UNIT-2

CONCURRENT PROCESSES

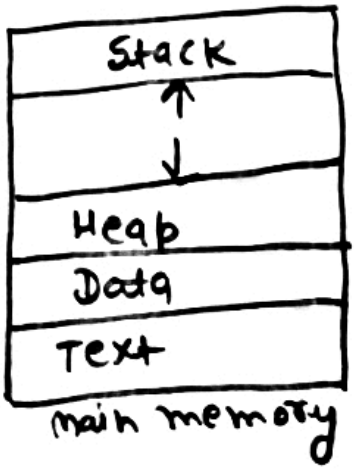
PROCESS:- A Process is a Program in execution including the current values of the Program counter, registers and variables.

The difference between a Process and a Program is that the Program is the group of instruction where as the Process is the activity.

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 the Program.

When a Program is loaded into the memory and it becomes a Process which Performs all the tasks mentioned in the Program.

When a Program is loaded into the memory and it becomes a Process it can be divided in to four section Stack, Heap, text and Data.



COMPONENT:-

(1) STACK:- The Process Stack contains the temporary data such as method function, Parameters return address, and Local variables.

(2) HEAP:- This is dynamically allocated memory to a Process during its runtime.

(3) TEXT:- This includes the current activity represented by the value of Program counter and contents of the Processor's Registers.

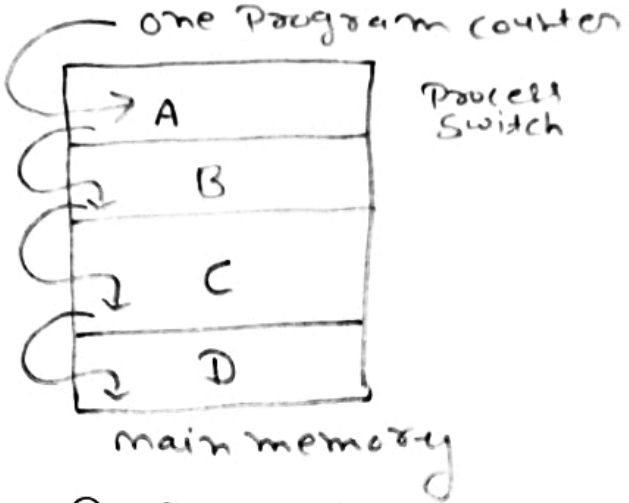
(4) DATA:- This section contains the all global and static variables.

PROCESS MODEL:- conceptually every Process Has its own virtual CPU- but in reality CPU switches back and forth from Processes to Process. but to understand the System. it is much easier to think about a collection of Processes running in (Pseudo) Parallel than to try to keep track of How the CPU switches a from Program to Program. This rapid switching back and forth is called multiProgramming.

(1) first model: multiProgramming:-

In this model there is four Program in memory. we have single Shared Processes by all Programs.

There is only one Program counter for all Programs in memory.

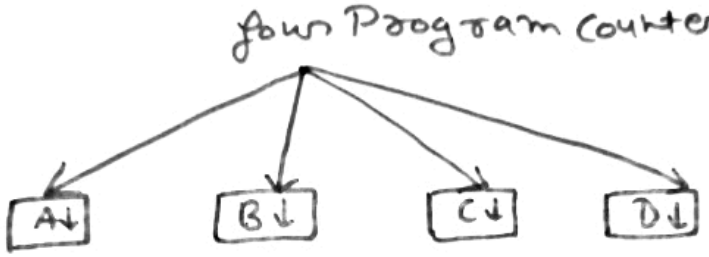


In this first - Program counter is initalized to execute the Part of Program A then with the help of Process switch it transfer control to Program B and execute same part of the. After that Program counter for Program C is active. and execute some Part of it and so on to Program D. Then all these step continues may be randoming with the help of Process Switcher until all Program task is not completed.

(2) Second model : multiProcessing :-

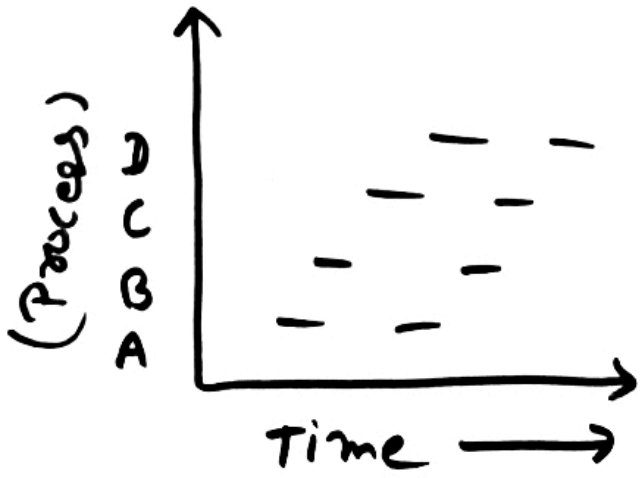
There is a 4 processes each with its own flow of control (i.e its own logical Program counter) and each one running independently of the other ones. There

There is only one Physical Program counter is loaded into the real Program counter. when it is finished (for the time being), the Physical Program counter is saved in the Process stored logical Program counter in memory.



