Operating System : Processes

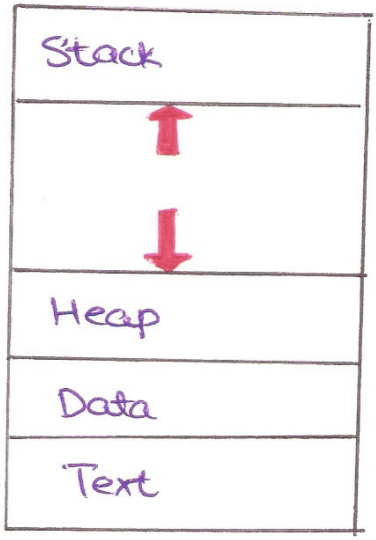
What is a process?

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, it can be divided in to four sections- Stack, Heap, text and Data.



Main Memory

S.NO	Component & Description
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 & Contents of the processor's Registers.
4.	Data: This section contains the all global and static variables.

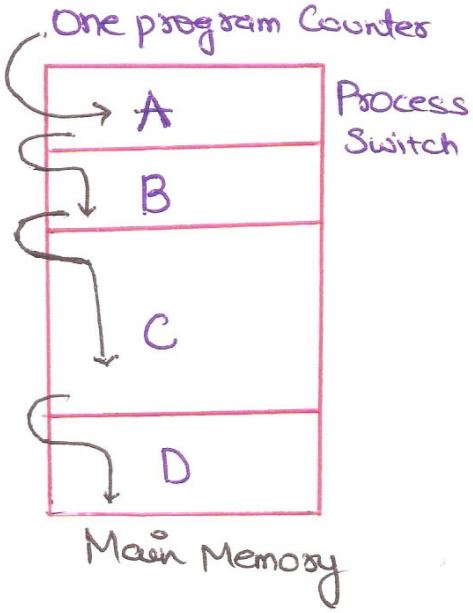
Processes :- The Process Model.

Conceptually, every process has its own Virtual CPU-but in reality CPU Switches back and footh from process 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 from program to program. This rapid Switching back and footh is called Multiprog-ramming.

First Model : Multiprogramming :

In this, there is four programs in Memory.

We have Single Shared processor by all programs

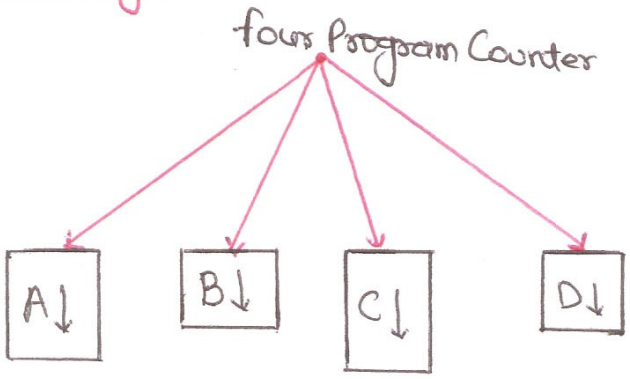


There is only One program Counter for all programs in memory.

In this first - program Counter is Initialized to execute the part of program A then with the help of Process Switch it transfer Control to program B and execute Some Part of this. 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 randomly with the help of process Switches until all program

2nd Model : Multiprocessing :-

There is 4 processes, each with its own flow of control (i.e its own logical program Counter), and each

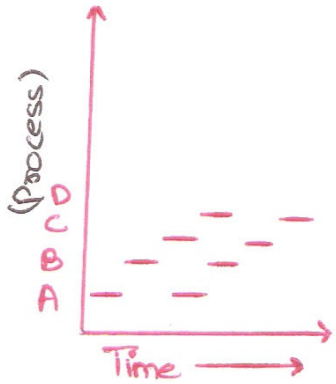


One running independently of the other ones. There is only one physical program Counter, so when each process runs, its logical program Counter is loaded into the real program Counter. When it is finished (for the time being), the physical program Counter is Saved in the process stored logical program Counter in Memory.

3rd process Model : One program at one

At any given instant only one process run and Other process after some interval of time.

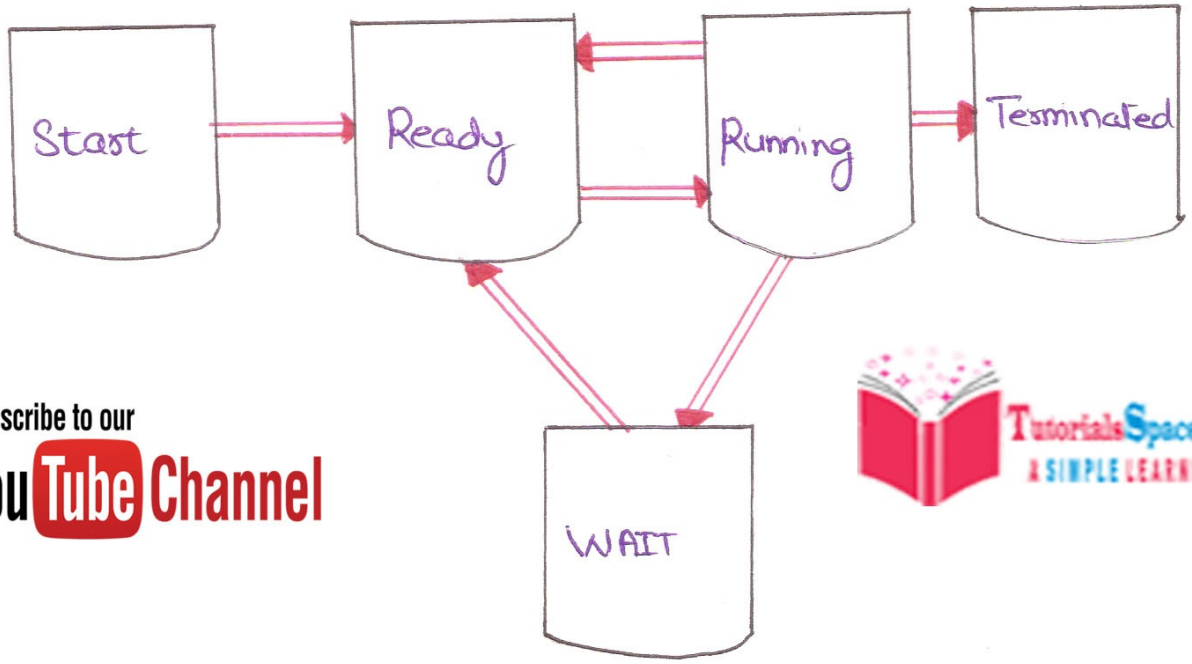
in Other point of view all processes progress but only one process actually runs at given instant of time.



