MESSAGE-PASSING SYSTEMS VERSUS SHARED MEMORY SYSTEMS

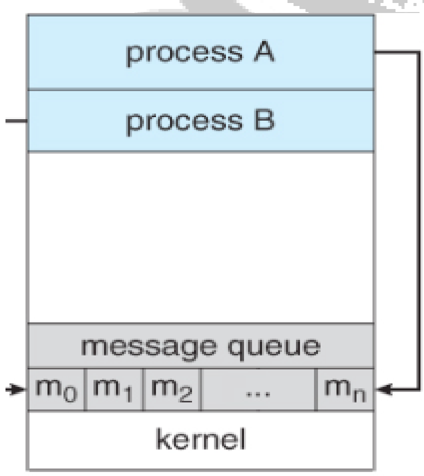
Communication among processors takes place via shared data variables, and control variables for synchronization among the processors**.** The communications between the tasks in multiprocessor systems take place through two main modes:

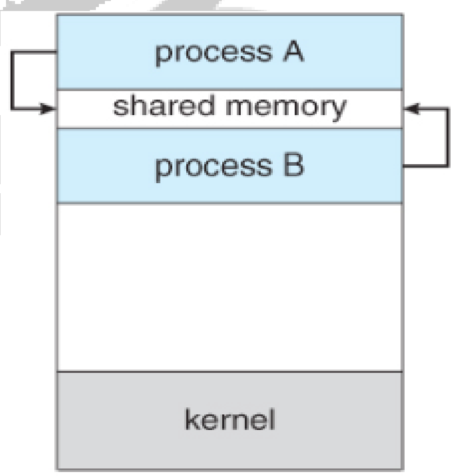
Message passing systems:

- This allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for inter process communication and are used by most operating systems.

Shared memory systems:

- The shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other.
- Communication among processors takes place through shared data variables, and control variables for synchronization among the processors.
- Semaphores and monitors are common synchronization mechanisms on shared memory systems.
- When shared memory model is implemented in a distributed environment, it is termed as **distributed shared memory.**





a) Message Passing Model **b) Shared Memory Model**

Fig : Inter-process communication models

Differences between message passing and shared memory models

Message Passing	Distributed Shared Memory
Services Offered: Variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process.	The processes share variables directly, so no marshalling and unmarshalling. Shared variables can be named, stored and accessed in DSM.
Processes can communicate with other processes. They can be protected from one another by having private address spaces.	Here, a process does not have private address space. So one process can alter the execution of other.
This technique can be used in heterogeneous computers.	This cannot be used to heterogeneous computers.
Synchronization between processes is through message passing primitives.	Synchronization is through locks and semaphores.
Processes communicating via message passing must execute at the same time.	Processes communicating through DSM may execute with non-overlapping lifetimes.
Efficiency: All remote data accesses are explicit and therefore the programmer is always aware of whether a particular operation is in-process or involves the expense of communication.	Any particular read or update may or may not involve communication by the underlying runtime support.

Emulating message-passing on a shared memory system (MP → SM)

- The shared memory system can be made to act as message passing system. The shared address space can be partitioned into disjoint parts, one part being assigned to each processor.
- Send and receive operations care implemented by writing to and reading from the destination/sender processor's address space. The read and write operations are synchronized.
- Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes.

Emulating shared memory on a message-passing system (SM → MP)

- This is also implemented through read and write operations. Each shared location can be modeled as a separate process. Write to a shared location is emulated by sending an update message tothe corresponding owner process and read operation to a shared location is emulated by sending a query message to the owner process.
- This emulation is expensive as the processes have to gain access to other process memory location. The latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication.

PRIMITIVES FOR DISTRIBUTED COMMUNICATION

Blocking / Non blocking / Synchronous / Asynchronous

- Message send and message receive communication primitives are done through Send() and Receive(), respectively.
- A Send primitive has two parameters: *the destination*, and *the buffer* in the user space that holds the data to be sent.
- The Receive primitive also has two parameters: *the source* from which the data is to be received and the *user buffer* into which the data is to be received.

