Inter Process Communication

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.

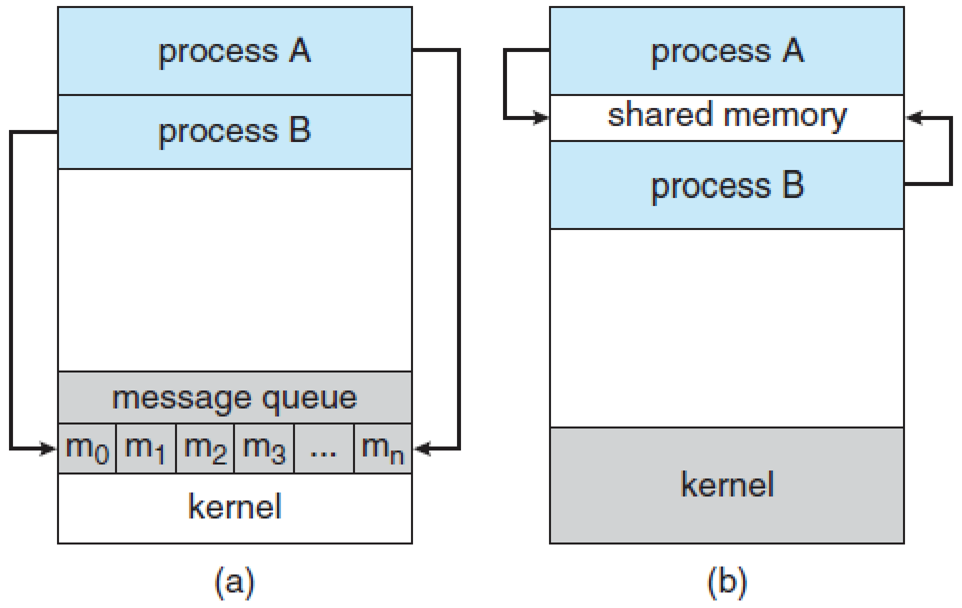
Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity.

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.

- 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 cores.
- **Modularity**. We may want to construct 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, listening to music, and compiling in parallel.

- Processes can communicate with each other using these two ways:
- Shared Memory
- Message passing

- The Figure 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.



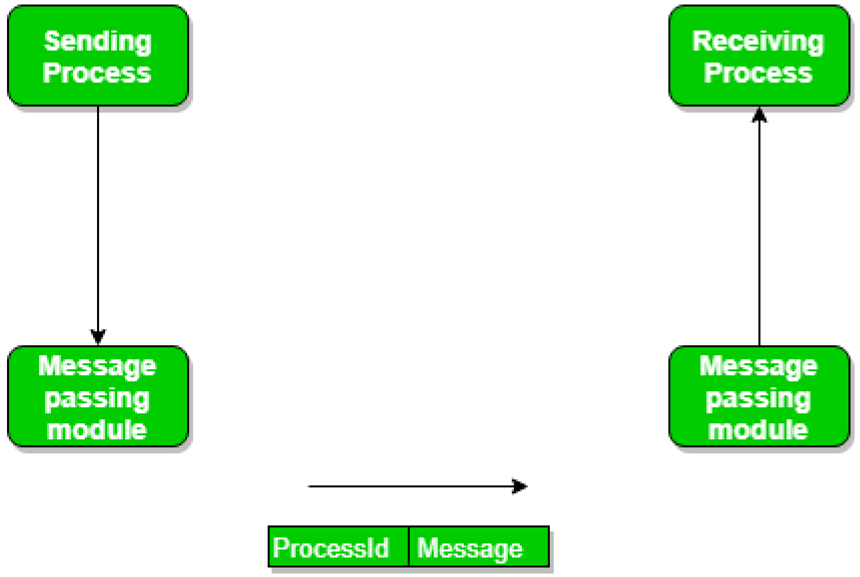
Communications models. (a) Message passing. (b) Shared memory.

Shared Memory

- 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 process 1 and process2 are executing simultaneously and they share some resources or use some information from other process, process 1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process 2 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.