PROCESSES :- 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.

2. Ready :- The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating System so that they can run. Process may come in to this State after START State or while running it by but interrupted by the Schedular to assign CPU to some other process.

3. Running :- After Ready State, the process State is set to running and the processor executes its instruction.

Waiting: Process moves into the waiting State if it needs to wait for a resource, such as waiting for user input, or waiting for a File to become available.

Terminated or Exit: Once the process finishes its execution, or it is terminated by the operating System, it is moved to the terminated State where it waits to be removed from Main Memory.

Process Hierarchy

Modern general purpose O/S permits a user to create and destroy processes. A process 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:-

Execution : The parent process continues to execute Concurrently with its children processes or it waits until all of its children processes have terminated (Sequential)

Sharing : Either the parent and children processes share all resources (likes memory or files) or the children processes share only a subset of their parent's resources or the parent and children process share no resources in common.

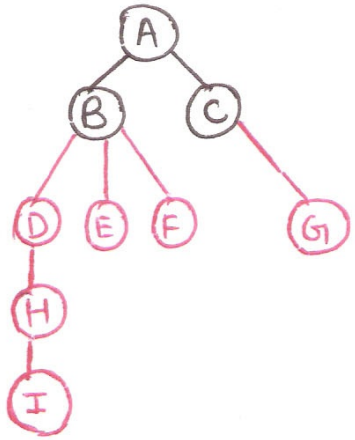
A Parent process can terminate the execution of one of its children for one of these reasons:-

1) The child process has exceeded its usage of the resources it has been allocated. for this a mechanism must be available to allow the parent process to inspect the state of its children process.

2) Task Assigned to child process is no longer required.

Let us discuss this with an example :-

In Unix this is done by the 'fork' System call, which creates a 'child' process and the 'exit' System call, which terminates current process.



A forks → B & C
B forks → D, E & F
C forks → G
D forks → H
H forks → I

The root of tree is a special processes created by the OS during startup.

A process can choose to wait for children to terminate

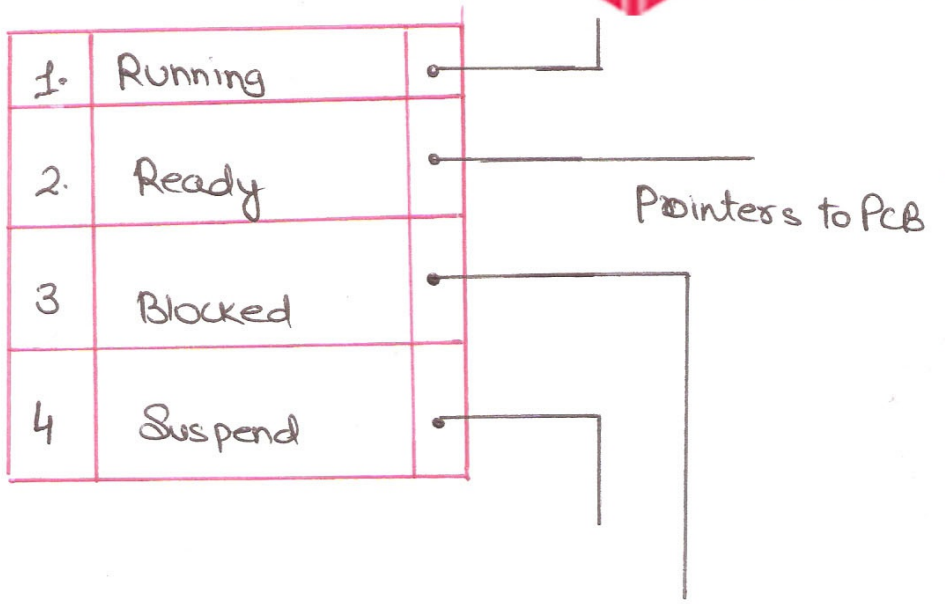
Example C issued a wait() System call it would block until G finished.

Process : Data Structure Used - Process Table & Process Control Block (PCB)

Process Table : The OS manages and controls the resources by having a table. Tables are important Data structures to store information about every process and resources.

This is the reason the OS maintains memory tables, I/O tables, and process tables.

The process tables store the ID of every process and corresponding to it, the pointer to its PCB.

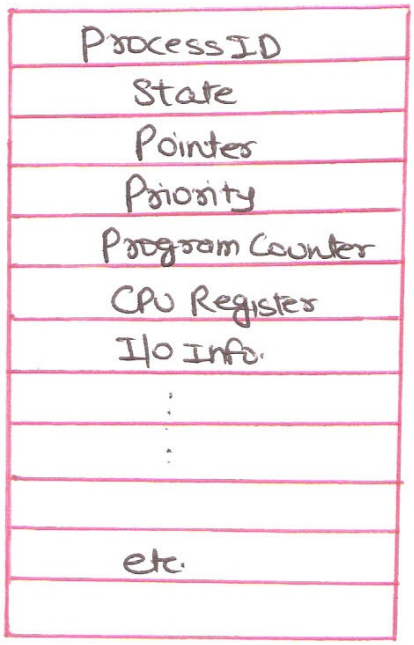


Process Control Block :- A process control block is a data structure maintained by the OS for every process.

** Some operating system maintain only PCB in that PCB has all entries stored by a process Table. **

A PCB keeps all the information needed to keep track of a process:-

1. Process State: Current state of the process i.e, Ready, Running or waiting etc.
2. Process privileges:- Allow / disallow access to System Resource.
3. Process ID: Unique identification for each of the process in the Operating System.
4. Pointer: A pointer to parent process.
5. Program Counter: program counter is a pointer to the address of the next instruction to be executed for this process.
6. CPU registers: Various CPU registers where process need to be stored for execution for running state.
7. CPU Scheduling info.
8. Memory Management information: info. of page table, memory limit, Segment table info.
9. Accounting info:- It includes the amount of CPU used for process execution, time limits, execution ID etc.
10. IO Status information: This includes a list of I/O devices allocated to the process.



