Inter Process Communication (IPC)

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.
Let’s discuss an example of communication between processes using shared memory method.

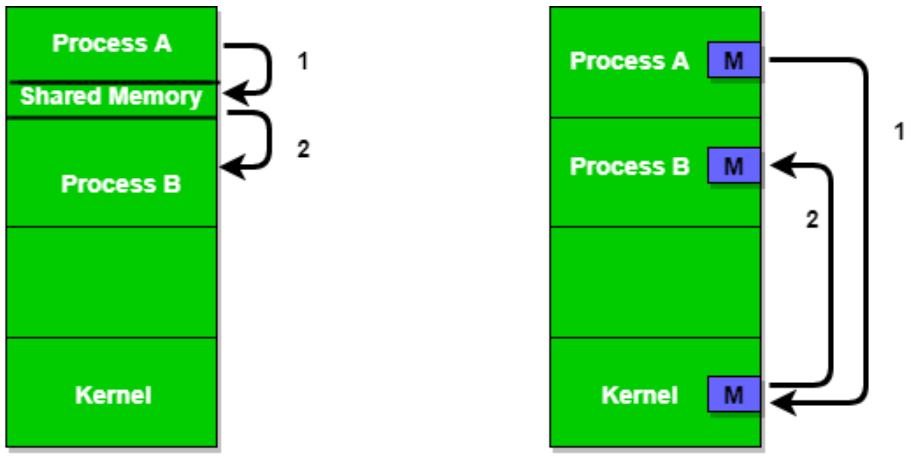


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Example-Producer-Consumer-problem

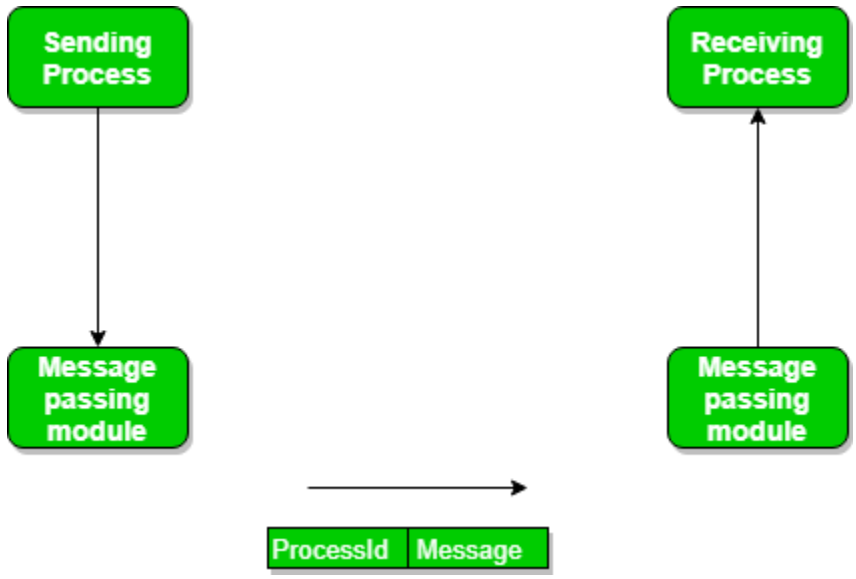
There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it. The pseudo code are given below:

ii) Messaging Passing Method

Now, We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)

- Start exchanging messages using basic primitives.
  We need at least two primitives:
  – **send** (message, destination) or **send**(message)
  – **receive** (message, host) or **receive**(message)



Basic Structure:

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variable.

An IPC facility provides at least the two operations.