Message Passing

- In this method, processes communicate with each other without using any kind of 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, destinaion) or **send**(message)
- **receive**(message, host) or **receive**(message)


Sending Process
Message passing module
ProcessId
Message
Receiving Process
Message passing module

- The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for 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 queue

- A message queue is an inter process communication mechanism which uses an n byte memory block as queue.
- A queue is a common form of message passing
- The queue uses a FIFO discipline and holds records that represent messages
- Every OS provides message queue IPC functions.

- **Features:**

1. OS provides for inserting and deleting the message-pointers or messages.
2. Each queue for a message need initialization (creation) before using the functions in the scheduler for the message queue

3. There may be a provision for multiple queues for the multiple types or destinations of messages. Each queue may have an ID.

4. Each queue either has a user definable size (upper limit for number of bytes) or a fixed predefined size assigned by the scheduler

5. When the queue becomes full, there may be a need for error handling and user codes for blocking the tasks

- **<u>Queue IPC functions</u>**

- Figure (a) shows the memory blocks at OS for inserting deleting and other functions.
- Figure (b) shows the functions for the queue in the OS.
- Figure (c) shows a queue message block with the messages or message pointers.
- Two pointers *QHEAD and *QTAIL are for queue head and tail memory locations

Figure (a) shows the memory blocks at OS for inserting deleting and other functions.

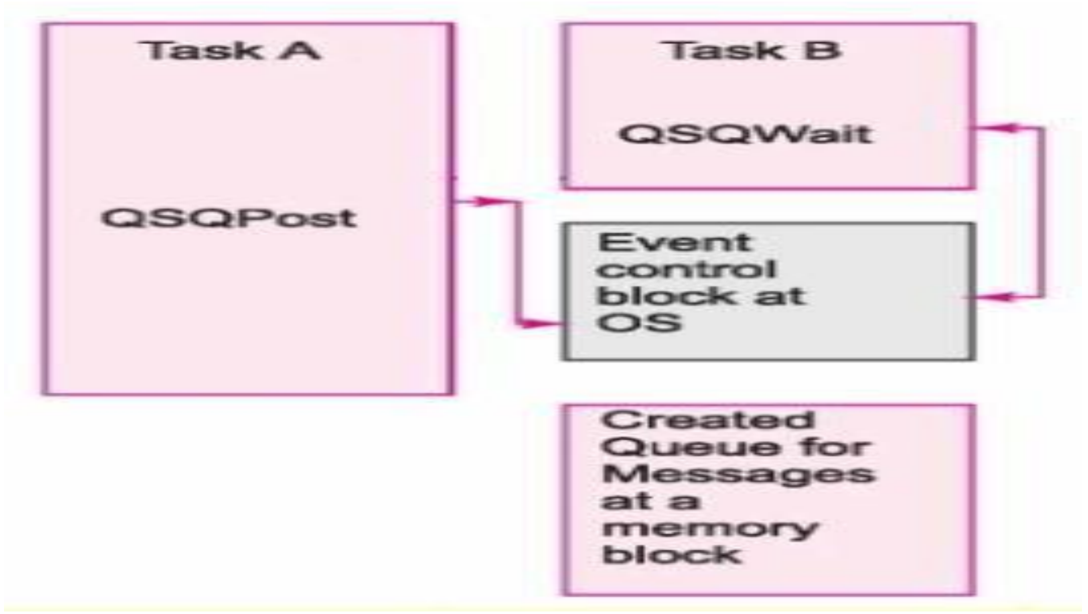

Task A
QSQPost
Task B
QSQWait
Event control block at OS
Created Queue for Messages at a memory block

Figure (b) shows the functions for the queue in the OS.