The PCB is maintained for a process throughout its life time, and is deleted once the process terminates:-

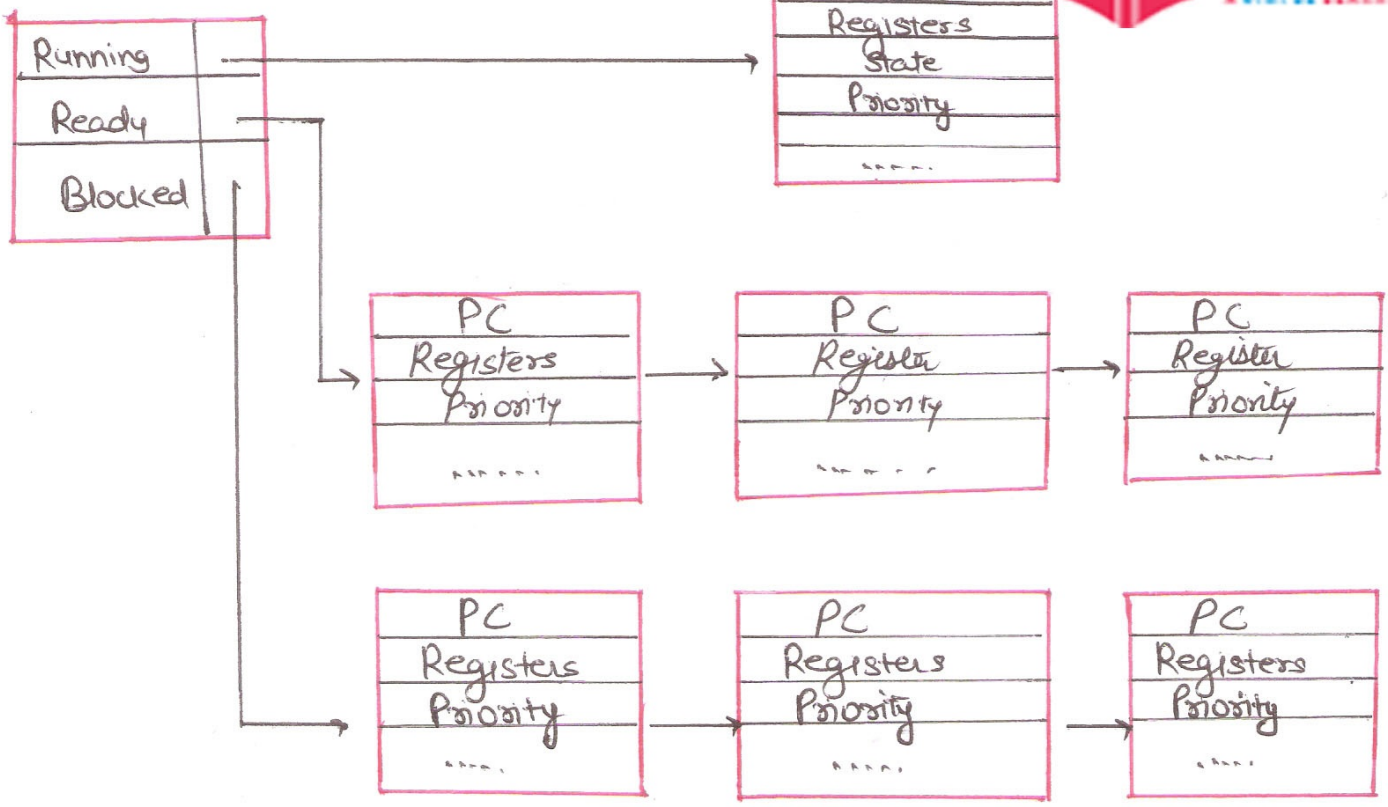
Implementation of processes

Process Model is implemented by the process Table and process Control Blocks which keep track all information of process.

→ At the time of creation of a new process, the OS allocates a memory for it loads a process code in the allocated memory and setsup data space for it.

→ The state of process is stored as 'New' in its PCB and when this process move to Ready state its state is also changes in PCB.

→ When a running process needs to wait for an I/O device, its state is changed to 'Blocked'. The various queues used for this which is implemented as linked lists.



There are mainly following queues.

Read queue: for storing the processes with State Ready

Blocked queue: for storing the processes that needs to wait for an I/O device or a Resource.

Suspended queue :- for storing the blocked processes that have been suspended.

Free-process queue : for the information of empty space in the memory where a new PCB can be created.

Each PCB has a pointer that points to next PCB. There is a header for each type of queue. The header stores the information about the first and last PCBs in that queues.

There is one more header giving the information about the running process information; however there is no queue of running processes because there is only one process in the System.

One more queue → information about process' Area → which free after the termination of a process which is free or release the memory after process termination. and this memory area can be used for a new process.

Besides changing the state of a process, PCB saves its position for further resumption where it left off. Initially, the process program Counter, program Status word, registers and other information are stored on the stack.

After this, the stack information is stored in the corresponding PCB of the process.

This saving of the status of the running process in its PCB is called 'Context Saving'.

After saving the context of a process, the appropriate event - handling function is executed.

for example for Wait State for an I/O device, then it is sent to the queue of the blocked processes.

→ After this there is requirement that another process be dispatched to the CPU as current process has been interrupted.

Therefore, Scheduler is called to schedule a process from the ready queue.

After selecting the process from Ready queue, its PCB is loaded and dispatched to the CPU for execution.

In this way, the processes are implemented by saving their context in PCBs and making State change possible

PROCESS :- CONTEXT SWITCH

A Context Switch (also sometimes reffered to as a process Switch or a task Switch) is the Switching of the CPU from one process or to another.