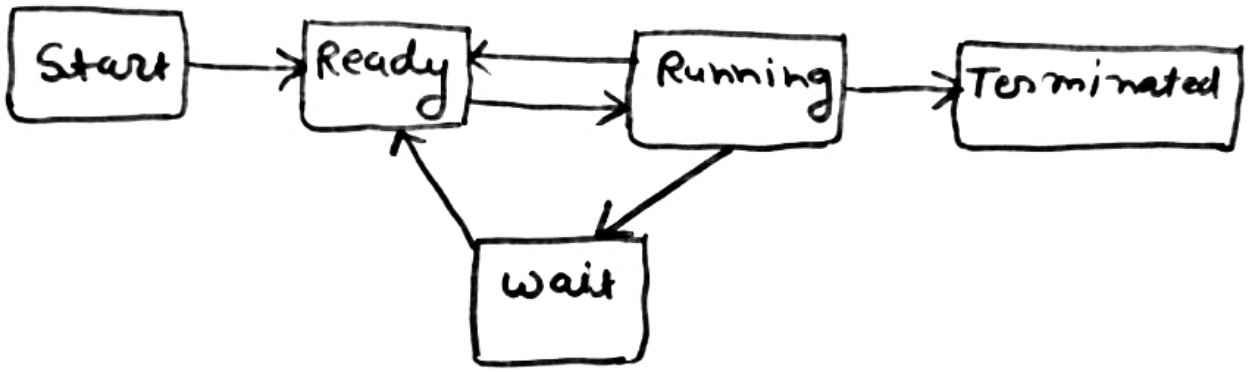
(3) Third Process model / one Programatone:-

At any given instant only one Process run and other Process after some interval of time. In other Point of view all Processes progress but only one Process actually runs at given instant of time.



PROCESS STATES:- When a Process executes, it passes through different states. these stages may differ in different operating systems.

In general a Process can have one of the following five states at a time.



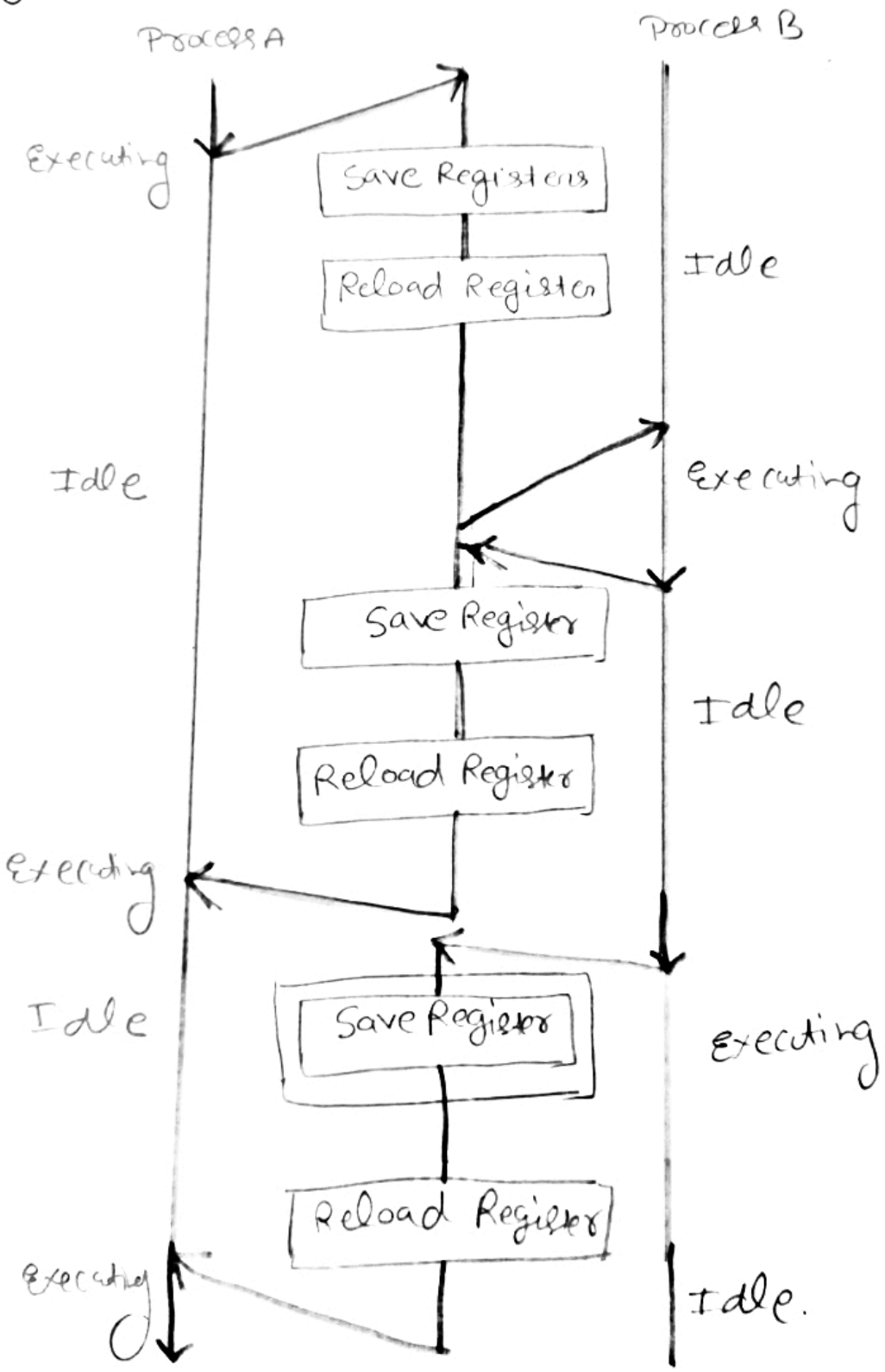
(1) **START:** This is the initial state when a Process is first started/created.