There are two ways of sending data when the Send primitive is called:

- **Buffered:** The standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. For the Receive primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.
- **Unbuffered**: The data gets copied directly from the user buffer onto the network.

Blocking primitives

- The primitive commands wait for the message to be delivered. The execution of the processes is blocked.
- The sending process must wait after a send until an acknowledgement is made by the receiver.
- The receiving process must wait for the expected message from the sending process

- The receipt is determined by polling common buffer or interrupt
- This is a form of synchronization or synchronous communication.
- A primitive is blocking if control returns to the invoking process after the processing for the primitive completes.

Non Blocking primitives

- If send is nonblocking, it returns control to the caller immediately, before the message is sent.
- The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- This is a form of asynchronous communication.
- A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
- For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer.
- For anon-blocking Receive, control returns to the process even before the datamay have arrived from the sender.

Synchronous

- A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other.
- The processing for the Send primitive completes only after the invoking processor learns
- that the other corresponding Receive primitive has also been invoked andthat the receive operation has been completed.
- The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.

Asynchronous

- A Send primitive is said to be asynchronous, if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.
- It does not make sense to define asynchronous Receive primitives.
- Implementing non -blocking operations are tricky.

- For non-blocking primitives, a return parameter on the primitive call returns a system-generated **handle** which can be later used to check the status of completion of the call.
- The process can check for the completion:
    - checking if the handle has been flagged or posted
    - issue a Wait with a list of handles as parameters: usually blocks until one of the parameter handles is posted.

The send and receive primitives can be implemented in four modes:

- Blocking synchronous
- Non- blocking synchronous
- Blocking asynchronous
- Non- blocking asynchronous

Four modes of send operation

Blocking synchronous Send:

- The data gets copied from the user buffer to the kernel buffer and is then sent over the network.
- After the data is copied to the receiver's system buffer and a Receive call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the Send operation and completes the Send.

Non-blocking synchronous Send:

- Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated.
- A parameter in the non-blocking call also gets set with the handle of a location that the user process can later check for the completion of the synchronous send operation.
- The location gets posted after an acknowledgement returns from the receiver.
- The user process can keep checking for the completion of the non-blocking synchronous Send by testing the returned handle, or it can invoke the blocking Wait operation on the returned handle

Blocking asynchronous Send:

- The user process that invokes the Send is blocked until the data is copied from the user's buffer to the kernel buffer.

Non-blocking asynchronous Send:

- The user process that invokes the Send is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated.
- Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later using the Wait operation for the completion of the asynchronous Send.