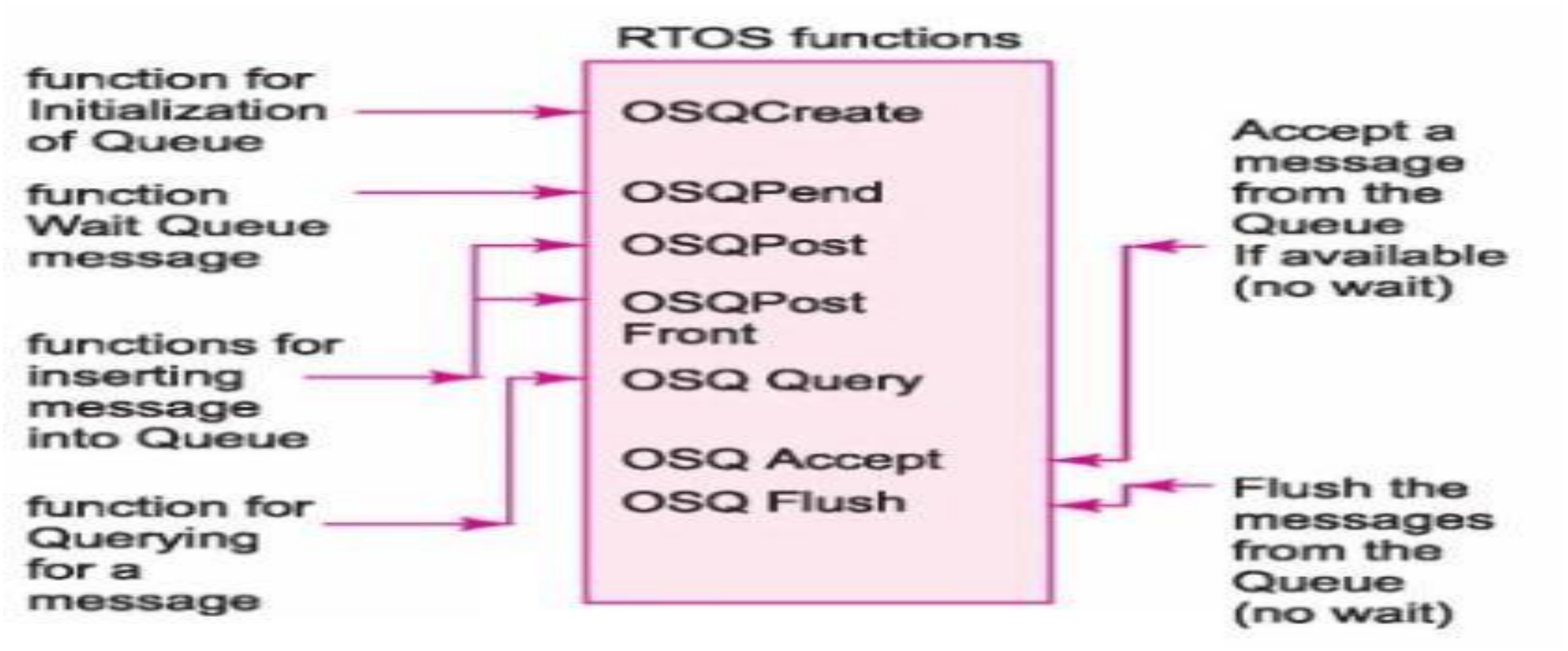
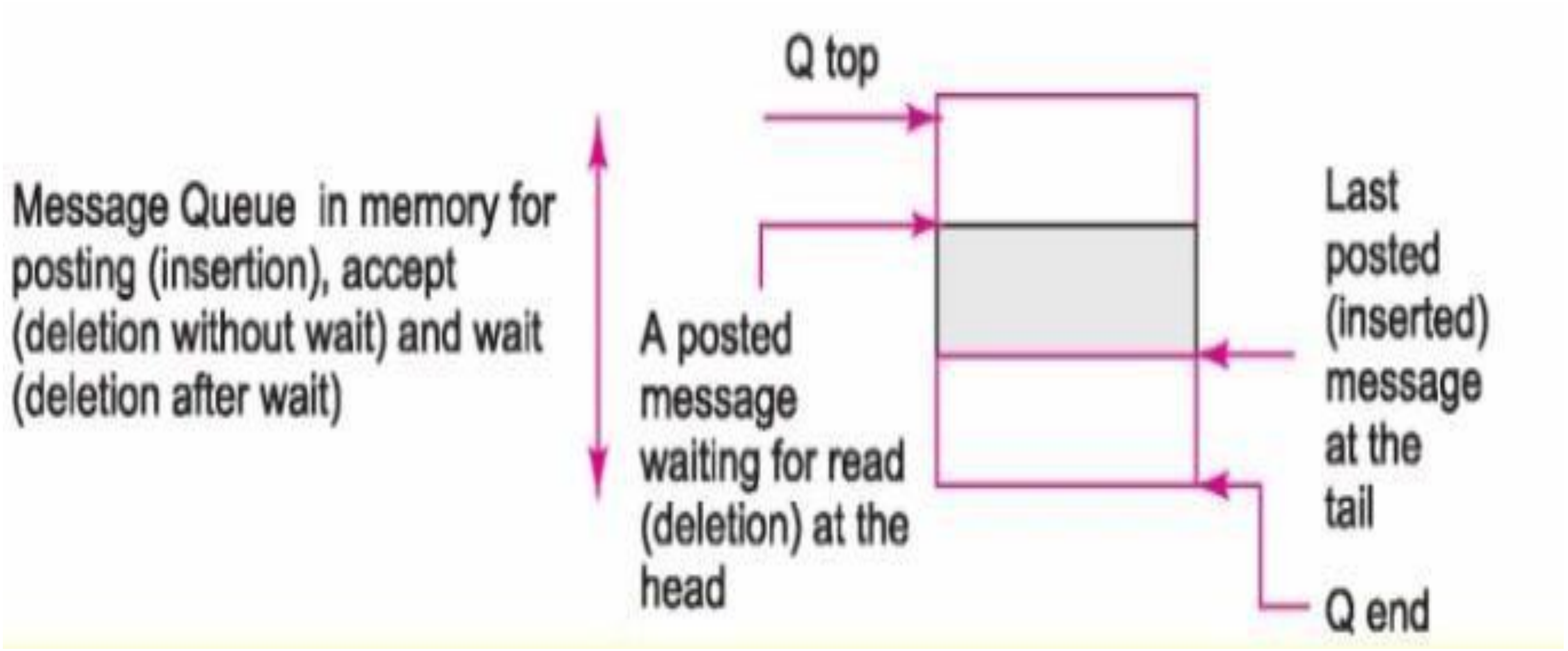