PROCESS HIERARCHY:-

Modern general Purpose operating system Permits a user to create and destory Processes. A Processes may create several new Processes during its time of execution the creating Process is called Parent Process. while the new Processes are called child Processes.

There are different possibilities concerning creating new Processes:-

the current information of Processes A in its PCB and context to the second Process namely Process B.



In doing so Program counter from the PCB of Process B is loaded and thus execution can continue with the new Process.

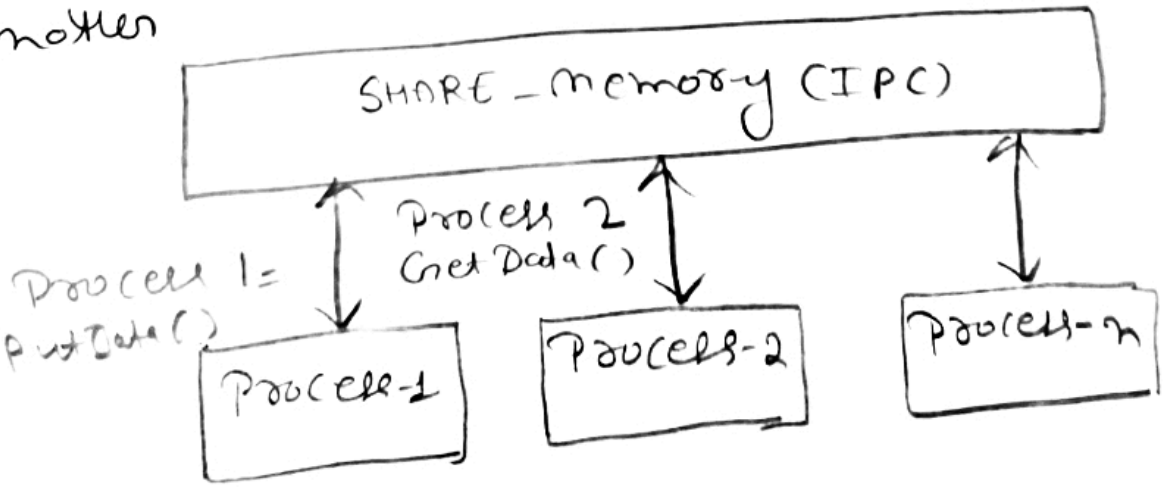
The switching between two processes, Process A and Process B needs PCB to save the state.

INTER PROCESS COMMUNICATION :- (IPC)

Inter Process communication (IPC) is a capability supported by operating system that allows one Process to communicate with another Process.

The Processes can be running on the same computer as an different computers connected through a network.

Inter Process communication (IPC) enables one application to control another application and for several application to share the same data with out interfering with one another