Send (message) and **receive(message)**

- Message sent by a process can be of either fixed size or variable size.
- For fixed message physical implementation is straight forward but this restriction makes the task of programming more difficult.
- For variable sized message require more complex physical implementation but the programming task becomes simpler.
- If process P and Q want to communicate, the must send message to and receive messages from each other; a **communication link must exist between them.**

Some basic implementation questions are as follows:

a) How are links established?

b) Can a link be associated with more than two processors

c) How many links can there be between every pair of processes?
d) What is the capacity of a link? That is, does the link have some buffer space? If it does, how much?
e) What is the size of messages? Can the link accommodate variable sized or only fixed-sized messages?

i. Is a link unidirectional (if each process connected to the link can either send or receive, but not both, and each link has at least one receive process connected to it) or bidirectional (if messages are flowing in both directions)

In addition, there are several methods for logically implementing a link and the send/receive operations.

i. Direct or Indirect communication
ii. Symmetric or Asymmetric communication
iii. Automatic or explicit buffering
iv. Send by copy or send by reference
v. Fixed-sized or Variable-sized messages

A standard message can have two parts: **header and body.**
The **header part** is used for storing Message type, destination id, source id, and message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication link. While implementing the link, there are some questions which need to be kept in mind like :

a) How are links established?
b) Can a link be associated with more than two processes?
c) How many links can there be between every pair of communicating processes?
d) What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
e) Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which Every link has a queue associated with it which can be either of zero capacity or of bounded capacity or of unbounded capacity. In zero capacity, sender wait until receiver inform sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the

sender must communicate to receiver explicitly. Implementation of the link depends on the situation, it can be either a Direct communication link or an In-directed communication link.

Direct Communication links are implemented when the processes use specific process identifier for the communication but it is hard to identify the sender ahead of time. **For example: The-Print-Server.**

In-directed Communication is done via a shred mailbox (port), which consists of queue of messages. Sender keeps the message in mailbox and receiver picks them up.

Message Passing through Exchanging the Messages.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so Message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that, for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes But the sender expect acknowledgement from receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution but at the same time, if the message send keep on failing, receiver will have to wait for indefinitely. That is why we also consider the other possibility of message passing. There are basically three most preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing:

The process which wants to communicate must explicitly name the recipient or sender of communication. e.g. **send(p1, message)** means send the message to p1. similarly, **receive(p2, message)** means receive the message from p2.
In this method of communication, the communication link get established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of link. Symmetry and asymmetry between the sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages or only sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.

In Indirect message passing, processes uses mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these link may be unidirectional or bi-directional. Suppose two process want to communicate though Indirect message passing, the required operations are: create a mail box, use this mail box for sending and receiving messages, destroy the mail box. The standard primitives used are : **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem in this mailbox implementation.

Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either forcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox

which can have multiple sender and single receiver. It is used in client/server application (Here server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver process or when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion.

Mutex mailbox is create which is shared by n process. Sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.

Examples of IPC systems

1. Posix : uses shared memory method.
2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

It provides a mechanism to allow processes to communicate and to synchronize their actions. Inter Process communication is best provided by a message system. Message systems can be defined in many different ways.

The **shared memory** and **message system** communication schemes are not mutually exclusive, and could be used simultaneously within a single OS or even a single process.

Processes may also share a mailbox through the process creation facility.

If the mailbox is no longer accessible by any process, the OS reclaim space was used for the mailbox through garbage collection.

- **Buffering -** A link has some capacity that determines the exchange of messages that can reside it temporarily. This property can be viewed as a queue of messages attached to the link. There are three ways that such a queue can implemented.
- **Zero Capacity:** The queue has maximum length zero.Thus, the link cannot have any message waiting in it. In this case, the sender must wait until the recipient receives the message. The two processes must wait be synchronised for the message transfer to take place. This synchronisation is called rendezvous.
- **Bounded Capacity:** The queue has finite length of finite capacity. thus, almost n messages can reside in it. If the queue is not full when a new message is sent., the later is placed n the

queue (either the message is copied or a pointer to the message is kept and the sender can continue execution without waiting. If the link is full, the sender must be delayed until the space is available in the queue.

- **Unbounded Capacity:** The queue has potentially infinite length; thus any number of messages can wait in it. The sender never blocks.
- The zero capacity case is sometimes referred to as a message system with no buffering; the other cases provide automatic buffering.
- If the sender process say P can continue its execution only after the message is received by receiver process say Q. Then, the process P executes the sequence:
- Send (Q, message)
- Receive (Q, message)
- Process Q executes the following sequence:
- Receive (P, message);
- send(P, acknowledgement);

- Such processes are said to communicate asynchronously