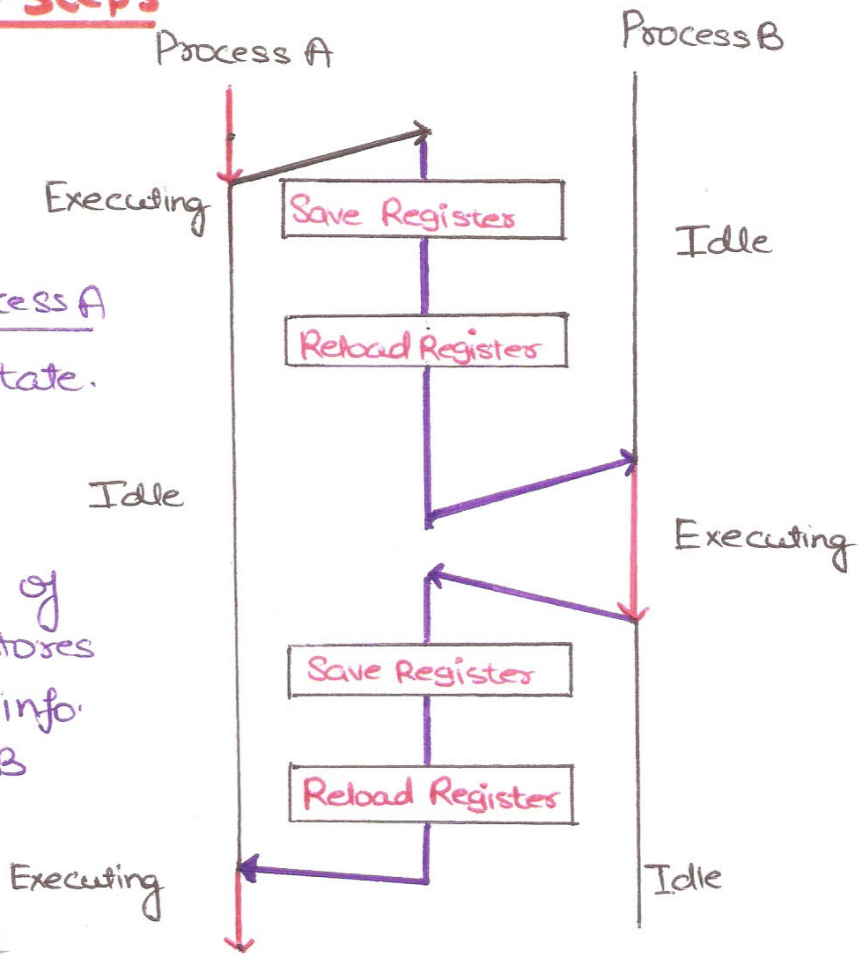
A Context is the contents of a CPU's registers and program Counter at any point in time.

Context Switch - Steps

If two processes A and B are in ready queue.

If CPU is executing process A and process B in wait state.

If an interrupt occurs for process A, the OS Suspends the execution of the first process, and stores the store the Current info. of process A in its PCB and Context ~~the~~ 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)

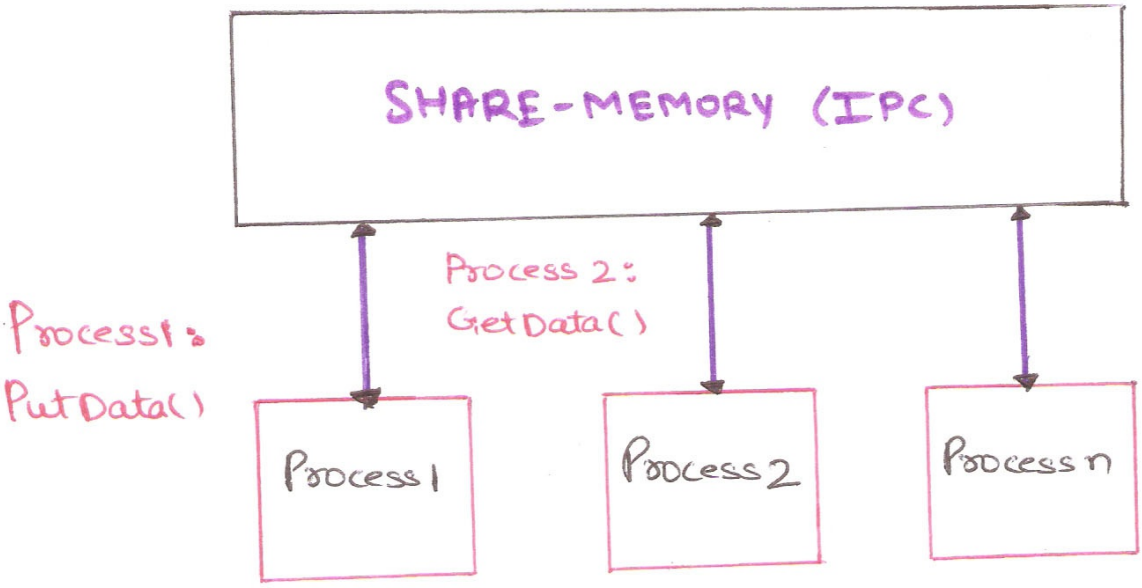
Race Conditions:-

What is 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 or on different Computers Connected through a network.

IPC enables one application to Control another application, and for several applications to share the Same data without interfering with one another.



We need a well structured way to facilitate interprocess Communication which

→ Which Maintain integrity of the System.