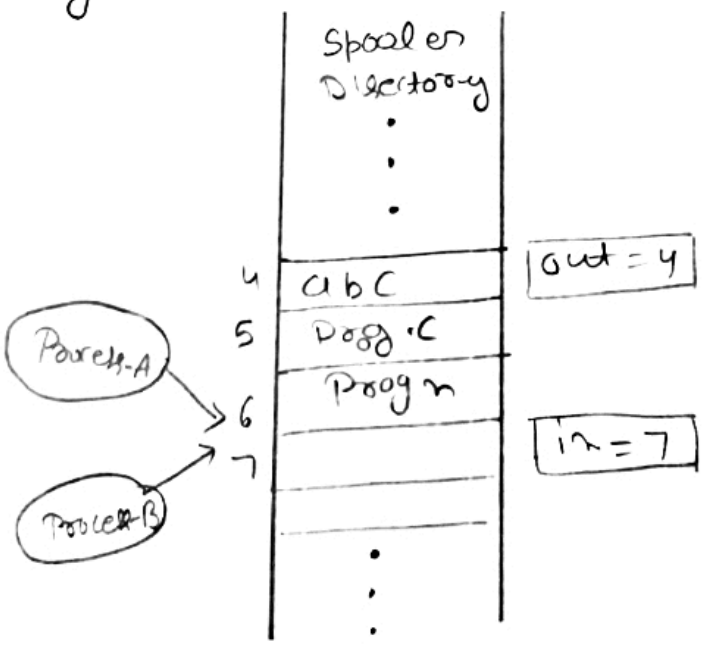


we need a well structured way to facilitate interprocess communication which maintain integrity of the system. Ensure Predictable behaviour.

RACE CONDITIONS!- The situation where two or more Processes are reading or writing some shared data and the final result depends on who runs Precisely when are called Race conditions.

ex

A Print spooler! when a Process wants to print a file it enters the file name in a special spooler directory. Another Process, the Printer Daemon, Periodically checks to see if there are any files to be Printed them and removes their names from the directory.



Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ... each one capable of holding a file name. Also imagine that there are two shared variables.

out: Points to next file to be Printed.

in: Points to next free slot in the directory.

Slots 0 to 3 files already Printed.

Slots 4 to 6 files names which has to be Printed.

PROCESS SYNCHRONIZATION:-

Producer-Consumer Problem:-

Producer-Consumer Problem is also called Bounded-Buffer the Problem describes two Processes. the Producer and the Consumer, who share a common fixed size buffer.

Producer Consumer Problem also known as the bounded-buffer Problem is a multiProcess Synchronization Problem.

Producer:- The Producer's job is to generate a piece of data, put it into the Buffer and Store again.

consumer:- The consumer is consuming the data (i.e removing it from the

buffer) one piece data time.
if the buffer is empty then a consumer should not try to access the data item from it. similarly a Producer should not Produce any data item if the buffer is full.

Counter counts the data items in the buffer. or to track whether the buffer is empty or full. Counter is shared b/w two processes and updated by both.

How it works!

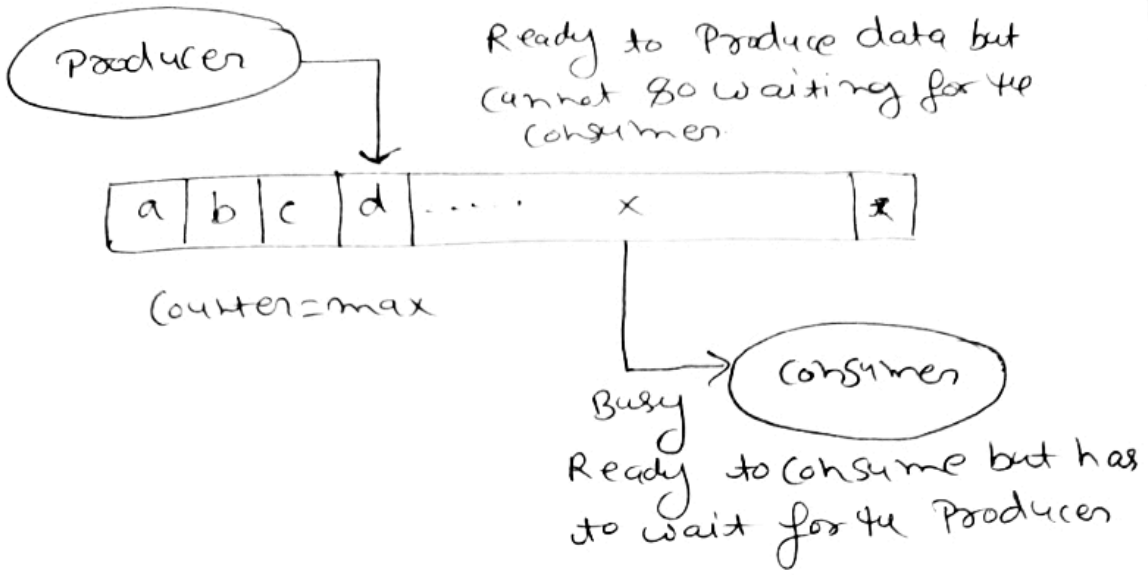
counter value is checked by consumer before consuming it. if counter is 1 or greater than 1 then it starts executing the process and updates the counter.
Producers checks the buffer for the value of counter for adding data.
if the counter is less than its maximum value it means that there is some space in the buffer

It starts executing for Producing the data items and updates

Let max = maximum size of the buffer.

if buffer is full then counter = max

and consumer is busy executing other instructions or has not been allocated its time slice yet.