RTOS functions
function for Initialization of Queue
OSQCreate
function Wait Queue message
OSQPend
OSQPost
OSQPost Front
functions for inserting message into Queue
OSQ Query
function for Querying for a message
OSQ Accept
OSQ Flush
Accept a message from the Queue If available (no wait)
Flush the messages from the Queue (no wait)

Figure (c) shows a queue message block with the messages or message pointers. Two pointers *QHEAD and *QTAIL are for queue head and tail memory locations


Q top
Message Queue in memory for posting (insertion), accept (deletion without wait) and wait (deletion after wait)
A posted message waiting for read (deletion) at the head
Last posted (inserted) message at the tail
Q end

- **OSQCreate**

This function is used to create a queue and initialize the queue.

- **OSQPost**

This function is used to post a message to the message block as per the queue tail pointer, *QTAIL.

- **OSQPend**

This function is used to wait for a queue message at the queue and reads and deletes that when received.

- **OSQAccept**

This function is used to delete the present queue head after checking its presence and after the read the queue –front pointer increments

- **OSQ Flush**

This functions read queue from front to back and deletes the queue block as it is not needed later after the flush the queue front and back points to QTOP

OSQ Query

- This function is used to query the queue message-block when read and but the message is not deleted. The function returns pointer to the message queue

- *QHEAD if there are the messages in the queue or else NULL. It return a pointer
- to data structure of the queue data structure which has *QHEAD, number of queued messages, size of the queue and. table of tasks waiting for the messages from the queue.

