Inter Process Communication (IPC)

A process can be of two types:

- 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, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with 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 through both:

1. Shared Memory
2. Message passing

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

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the 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 another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs 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 another process as well as for delivering any specific information to other processes.
Let’s discuss an example of communication between processes using the shared memory method.

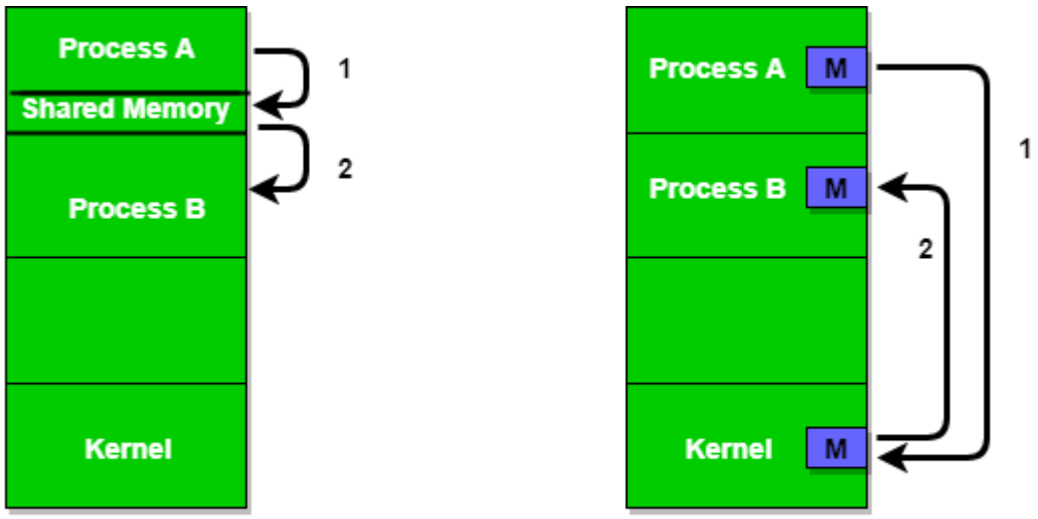


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

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

Shared Data between the two Processes

- C

```
#define buff_max 25
#define mod %

  struct item{

    // different member of the produced data
    // or consumed data
```

```
    ---------
}

// An array is needed for holding the items.
// This is the shared place which will be
// access by both process
// item shared_buff [ buff_max ];

// Two variables which will keep track of
// the indexes of the items produced by producer
// and consumer The free index points to
// the next free index. The full index points to
// the first full index.
int free_index = 0;
int full_index = 0;
```

Producer Process Code

- C

```
item nextProduced;

   while(1){

      // check if there is no space
      // for production.
      // if so keep waiting.
      while((free_index+1) mod buff_max == full_index);

      shared_buff[free_index] = nextProduced;
      free_index = (free_index + 1) mod buff_max;
   }
```

Consumer Process Code

- C

```
item nextConsumed;

   while(1){

      // check if there is an available
      // item  for consumption.
      // if not keep on waiting for
```

```
    // get them produced.
    while((free_index == full_index);

    nextConsumed = shared_buff[full_index];
    full_index = (full_index + 1) mod buff_max;
  }
```

In the above code, the Producer will start producing again when the (free_index+1) mod buff max will be free because if it is not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index point to the same index, this implies that there are no items to consume.

Overall C++ Implementation:

- C++

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

#define buff_max 25
#define mod %

struct item {
  // different member of the produced data
  // or consumed data
  // ---------
};

// An array is needed for holding the items.
// This is the shared place which will be
// access by both process
// item shared_buff[buff_max];

// Two variables which will keep track of
// the indexes of the items produced by producer
// and consumer The free index points to
// the next free index. The full index points to
// the first full index.
std::atomic<int> free_index(0);
std::atomic<int> full_index(0);
std::mutex mtx;

void producer() {
```

```
  item new_item;
  while (true) {
    // Produce the item
    // ...
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    // Add the item to the buffer
    while (((free_index + 1) mod buff_max) == full_index) {
      // Buffer is full, wait for consumer
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    mtx.lock();
    // Add the item to the buffer
    // shared_buff[free_index] = new_item;
    free_index = (free_index + 1) mod buff_max;
    mtx.unlock();
  }
}

void consumer() {
  item consumed_item;
  while (true) {
    while (free_index == full_index) {
      // Buffer is empty, wait for producer
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    mtx.lock();
    // Consume the item from the buffer
    // consumed_item = shared_buff[full_index];
    full_index = (full_index + 1) mod buff_max;
    mtx.unlock();
    // Consume the item
    // ...
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}

int main() {
  // Create producer and consumer threads
  std::vector<std::thread> threads;
  threads.emplace_back(producer);
  threads.emplace_back(consumer);