In this solution buffer is empty, that is counter=0 and the Producer is busy executing other instructions or has not been allocated its time slice yet. At this consumer is ready to consume an item from the buffer.
Consumer waits until counter=1

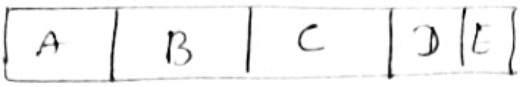
Producer

```
do
{
wait (E);      [wait until empty>0 and then decrement
wait(S);  → Lock
//Add item to Buffer
Signal(S): →Released
Signal(F); // increment full
}
while(true);
```

Consumer

```
do
{
wait (F);  → until full>0 and then full ↓
wait(S);   //Acquire Lock
//consume items
signal(S); //Released
signal(E); //nEmpty
}
while (true);
```

Ex



Semaphore F = 0 = 0+1 = 1
↳ Full

S = 1 — only one Process is accessing critical Section.

E = n (size of the buffer) — empty E = 5 = 5-1 = 4

wait → decrement the count

Signal → Increment the count

E = 0 ⇒ 0+1 = 1, F = 5-1 = 4

CRITICAL SECTION PROBLEM :-

Consider a System consisting of n Processes (P0, P1, - - - - Pn)

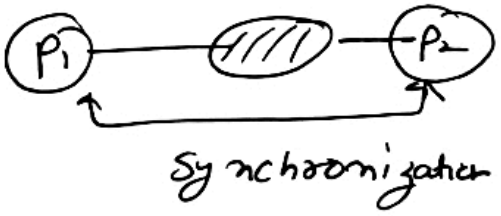
Each Process has a Segment of code called a critical section at which Process may be changing a common variables, updating a tables, writing a file etc.

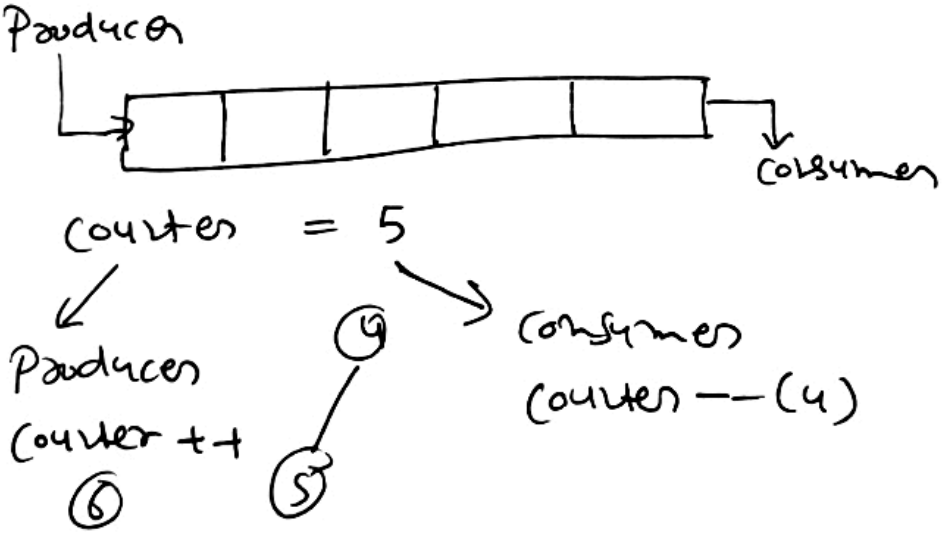
Race condition :- It is the situation where several processes access and manipulates some data concurrently and the out come of execution depends on the particular order in which access takes place.

Cooperating Process:- Cooperating Process is the Process that can affect or be affected by other Processes executing in the system.

P_1
P_2
:
:
P_n

Share logical address space
or
Share data through file or message.





Problem arises in concurrent Access to shared data and it may lead to inconsistency.

CRITICAL SECTION PROBLEM :-

Critical Section is a code segment that accesses Shared variables and has to be executed as an atomic action.

only one Process must be executing to critical Section at a given time.

```
do
{
entry. Section
Critical. Section   } only one Process can enter
Exit - Section
remainder-Section
}
while (true);
```

entry. Section => Gives locks to a Process to enter critical Section.

Exit. Section => remove the Lock

Solution criteria for critical. Section :-

(1) mutual exclusion
(2) Progress
(3) Bounded waiting

(1) MUTUAL EXCLUSION :- In mutual exclusion only one process should execute in its critical section.

(2) PROGRESS :- if no process is executing in its critical section and some process wish to enter the critical-section then only those processes that are not executing in its remainder section can participates.

(3) BOUNDED WAITING :- There is a limit on the no of times that other process are allowed to enter their critical section.

} Mutual exclusion and Progress are mandatory
} Bunded waiting is optional.