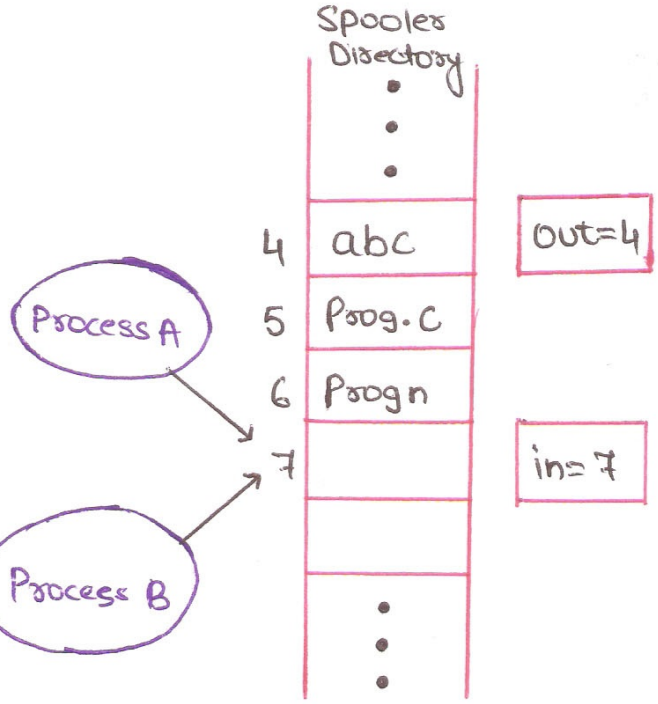
→ Ensure predictable behavior.

RACE CONDITIONS:-

The Situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when are called RACE CONDITIONS.

Lets take an example :-

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, and if there are, it prints them and removes their names from the directory.



Ex: Imagine that our spooler directory has a large no. 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.

NOW The Main ISSUE Comes :

Process A reads in and stores the value, 7, in a local variable

the CPU decides that process A has run long enough, so it switches to process B.

Process B also reads in, and also gets a 7, so it stores the name of its in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks 'next-free slot', finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes next_free_slot+1 which is 8, and sets in to 8.

The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong; but process B will never receive any output.

CRITICAL SECTION & Mutual Exclusion

Critical Section or Critical Region:- That part of the program where the shared memory is accessed is called critical section

To avoid race condition we need Mutual Exclusion.

Mutual Exclusion :- It is some way of making sure that if one process is using a shared variable or file, the other process will be excluded from doing the same things.

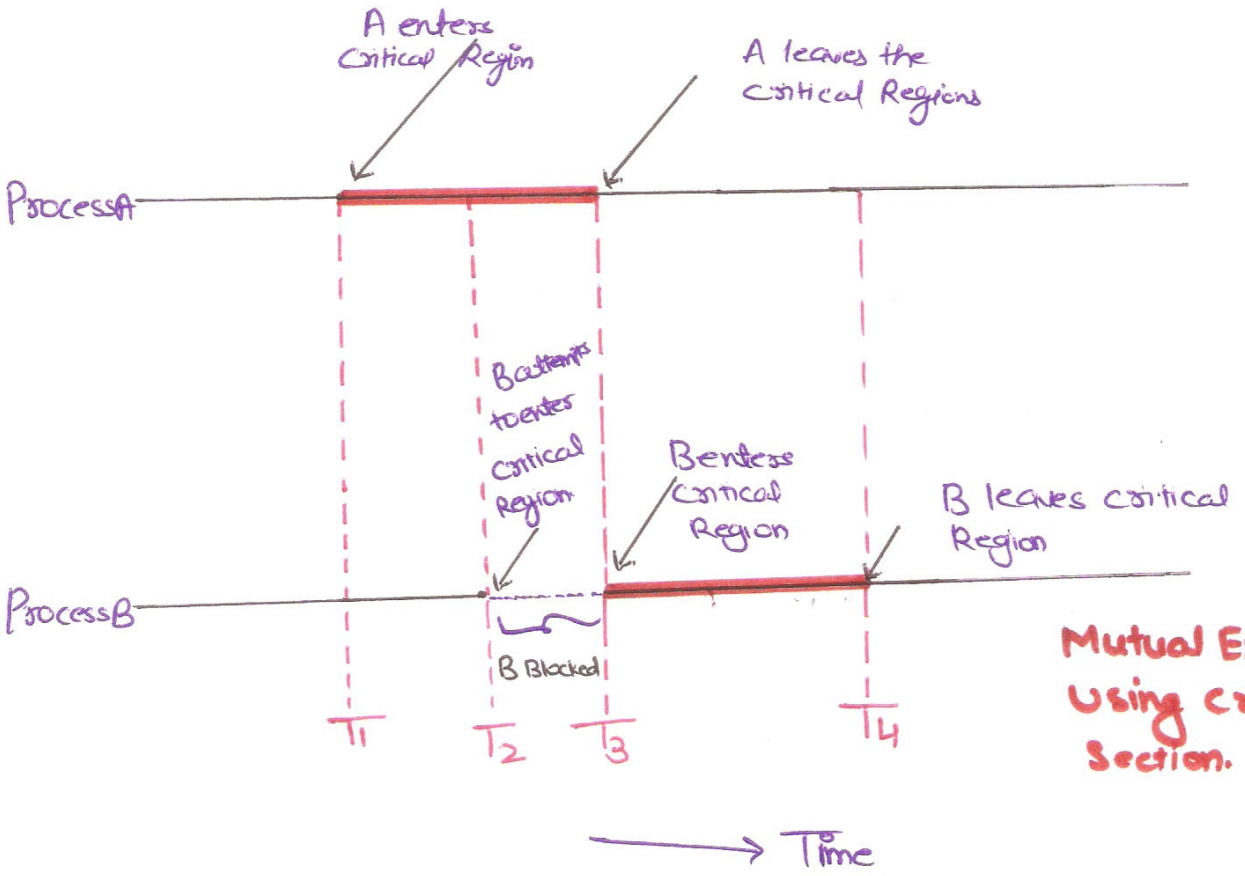
The difficulty in printer spooler occurs because Process B started using one of the shared variables before process A was finished with it.

If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions

Solution to Critical Section problem :

1) No two processes may be simultaneously inside their critical regions.
2) No assumptions may be made about speed or the number of CPUs.
3) No process running outside its critical region may block other processes.
4) No process should have to wait forever to enter its critical Regions.

Process A enters its Critical Region at time T_1. A little later, at time T_2 Process B attempts to enter its Critical region but fails because another process is already in its Critical region and we allow one at a time only.



Mutual Exclusion Using Critical Section.

Consequently, B is temporarily Suspend until process A [redacted] leaves the Critical Region.

After Process A leaves the Critical Region, Process B enter the Critical Region.

