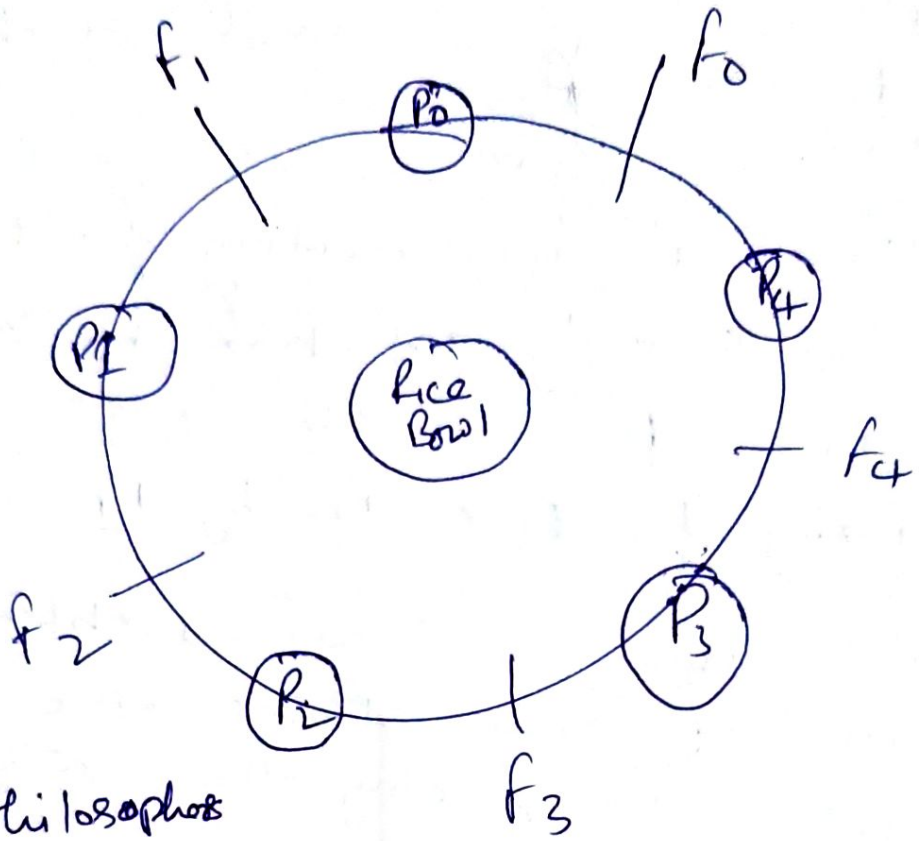
```
while (test_and_Set(&lock));
  CS
lock = false
```

Definition of function

```
boolean test_and_Set(boolean *target)
{
    boolean rv = *target;
    *target = True
    return rv
}
```

P_1	P_2	L=0 initially i.e
- First P_1 arrives the definition of function will return false and P_1 will get into it C.S	Now P_2 arrives after P_1 is executing its C.S. Now at this time the Lock value is True. So in rv it will be updated to True. Here a true value will be returned and while(True); will be executed and P_2 will not enter in C.S.	

Dining Philosopher Problem



'5' Philosophers

'5' forks/chopsticks

Case-1 - P_0 (let's assume)

will P_1 & f_0

N = No of forks

$(0+1) \bmod 5 = 1$

then

Put fork = f_0

Put fork = f_1

Case-II - P_1 (let's assume)

P_1 & f_1

P_1 & $(1+1) \bmod 5$

i.e f_2

Put f_1

Put f_2

```
Void Philosopher(Void)
{
  while(True)
  {
   Thinking();
   take_fork(i); ← left fork
  take_fork((i+1)%N); ← right fork
   Eat();
   Putfork(i);
  Putfork((i+1)%N);
  }
}
```

So, if the philosophers arriving serially then there is no problem with this code.

