Inter Process Communication

Inter process communication(IPC) in OS is way by which **multiple processes** can **communicate** with each other. **Shared memory** in OS, **message queues**, **FIFO** etc are some of the ways to achieve IPC in os.

The process executing in the Operating system can be

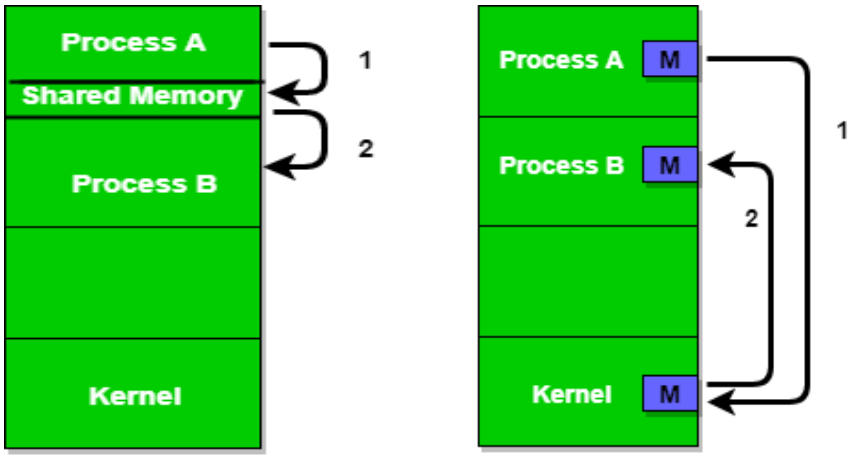
Independent: if it cannot affect OR be affected by other processes. This process does not share data with other processes

Co-operating: if it can affect OR be affected by other processes. This process shares data with other processes. Process co-operation environment is provided for

- Information Sharing
- Computation Speed-up
- Modularity and
- Convenience.

Co-operating process require an Interprocess Communication (IPC). **Synchronization** in Inter Process Communication (IPC) is the process of ensuring that multiple processes are coordinated and do not interfere with each other. This is important because processes can share data and resources, and if they are not synchronized, they can **overwrite** each other's data or cause other problems. The fundamental models of Interprocess communication are

Shared Memory: A region of memory that is shared by the co-operating process is established. Process can exchange information by reading and writing data to the Shared region. Shared memory allows maximum speed and convenience of communication.



Shared Memory **Message Passing**

Message Passing: Communication takes place by means of **messages exchange.** Message passing is useful for exchanging smaller amounts of data, because no conflicts need to be avoided. It is easier to implement than shared memory for intercomputer communication.

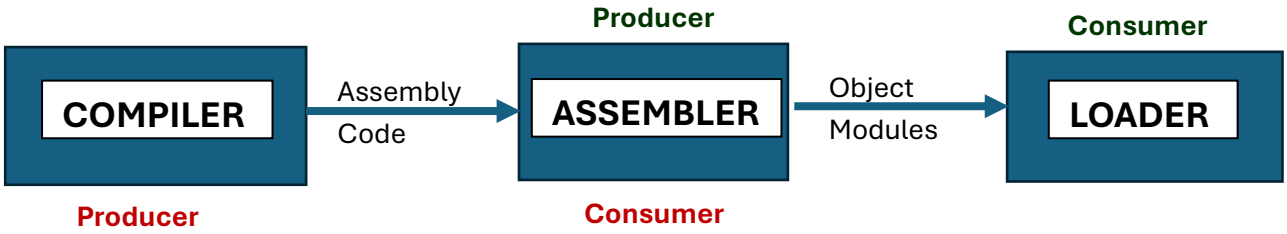
Shared Memory	Message Passing
Faster, because system calls are required only to establish the shared memory regions.	More **time consuming**, as they are implemented using system calls, thus require Kernel intervention.

SHARED MEMORY SYSTEMS

- The communication process **establishes** a region of shared memory. A shared memory region **resides** in the **address space of the process** creating the shared memory segment.
- Other processes sharing the memory segment for communication should attach it to their respective address space. Information exchange happens by reading and writing data in the shared areas.
- The form of data and location is determined by the processes and is not under the control of Operating System.
- The processes are responsible for ensuring that they do not write to the same location simultaneously.
- These processes should share a region of the memory and the code for accessing and manipulating the shared memory should be written explicitly written by the application programmer.

Producer-Consumer Problem:

A **Producer** process generates information that is used by a **Consumer** process.



The above figure illustrates the role of Producer and Consumer. The Producer Consumer problem provides a useful metaphor for the client-server paradigm.

To allow the Producer and consumer to run concurrently, sufficient buffer items must be available to be filled by the producer and emptied by the consumer.

This buffer resides in a location in the memory that is shared by the Producer and consumer. The Producer and Consumer must synchronize so that the Consumer does not try to consume an item that is not yet produced. Two types of buffers are used,

Unbounded buffer: Has no limit on the size of the buffer. The Consumer may have to wait for new items, but the Producer can always produce new ones.