Peterson's Solution for Critical Section

Peterson's Solution is used for mutual exclusion and allows two processes to share a single-use resource without conflict. It uses only shared memory for communication. Peterson's Solution originally worked only with two processes, but hence been generalized 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 parameter. This call will cause it to wait, if need be until it is safe to enter.

ANSI C Code

```
Sdefine FALSE 0
Sdefine TRUE = 1
#define N
 int turn;
INT interested [N]
 Void enter_region (int Process).
{  int other
    other = 1-process;
   interested[process = TRUE;
    turn = process;
while(turn == process&& interested[other]==TRUE)
}
 Void leave_region (int process)
 { interested f process -f ALSE
 }
```

After it has finished with the shared variables, the process calls Leave_region to indicate that it is done and to allow the other

process to enter.

→ process 0 calls enter^region by setting its array element and sets 'turn' to 0.

→ If process 1 now makes a call to enter^region, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls Leave^region to exit the critical region.

Now Consider the case that both processes call enter-region almost simultaneously. Then both will store their process no. in turn. Which ever store is done last is the one that counts. the first one is overwritten and lost. Suppose that process 1 stores last, so turn is 1, When both processes come to the while statement, process 0 executes it zero times and enters its critical region.
Process 1 loops and does not enter its critical region until process 0 exists its critical region.

Producer-consumer problem

Producer-Consumer Problem :- 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 start again.

Consumer : The consumer is consuming the data (i.e removing it from the buffer) one piece at a 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 2 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.

→ -lly Produces checks the buffer for the value of Counter ~~fo~~ for adding data.

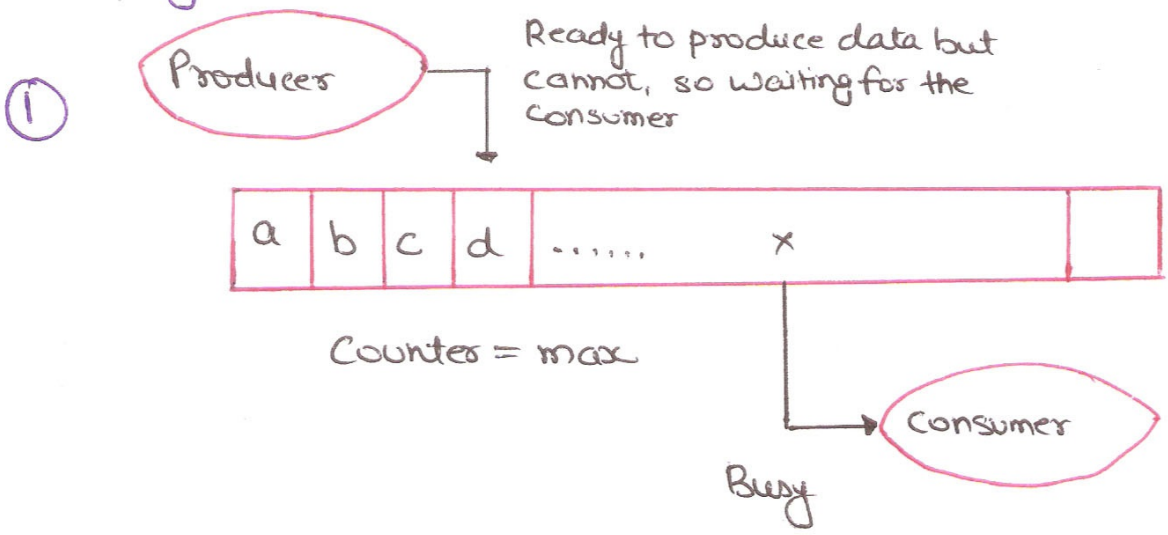
→ If the Counter is less than its maximum value, it means that there is some space in the buffer.

It starts executing for producing the data item and updates the

Counter by incrementing it by one.

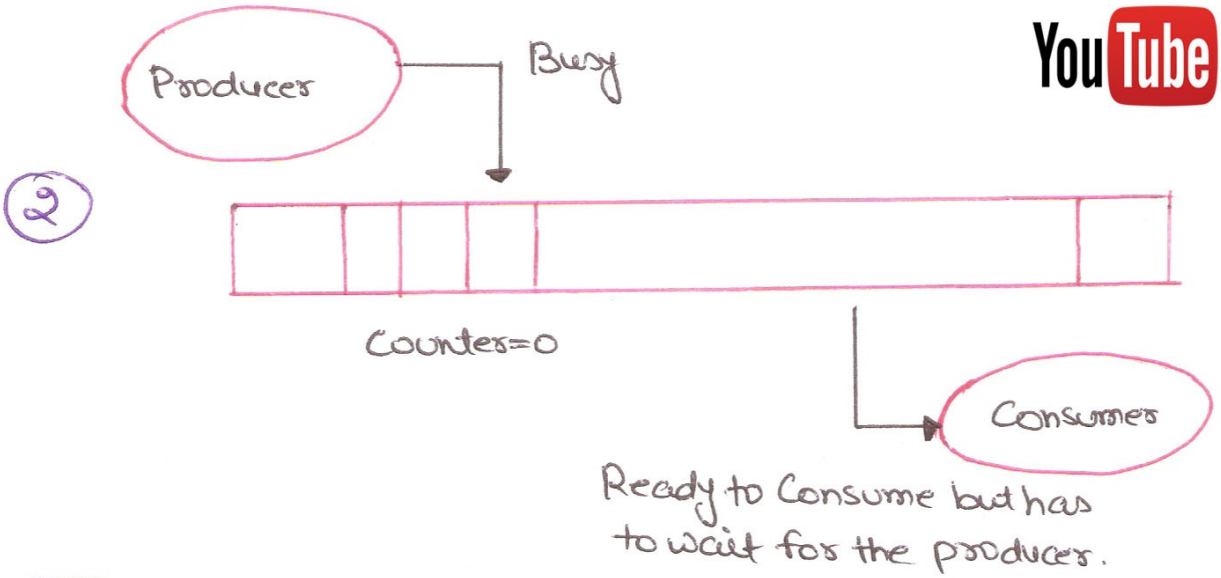
Let max = maximum size of the buffer.

If buffer is full then Counter = max and Consumer is busy executing other instructions or has not been allotted its time Slice yet.