CRITICAL PROBLEM SOLUTION :-

Two way solution by using turn variable :-

It is a software mechanism implemented in user mode. It is a busy waiting solution. It can be implemented only for two processes.

for process P_0 & P_1

Solution-1 (Binary variable) turn = 0/1

for Process (P_0)

```
while (1)
{
entry section { while (turn != 0);   // true
if false turn=0 → Po in [critical-section]
turn = 1
exit
Remainder section
}
```

for Process (P_1)

```
while (1)
{
while (turn != 1);   // true
if false turn=1 → P1 in [critical-section]
turn = 0;
exit
Remainder section
}
```

(1) M.E - ✓
(2) P - ✗
(3) B.W - ✗

two way- Process by using Flag variable

Solution-2'

Array

	P_0	P_1
Flag	F	F
	T	T
	F	F

for Process (P_0)

```
while (1)
{
entry section { while (flag[1]);   // true
if false flag[1]=T → Po in [critical-section]
```

for Process (P_1)

```
while (1)
{
while (flag[0]);   // true
if false flag[0]=T → P1 in [critical section]
```

Flag [0] = F	Flag [1] = F
Exit	Exit
Remainder Section	Remainder Section
}	}

{ mutual Exclusion = ✓
Progress = x
Bounded wait = x

Lock variable (Synchronization mechanism)

Lock variable it is a Software mechanism implemented in user mode. It is a busy waiting solution. It can be used for more than two Process.

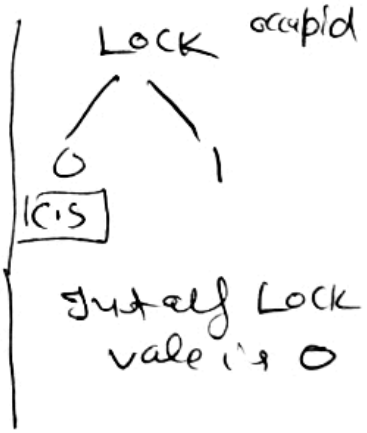
entry section
① while (Lock ! = 0);
② Lock = 1;

P1 in Critical Section

Lock = 0;

Exit

}



Test Set Lock (TSL) mechanism:-

Lock variable mechanism

entry section | while (test_set (lock));

Critical Section

lock = 0;

exit

}

Solution 3

PETERSON'S SOLUTION:-

It is a Software mechanism implemented at the user mode. It is a busy waiting solution can be used for two process.

Peterson's solution is used for mutual exclusion and allows two Process to share a single use resource without conflict.

It is user only shared memory for communication. Peterson's solution originally worked only with two Processes, but hence been generilized for more than two.

Before using the shared variables (i.e before entering its critical region. Each Process calls enter region with its own Process number 0 or 1 as Parameters) Their call will (a use it to wait if need be until it is safe to enter.

So

```
for Process(P0)                    for Process(P1)
while(1)                           while(1)
{                                  {
flag[0] = T;                       flag[1] = T;
turn = 1;                          turn = 0;
while(turn==1 &&                   while(turn==0 &&
      flag[1]==T);                       flag[0]==T);
critical section                   critical section
flag[1] = F;                       flag[1] = F;
exit remainder section             exit } remainder section
}
```

SEMAPHORES:-

Semaphores is a very popular tool used for Process Synchronization Semaphore is used to Protect

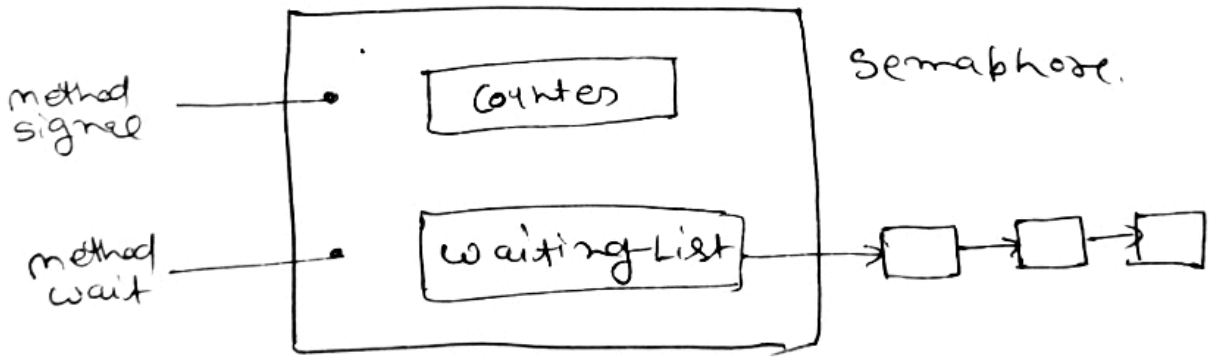
Protect any resources such as global shared memory that needs to be accessed and updated by many Process simultaneously

semaphore act as a guard / Lock on the Resources.

When ever a Process needs to access the resource, it first needs to take Permission from the semaphore.

semaphore gives Permission to access a resource if resource is free otherwise Process has to wait

The semaphore is implemented by variables Like counter, a waiting List of Process and two methods (e.g function) signal and wait with integer values.