  // Wait for threads to finish
  for (auto& thread : threads) {
    thread.join();
  }
```

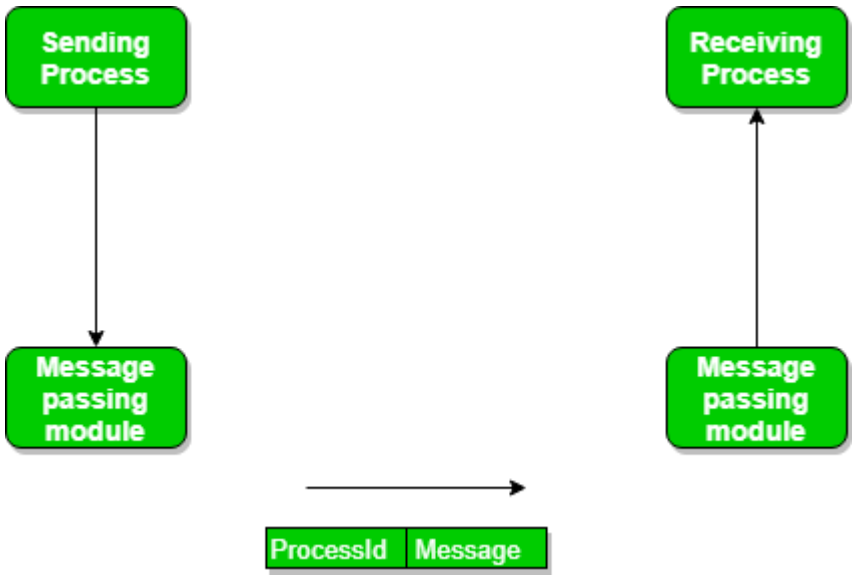
```
  return 0;
}
```

Note that the atomic class is used to make sure that the shared variables free_index and full_index are updated atomically. The mutex is used to protect the critical section where the shared buffer is accessed. The sleep_for function is used to simulate the production and consumption of items.

ii) Messaging Passing Method

Now, We will start our discussion of 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 follows:

- 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)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing message type, destination id, source id, 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 links. While implementing the link, there are some questions that need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. 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 of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the 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 with the 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 a specific process identifier for the communication, but it is hard to identify the sender ahead of time.
For example the print server.
In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the 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. However, the sender expects acknowledgment from the 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. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three 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 the communication.
e.g. **send(p1, message)** means send the message to p1.
Similarly, **receive(p2, message)** means to receive the message from p2.
In this method of communication, the communication link gets 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 links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the 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 use 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 links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. 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 with 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 enforcing 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 that can have multiple senders and a single receiver. It is used in client/server applications (in this case the 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 processor 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 created which is shared by n process. The 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.
Now, let's discuss the Producer-Consumer problem using the message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Producer Code

- C

```
void Producer(void){

    int item;
    Message m;

    while(1){

       receive(Consumer, &m);
       item = produce();
       build_message(&m , item ) ;
       send(Consumer, &m);
    }
}
```

Consumer Code

- C

```
void Consumer(void){

    int item;
    Message m;

    while(1){

       receive(Producer, &m);
       item = extracted_item();
       send(Producer, &m);
       consume_item(item);
    }
}
```

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

Communication in client/server Architecture:

There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

The above three methods will be discussed in later articles as all of them are quite conceptual and deserve their own separate articles.

References:

1. Operating System Concepts by Galvin et al.
2. Lecture notes/ppt of Ariel J. Frank, Bar-Ilan University

Inter-process communication (IPC) is the mechanism through which processes or threads can communicate and exchange data with each other on a computer or across a network. IPC is an important aspect of modern operating systems, as it enables different processes to work together and share resources, leading to increased efficiency and flexibility.

Advantages of IPC:

1. Enables processes to communicate with each other and share resources, leading to increased efficiency and flexibility.
2. Facilitates coordination between multiple processes, leading to better overall system performance.
3. Allows for the creation of distributed systems that can span multiple computers or networks.
4. Can be used to implement various synchronization and communication protocols, such as semaphores, pipes, and sockets.

Disadvantages of IPC:

1. Increases system complexity, making it harder to design, implement, and debug.
2. Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.
3. Requires careful management of system resources, such as memory and CPU time, to ensure that IPC operations do not degrade overall system performance.
   Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.
4. Overall, the advantages of IPC outweigh the disadvantages, as it is a necessary mechanism for modern operating systems and enables processes to work together and share resources in a flexible and efficient manner. However, care must be taken to design and implement IPC systems

carefully, in order to avoid potential security vulnerabilities and performance issues.