Producer-Consumer problem : Buffer is full.

In this buffer is full so producer has to wait until consumer set counter by decrementing its value by 1.



In this Situation, buffer is empty, that is Counter = 0, and the Producer is busy executing Other instructions or has not been

allotted its time slice yet. At this consumer is ready to consume an item from the buffer.

Consumer waits until Counter = 1

When the buffer is empty and producer busy in filling data items in buffer in while Consumer goes to SLEEP.

When the Counter goes to 1 then System generates WAKEUP Calls to make Consumer to wakeup & start executing it.

SEMAPHORES

Semaphore is a very popular tool used for process synchronization.

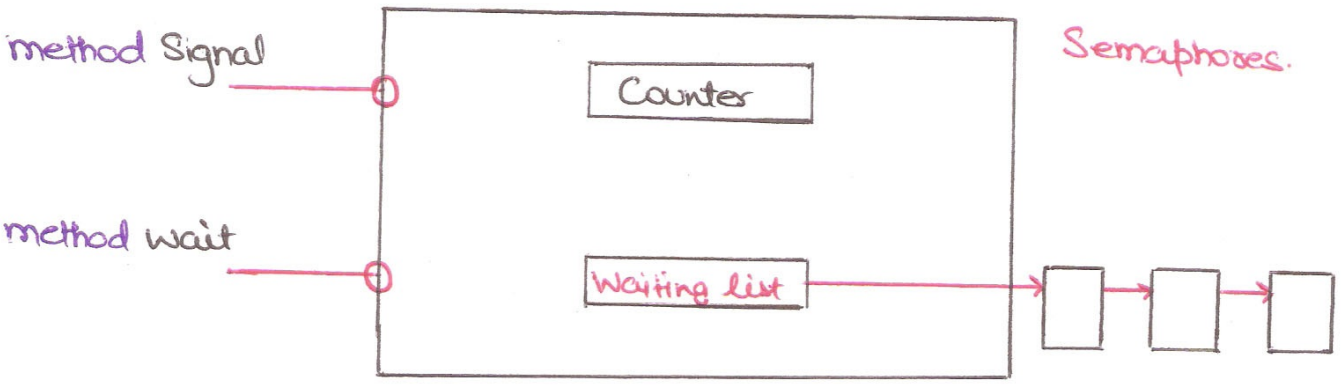
Semaphore is used to protect any resources such as global shared memory that needs to be accessed and updated by many processes simultaneously.

Semaphore acts 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 give 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 processes and two methods (eg. functions) Signal & wait. with integer values.



The Semaphore is accessed by only two indivisible operations known as 'wait' and 'signal' operations which is denoted by P and V.

Example:-

→ Process performs 'wait' operation when tries to enter 'criticalSection'

→ Semaphore allow process to enter Critical section is not being used by any process otherwise deny it

→ The count of the Semaphore is decremented if a process access critical section

→ Initially the count of Semaphore is 1. and if it is accessed then count decremented and become zero.(0).

→ When a process exits the CS, it peoforms the Signal operation which is an exit criterion.

In this way, the solution to CS using Semaphore satisfied the designed protocols.

The Semaphore whose value either 0 or 1 is known as Binary Semaphore. This Concept is basically used in mutual-exclusion.

```
do {
Wait( Semaphore)
{
    Critical
      Section
}
Signal ( Semaphore )
{ ....
  .....
}
```


When there is more processes which want to access more available resources say 3 memories unit then the Semaphore is taken to guard all the three memory locations with value 3. It means that 3 processes at the same time can access the Semaphore. After giving the access to 3 third process the value of count become 0.

This type of Semaphore that takes a value greater than one is known as 'Counting Semaphore'.

MONITORS

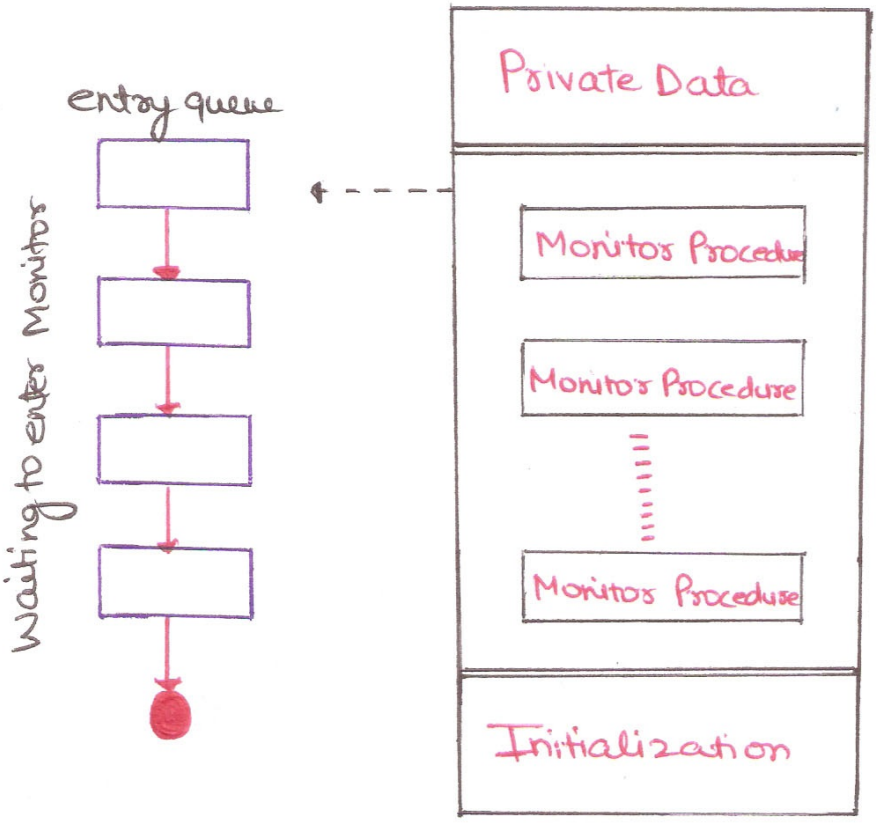
A monitor is a programming Language Construct that Controls access to shared data. OR

Monitor is a module that encapsulates

- 'Shared Data Structure'
- 'procedures' that operate on the shared data structure
- Synchronization between Concurrent procedure invocations

A monitor is same as a class type: like objects of a class are created, the variables of monitor types are defined.

