**INTERPROCESS COMMUNICATION (IPC)**

**Inter-Process Communication**or**IPC** refers to a mechanism in OS that allow multiple processes to communicate with each other. Processes executing concurrently in the operating system may be either **independent processes** or **cooperating processes**. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

**Need of Interprocess communication:**

There are several reasons for providing an environment that allows process cooperation:

 • **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

•**Modularity**. We may want to construct or design the system in a modular fashion, dividing the system functions into separate processes or threads.

 • **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter process communication:

1. Shared memory.
2. Message passing.

**1)Shared Memory:**

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

A shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.

The form of the data and the location are determined by these processes and are not under the operating system’s control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

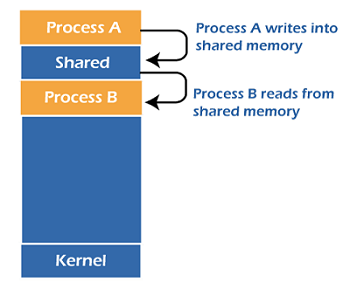


Fig: Shared Memory

**Example:** Producer–consumer problem

Producer–consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

The producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

**Ex:** The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in ==out; the buffer is full when ((in + 1) % BUFFER SIZE) == out. The producer process has a local variable next produced in which the new item to be produced is stored. The consumer process has a local variable next consumed in which the item to be consumed is stored.

**The producer process using shared memory.**

item next produced;

while (true) {

/\* produce an item in next produced \*/

while (((in + 1) % BUFFER SIZE) == out); /\* do nothing \*/

buffer[in] = next produced;

in = (in + 1) % BUFFER SIZE;

}

**The consumer process using shared memory.**

item next consumed;

while (true) {

while (in == out); /\* do nothing \*/

next consumed = buffer[out];

out = (out + 1) % BUFFER SIZE;

/\* consume the item in next consumed \*/

}

**2)Message Passing:**

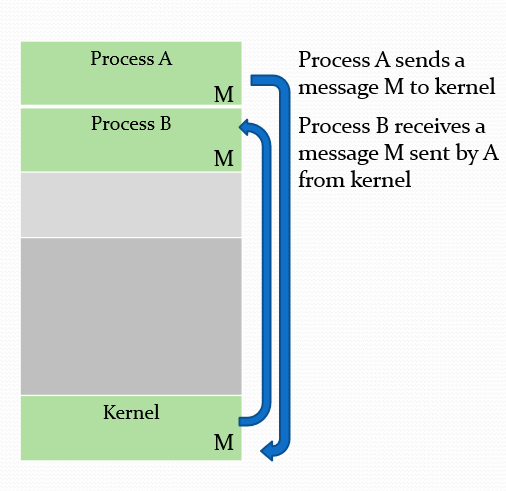
Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

1) send(message)

2) receive(message)



Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system level implementation, but the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send()/receive() operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

There are several issues related with features like:

i)Naming.

ii)Synchronization.

iii)Buffering.

**I)Naming:**

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, 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 has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other’s identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate.

A **variant** of this scheme employs **asymmetry in addressing**. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send () and receive () primitives are defined as follows:

• send (P, message)—Send a message to process P.

• receive (id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

The **disadvantage** in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

With **indirect communication**, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive()

primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links

may exist, with each link corresponding to one mailbox.

A mailbox may be owned either by a process or by the operating system. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

A mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

**ii)Synchronization:**

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.

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

**iii)Buffering:**

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

**• Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**• Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The link’s capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

**• Unbounded capacity:** The queue’s length is potentially infinite; 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 are referred to as systems with automatic buffering.