Now

Case-3 — P_0 (arrives) | P_1

(P_1 arrives at same time after P_0 Picked left fork)

(P_0 (left) → f_0 (Pick))

(P_1 left → f_1 Picked)

Now, Here P_0 is waiting for fork f_1 and P_0 will only get fork f_1 only after P_1 will f_1 and f_2 and after eating and after putting for f_1 and f_2 by P_1

// So, when two are more Problems arriving at same time then we are facing the race Condition.

// So, Here we will solve this Problem by Binary Semafore

// S[i] five Semafores

S_0, S_1, S_2, S_3, S_4

1 1 1 1 1

// lets assume all are initialized with 1

Let's rewrite the Code —

```
Void Philosopher(void)
{
    while(True)
    { Thinking();
      wait (take_fork(Si))
      wait (take_fork(S(i+1) mod N)
       Eat();
      Signal( Put fork(i));
      Signal ( Put fork( i+1) % N));
    }
}
```

// So, if Po want to take both fork then it needs two semafore value that is S0 & S1

↓ takefork(Si) ↓ take_fork(S(i+1) mod N)

// Like P1 need S1 & S2

// Like P2 comes then it needs S2 & S3

// Like P3 // then it needs S3 & S4

// Like P4 // // // // S4 & S0 ← Imp

↓ S(4+1) mod 5 //

= 0

initially S_0 S_1 S_2 S_3 S_4
1 1 1 1 1

Case-1 Now

$P_0 \rightarrow S_0$ S_1

will chang S_0 & S_1
1 1
↓ 0 0 (executing wait codes)

then will chang S_0 & S_1
0 0 (Executing Signal Code)
↓ ↓
1 1

Case-2

Let us assume P_0 arrives and it executes wait code i.e it decreaments S_0 & S_1 and down the value from 1 to 0

i.e S_0 & S_1
1 1
↓ ↓
0 0

Now P_0 started eating

* ⇒ let us assue at the same if P_1 arrives it will try to execute wait code and try to decrement the vake of Semafor S_1 which is already '0' value so it will be blocked as per semafor code

i.e P_0 P_1 P_2
↓ executing Critical Section
↓ Blocked
if P_2 arrives then it will decrement the value of S_2 & S_3 (So, it's executing in critical section)
S_2 S_3
↓ ↓
1 1
to to
0 0

// In Dining Philosopher Proble Mutual exclusion Principle is not valid (Special Case)

Imp

because if Philosophe's are independent then they can eat

So, Her P_0 & P_2 can run at the same time

Case-3

S_0 S_1 S_2 S_3 S_4

1 1 1 1 1

No Philosopher's are eating

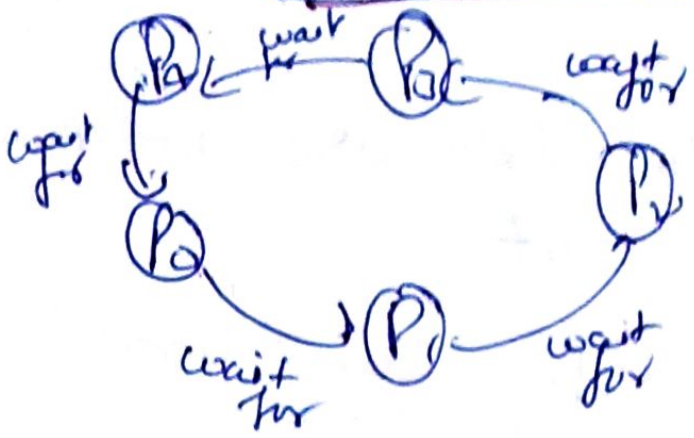
but there is situation of Deadlock even no Philosopher's are eating

let's assue P_0 arrive and it chaged the Value of S_1 from 1 to 0 i.e Successful but when P_0 is trying to pic S_1 i.e right fork it gets Prompted and the same time P_1 arrives and tries to get left fork i.e S_1 and chaged it from 1 to 0 (i.e Successful)

Now when P_1 is trying to pic right fork, the same time P_2 arrives

Like, Sue ... - - - -

ie Deadlock Situation



(Circular wait)