Critical Regions are written as procedures and encapsulated together in a Single Module.



```
type <monitor type> = monitor
  ... data declaration
  ....
  monitor entry <name and its para>
{
  ....
}
```

The Synchronization among the processes is provided through the monitor entry procedures.

** Wait and Signal introduced in Semaphores are also implemented in monitors through the use of 'Condition' Variables.

Condition Variables are different from normal Variables because each of them has an associated queue.

A process calling wait on a particular Condition is put into the queue associated with that Condition Variable.

It means that the process is waiting to enter a CS guarded by the monitor.

A process calling the Signal causes the waiting process in the queue to enter the monitor.

These all variables are declared as:-

Condition <name of Variable>

wait < Condition Variable >

Signal< Condition Variable>

To force a process to exit immediately after the Signal operation, Signal-and-exit monitor is used.

If a process needs to be inside the monitor for some more time after signaling, then Signal-and-Continue monitor is used.

It means that the signaling has been done by the process, but it still maintains a lock on the Semaphore.

Java programming language implements the Signal-and-Continue monitor.

Message Passing

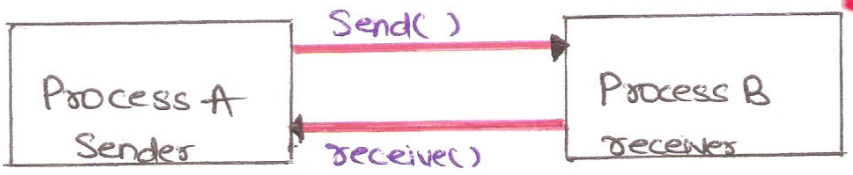
Message Passing refers to means of Communication b/w

- different threads within a process.
- different Processes running on same Node.
- different processes running on different Node.

In this a sender or a source process sends a message to a known receiver or destination process.

message has a predefined structure and message passing uses two System call : Send and Receive

Send (name of destination process, message);
receive (name of source process, message).



In this calls, the Senders and receiver processes address each other by names.

Mode of communication between two processes can take place through two methods

- Direct Addressing
- Indirect Addressing

Direct Addressing :-

In this type, the two processes need to name each other to Communicate.

This becomes easy if they have the same Parent. Example

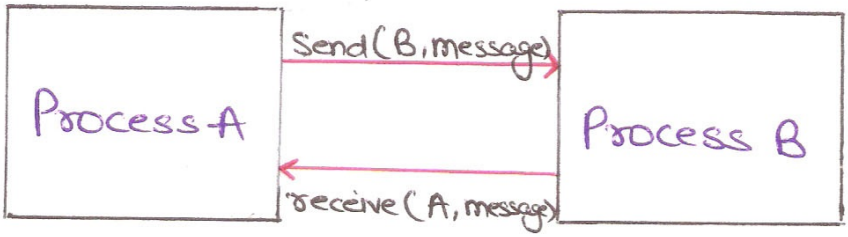
If process A sends a message to process B, then

```
Send ( B, message);
receive ( A, message).
```

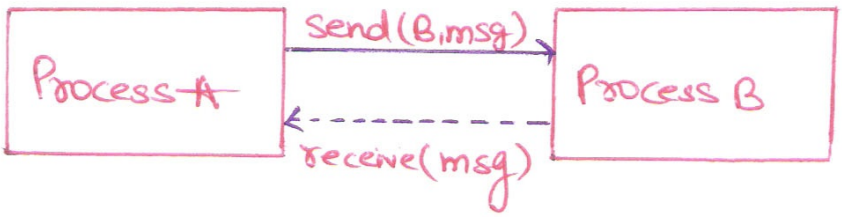
By message passing a link is established b/w A and B.

Here the receiver knows the identity of sender message destination

This type of arrangement in direct Communication is known as Symmetric addressing.



Another type of addressing known as asymmetric addressing where receiver does not know the ID of the sending process in advance.



Asynchronous or Asymetric addressing

Indirect Addressing :-

In this message send and receive from a mailbox. A mailbox can be abstractly viewed as an object into which messages may be placed and from ~~the other~~ which messages may be

removed by processes.

The Sender and receiver processes should share a mailbox to communicate

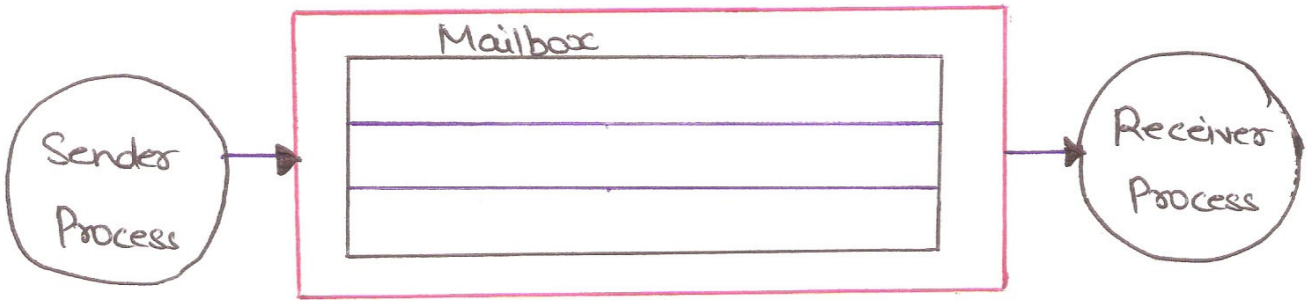
The following types of communication link are possible through mailbox.

→ One to One link :- One Sender wants to communicate with one receiver. Then single link is established.

→ Many to-one-link :- Multiple Senders want to Communicate with single receiver. Eg in Client-Server System, there are many clients processes and one server process. The mailbox is here known as PORT.

→ One-to-many link :- One sender wants to Communicate with multiple receivers, that is, to broadcast a message.

→ Many-to-Many link :- Multiple Senders want to Communicate with multiple receivers.



MAILBOX