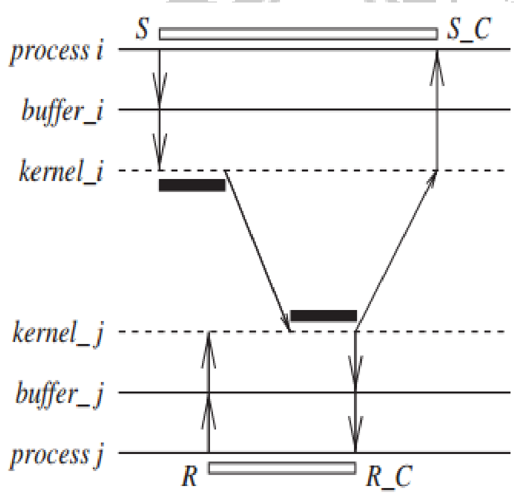


Fig a) Blocking synchronous send and blocking receive

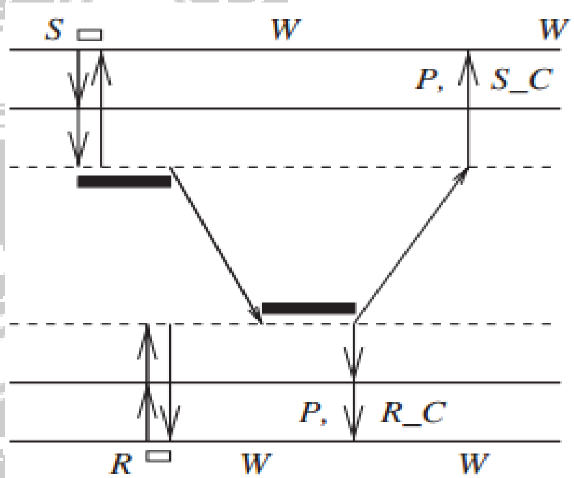


Fig b) Non-blocking synchronous send and blocking receive

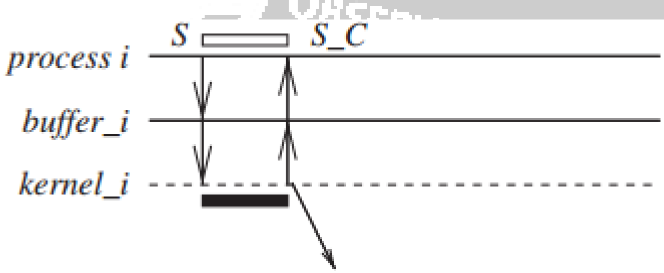


Fig c) Blocking asynchronous send

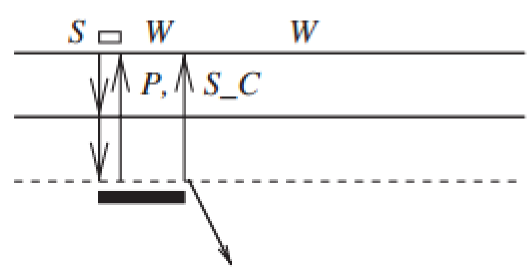


Fig d) Non-blocking asynchronous send

- The asynchronous Send completes when the data has been copied out of the user's buffer. The checking for the completion may be necessary if the user wants to reuse the buffer from which the data was sent.

Modes of receive operation

Blocking Receive:

The Receive call blocks until the data expected arrives and is written in the specified user buffer. Then control is returned to the user process.

Non-blocking Receive:

- The Receive call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking Receive operation.
- This location gets posted by the kernel after the expected data arrives and is copied to the user-specified buffer. The user process can check for the completion of the non-blocking Receive by invoking the Wait operation on the returned handle.

Processor Synchrony

Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.

Since distributed systems do not follow a common clock, this abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Libraries and standards

There exists a wide range of primitives for message-passing. The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community

- **Message Passing Interface (MPI):** This is a standardized and portable message-passing system to function on a wide variety of parallel computers. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- The primary goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.
- **Parallel Virtual Machine (PVM):** It is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.
- **Remote Procedure Call (RPC):** The Remote Procedure Call (RPC) is a common model of request reply protocol. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.
- **Remote Method Invocation (RMI):** RMI (Remote Method Invocation) is a way that a programmer can write object-oriented programming in which objects on different computers can interact in a distributed network. It is a set of protocols being developed by Sun's JavaSoft division that enables Java objects to communicate remotely with other Java objects.
- **Remote Procedure Call (RPC):** RPC is a powerful technique for constructing distributed, client-server based applications. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. RPC makes the client/server model of computing more powerful and easier to program.

Differences between RMI and RPC

RMI	RPC
RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke.	RPC is not object oriented and does not deal with objects. Rather, it calls specific subroutines that are already established

With RPC looks like a local call. RPC handles the complexities involved with passing the call from the local to the remote computer.	RMI handles the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called.

The commonalities between RMI and RPC are as follows:

- ✓ They both support programming with interfaces.
- ✓ They are constructed on top of request-reply protocols.
- ✓ They both offer a similar level of transparency.

- **Common Object Request Broker Architecture (CORBA):** CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects. The data representation is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.