The Semaphore is accessed by only two Indivisible operations knows as wait and Signal operations which is denoted by P & V.

Ex

Process performs "wait" operation when tries to enter "critical section", In this way the solution to critical section using semaphore satisfied the designed protocal.

The semaphore whose value either 0 or 1 is know as Binary semaphore. This concept is basically used in mutual-execlusion.

Syntax

```
do
{
wait (semaphore)
{
critical section
}
Signal (semaphore)
{
- - -
- - -
}
```

Primitive semaphore operation:-

```
wait (S)
{
while(S<=0);
  S = S-1;
}
signal (S)
{
S = S+1;
}
```

```
int S= 1/0;
do
{
wait(S);
// critical section
  signal(S);
// Exit section
while(T);
```

mutual exclusion = ⌊

Progress = ⌊

Bound of wait = ⌊

PROCESS SYNCHRONIZATION:-

Types of Process:-

(1) Independent Process
(2) Co-operative Process
  (a) Shared variables
  (b) memory
  (c) Code
  (d) Resources <- CPU, printer

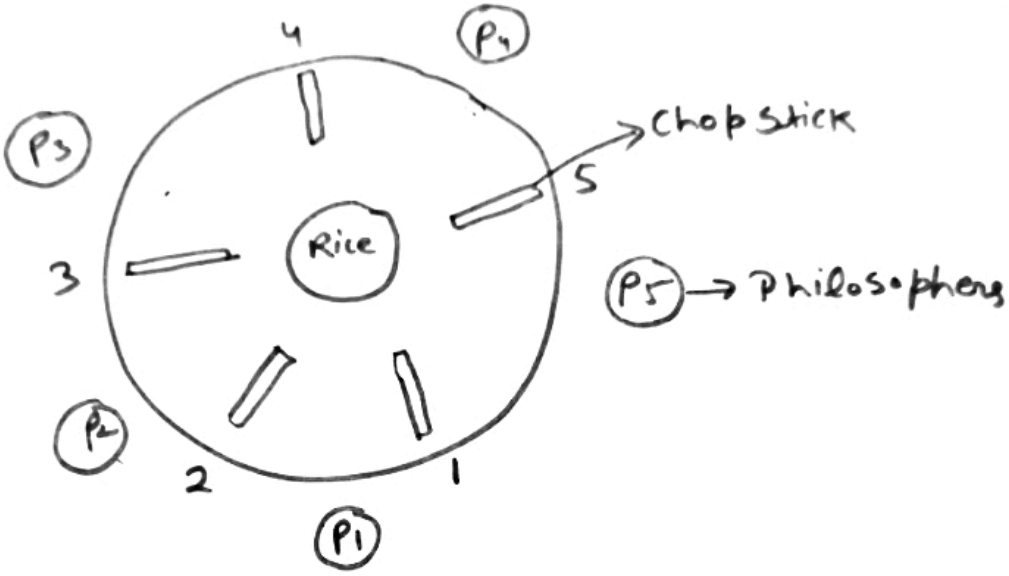
Race condition:

int Share = ~~10~~ // 9

P_1		P_2	
$x = 10$	int x = Share	int y = Share	y = 10
$x = 11$	x = x + 1	y = y - 1;	y = 9
	Share = x;	Share = y	Share = 9
	Share = 11		
	↗		
	Updated value		

DINING PHILOSOPHER'S PROBLEM:

It is a classical Problem of Process Synchronization because it demonstrate a Large class of concurrency control Problem.



Five Philosophers are sitting on a round table and are in a thinking - eating - cycle.

when a Philosopher gets hungry. he needs to pick-up two nearest Chop-Sticks and eats.

A Philosopher can pick-up one Chopsticks at a time.

The dining Philosopher Problem is ~~number~~ to choose (two Chopsticks one his own and one of another.

```
Void  Philosphes
{
while (1)
{
think ();
take_ chopstick[i];
take_ chopstick [(i+1)%5];
eat()  critical-section
put_ chopstick[i];
put_ chopstic[(i+1)%5];
think();
}
}
```

entry code

exit code

OR

Solution for using semaphore:-

```
semaphore chopstick[5]=1;
Philosopher = i;
do
{
wait (chopstick[i]);
:
wait (chopstick (i+1) %5);
:
```

```
eat
signal ((chopstick[i]);
signal ((chopstic (i+1)%5);
:
think.
}
while(1)
```

Although solution guarntees that no two neighbours are eating simultaneously but it has the possibility to creating deadlock.

Suppose all 5- philosophore grab their right chopstic simultaneously.

wait operation when 1 or more than 0 1.

```
S > 0
wait (S)
{
S = S-1;
} signal (S)
{
S = S+1;
}
```

MONITOR:-

A monitor is a programming language construct that controls access to shared data.

```
Type < monitor type > = monitor
- - - data declaration
- - -
monitor entry < name and its parameter>
{
-
}
```

Monitor is a collection of Procedures variables and data structures that all are grouped in a special kind of module or package is known as monitor.