So, no Philosophers are eating but they just created a deadlock situation

Solution =>

P_0 S_0 S_1

P_1 S_1 S_2

P_2 S_2 S_3

P_4 S_3 S_4

→ P_5 S_0 S_4 // Just changed the code for fifth Philosopher

// change means the fifth Philosopher will pic right fork first

Let's see

S_0 S_1 S_2 S_3 S_4

1 1 1 1 1 (initially)

P_0 — S_0 S_1

P_1 — S_1 S_2

P_2 — S_2 S_3

P_3 — S_3 S_4

P_4 — S_0 X S_4 → changed Sequence

→ will be blocked because its value is already "0" done by P_0

// So, P_3 will gain access to S_4 ie right fork.

⇒ In this way will recover from Deadlock

ie we can reverse any one Philosopher's code

Final Conclusion for (N-1) Philosopher's we will run the code as it it but for N^{th} philosopher it will be like

wait (takefork ((i+1) mod N) } Reversed
wait (takefork (i))

Sleeping Barber Problem

(1) There is one barber, one barber chair and n waiting chairs

(2) If there is no customer, barber sleeps in his own chair.

(3) When customer arrives, he has to wake up barber.

(4) When many customers come and waiting chairs are empty, they sit on waiting chairs else they leave if no chairs empty.

(5) (i) Semaphore Customer — Counts waiting customers (excluding the customer in the barber chair, who is not waiting)

(ii) Semaphore Barber → Counts the no. of idle/active (0/1) barber

(iii) Semaphore mutex — used for mutual exclusion.

(6) Integer Variable waiting — also Counts the no. of waiting customers.

"the reason for using variable waiting is that there is no way to read the current value of Semaphore Customer

---

Chairs = N

Semaphore Customer = 0 (no of Customers in waiting room)

Semaphore Barber = 0 (Barber is Idle i.e Sleeping)

Semaphore Mutex = 1 (to Mutual Exclusion)

int waiting = 0 (No of waiting Customers)

Customer Process

```
while (True)
{
    wait(mutex);
    if (waiting < chairs)
    {
        waiting = waiting + 1;
        Signal (customer);
        Signal (Mutex);
        wait ( Barber);
        get_haircut();
    }
    else
        Signal (mutex);
}
```

Barber Process

```
while(True)
{
    wait (customer);
    wait (mutex);
    waiting = waiting - 1;
    Signal (barber);
    Signal (mutex);
    cut_hair();
}
```

it is a Deadlock free

Problem = Starvation // A queue can be used to add customers, FIFO order

Readers & Writers Problem

- A database is to be shared among several concurrent Processes
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**

→ Obviously, if two readers access the shared data simultaneously, no adverse effects will result.

→ However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, Chaos may occur.

⁂ To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database

→ Thus synchronization Problem is referred to as the readers-writers problem.

Solution to the Readers-Writers Problem using Semaphores: (Binary Semaphores)

Two semaphores

1- mutex, a semaphore (initialize to 1) which is used to ensure mutual exclusion when readcount is updated i.e when any reader enters or exit from the critical section

2- wrt (Db), a semaphore (initialized to 1) Common to both reader and writer processes

3- readcount (for readers), an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

writer

```
do{ /* writer requests for Critical Section */
    wait(wrt);
    /* perform the write */ (Db)
    // leaves the critical section
    signal(wrt);
} while(true);
```

Reader Process

```
do {
P/down/wait(mutex);
    readcnt++;   // The number of readers has now increased by 1
    if(readcnt == 1)
    wait(wrt); // this ensure no writer can enter if there is even on readers
    Signal(muter); // other readers can enter while this current reader is inside the critical section
```

CR → Db

```
/* Current reader Performs reading here */
    wait(mutex);
    readcnt--; // a reader wants to leave
    if(readcnt == 0)   // no reader is left in the critical section
    signal(wrt);   // writer can enter
    Signal(mutex);   // reader leaves
} while(true);
```