Bounded buffer: Has a fixed buffer size. In this case the **Consumer** must **wait** if the buffer is **empty** and **Producer** must **wait** if the buffer is **Full**.

The shared bounded buffer is implemented as a **circular array** with two logical pointers **in** and **out**. The variable **in** points to the **next free position** and **out** to the **first full position** in the buffer. The buffer is **empty when in==out** and **full when ((in+1) % buffer_size) == out.**

MESSAGE-PASSING SYSTEMS

Message passing provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space. This is advantageous for the distributed environment. Message passing facility provides minimum two operations, sending and receiving.

When messages sent by the process are **fixed size**, **implementation is easy**, but it makes the task of **programming more difficult**. **Variable size** messages require **complex system level implementation** while making **programming tasks simpler**.

The methods for logically implementing a link between the sender and receiver are

- **Direct** or **Indirect** communication
- **Synchronous** or **Asynchronous** communication
- **Automatic** or **Explicit** buffering.

a) Naming: Processes to communicate must have a method to refer to each other.

DIRECT Communication: Each process must explicitly name the recipient or sender of communication. The send( ) and receive( ) primitives are defined as

Send(P, message) – Send a message to Process P.

Receive(Q, message) – Receive a message from process Q.

A communication link in this scheme is established **automatically** between every pair of processes that need to communicate. One link is associated exactly with one pair of processes only.

In **symmetric addressing**, both the sender and receiver process must name the other to communicate. In **Asymmetric addressing**, only the sender names the recipient. The **send(P, message)** and **receive(id, message)** primitives are defined, where in the receive function, the id is set to the process with which communication has taken place.

The disadvantage of these two schemes is that changing the identifier of a process may impose an overhead of finding the references of old identifier, so that they can be modified to the new identifier.

INDIRECT communication: The messages are sent to and received from **mailboxes** or **ports.**

Mailbox is an object, with an unique identification, into which messages can be placed and removed. **POSIX** message queues use an integer value to identify the mailbox.

The send( ) and receive( ) primitives are defined as:

Send(A, message) – send a message to mailbox A

Receive(A, message) – Receive a message from mailbox A

A communication link is established between a pair of processes only **if both members have a shared mailbox.** The link may be associated with **more than two** processes.

Between each pair of Communicating processes there may be multiple links, with each link corresponding to one mailbox.

When a process P1 sends a message and two processes P2 and P3 execute receive() command, the communication will depend on either of the following methods selected:

- Allow a link to be associated with at least 2 processes.
- Allow atmost one process at a time to execute a receive( ) operation.
- Allow the system to arbitrarily select one of the processes to receive.

A mailbox may be owned by a process or an operating system. When a mailbox is **owned by a process:**

- Owner (receives msgs through mailbox) and user (sends msgs to the mailbox) can be distinguished.
- Since each mailbox has a unique owner, confusion can be avoided regarding which process to receive message sent to this mailbox.
- When a process that owns a mailbox terminates, the mailbox disappears. This status will be communicated to the process that subsequently sends a message.

The mailbox **owned** by the **Operating system** has an **existence of its own**. The Operating system provides a mechanism that allows a process to do the following:

- Create / delete a new mailbox.
- Send / Receive messages through the mailbox.

The process that creates the mailbox will be the owner by default and initially will be the only process that will receive messages through this mailbox. Later the ownership can be passed to other processes, through system calls taking care to **handle multiple receivers for each mailbox**.

b) **Synchronization:** Message passing can be either **blocking (Synchronous)** or **non-blocking (asynchronous).** The design options for implementing send( ) and receive( ) are:

- **Blocking send:** The sending process is blocked until the message is received by the receiving process / mailbox.
- **Non-blocking send**: The sending process send the message and resumes operation.
- **Blocking receive**: The receiver blocks until a message is available.
- **Non-blocking receive**: The receiver retrieves either a valid message or a null.

When both send( ) and receive( ) are blocking:

- The **Producer** merely **invokes the send( )** call and waits until the message the message is delivered to either the receiver or the mailbox.
- The **Consumer invokes receive( )**, it blocks until a message is available.

c) **Buffering:** When communication is direct or indirect, messages exchanged by the communicating process reside in temporary queue. These queues are implemented in 3 ways:

- **Zero capacity:** This is a message system with no buffering. The **maximum length** of the queue is **ZERO**, hence the link will not have any waiting messages. The **sender blocks** until the recipient receives the message.
- **Bounded capacity:** The queue has **finite length n**, hence maximum n messages can reside in it. When the **queue is not full**, the new message sent will be placed in the queue and the **sender can continue** execution without waiting. If the **link is full**, the **sender must be blocked** until space is available in the queue.
- **Unbounded capacity:** The **queue length is infinite**, thus any number of messages can wait. The **sender never blocks**.