There are operating system resources executes in kernel mode.

It Provides atomic (Implicit) mutual exclusion

⇒ only one thread can be executing inside monitor at a time.

if another thread / Process tries to execute a monitor Procedure, it blocks until the first one left the monitor.

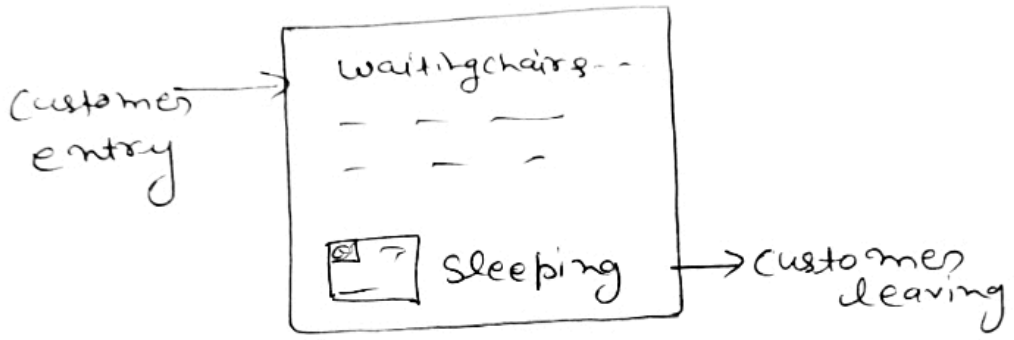
It is more restrictive than Semaphore.

⇒ It is easy to use (most of the time)

```
monitor dining Philosopher
{
enum { thinking, Hungry, Eating };
    int state[5];
    condition x[5];
    void take_fork()
    {
    }
```

SLEEPING BARBER PROBLEM:-

In computer system the sleeping Barber Problem is a classic interProcess communication and synchronization Problem between multiple operating system Process.



Condition:

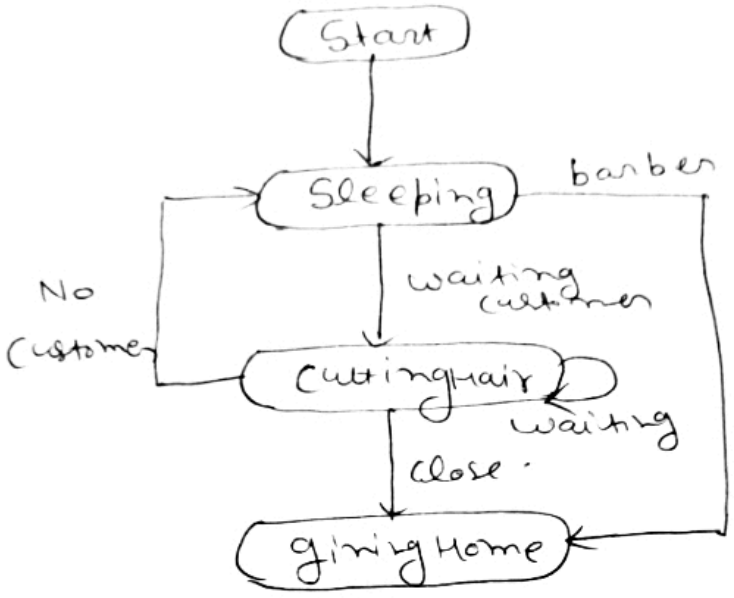
one waiting room with N Chairs.
one barber room with 1 barber Chair

Barber and Customer:-

if there are no customers the barber goes to sleep
if customer enter the shop if all Chairs are occupied the customer leaves the shop, otherwise he sits in one of the free chair.

⇒ If a barber is a sleep the customer wakes by the barber

if the barber is busy and one chair is free.



Now we write the Program to coordinate the actions of the barber and customer.

Sleeping Barber Solution:-

variables: Shared data

Semaphore customer = 0;

Semaphore barber = 0;

access Seate mutex = 1;

Barber:

while (true)

{

wait (customer); // waits for the customer (asleep)

```
wait(mutex); // whenever wait(1) is executed it decrement value of(0) is mutex to [illegible] the no of available seat
No of the seat ++; // a chair is free.

sem_post(Barber); // customer for hair cut
sem_post(mutex); // release the mutex on the chair
Barber is cutting hair
}
```

Customer:

```
while(1)
{
sem_wait(access seats);
if(Number off free seat > 0)
{
Number off free seat --; // sits down
```

```
sem_post(customers);
sem_post(access seat); //release the lock
sem_wait(barber); // wait in the waiting room if barber is busy

customer is having Haircut
    }
else
  {
  sem_post(access_seats); //release the lock
  }
}
customer leaves
```