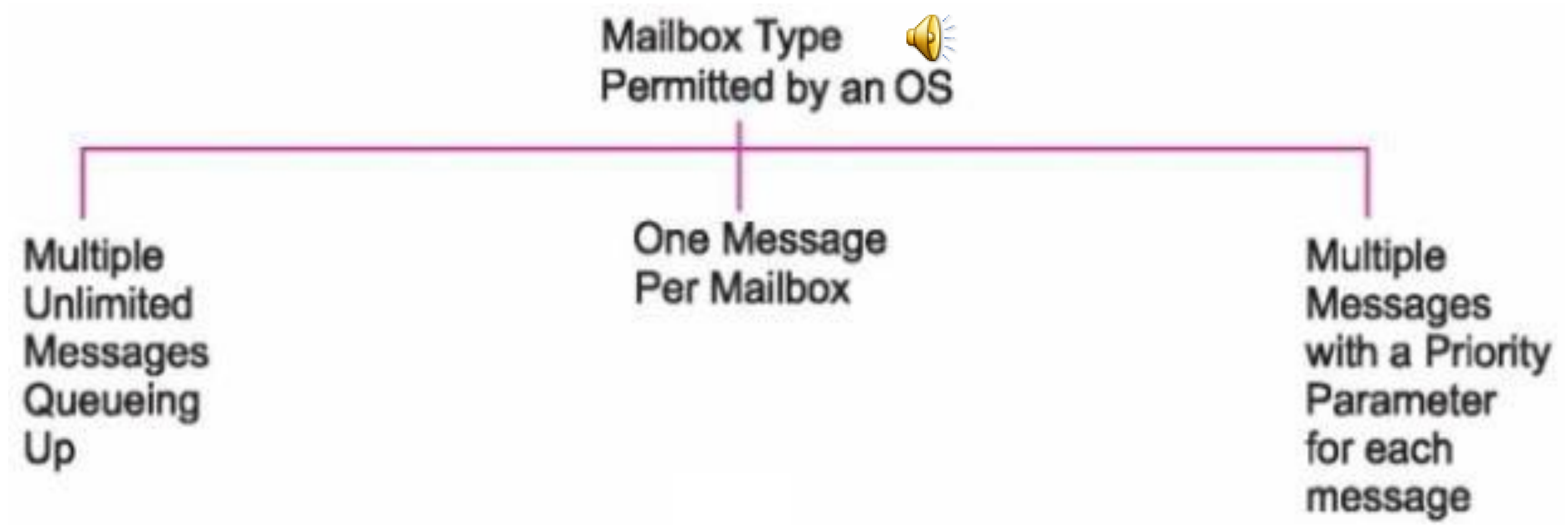
- **OSQPostFront**

➢ This function is used to send a message as per the queue front pointer, *QHEAD. Use of this function is made in the following situations. A message is urgent or is of higher priority than all the previously posted message into the queue.

MailBox

- **Features**
- A mailbox can be viewed abstractly as an object into which messages can be placed by process and from which messages can be removed
- Each mailbox has an unique identification
- A mailbox message may include a header to identify the message type specification
- A mail sender is a task that sends the message pointer to a created mailbox
- There may be a provision for multiple mailboxes for the multiple types or destinations of messages
- Each mailbox usually has one message pointer only, which can point to message.

- **Types:**
  There are three types of mailboxes as given below


Mailbox Type Permitted by an OS
Multiple Unlimited Messages Queueing Up
One Message Per Mailbox
Multiple Messages with a Priority Parameter for each message

- One type is only one message per mailbox is available
- Another type is mailbox with provision for multiple messages or message pointers
- Third one is OS can provide multiple mailbox messages with each message having a priority parameter. The read (deletion) can then only be on priority basis in case mailbox has multiple messages
- **Mailbox Functions**
- **OS-MBOX Create** creates a box and initializes the mailbox contents with a NULL pointer
- **OS-MBOX Write (Post)** sends a message to the box
- **OS-MBOX Wait** waits for a mailbox message which is read when received
- **OS-MBOX Read (Pend)** reads a message from the box
- **OS-MBOX Query** queries the mailbox when read and not needed later

Pipe

- Pipe is a device used for the inter process communication.
- A message-pipe is a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks
- Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process.
- Writing and reading from a pipe is like using a C command *fwrite* with a file name to write into a named file, and C command *fread* with a file name to read into a named file

- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a ***“virtual file”***.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.

- **Write and read using Pipe**
- One task using the function *fwrite* in a set of tasks can write to a pipe at the back pointer address, *pBACK.
- One task using the function *fread* in a set of tasks can read (delete) from a pipe at the front pointer address, *pFRONT.
- In a pipe there may be no fixed number of bytes per message but there is end pointer.
- A pipe can therefore be limited and have a variable number of bytes per message between the initial and final pointers.
- Pipe is unidirectional. One thread or task inserts into it and other one deletes from it.

Figure (a) shows the write and read the pipe using device drivers.

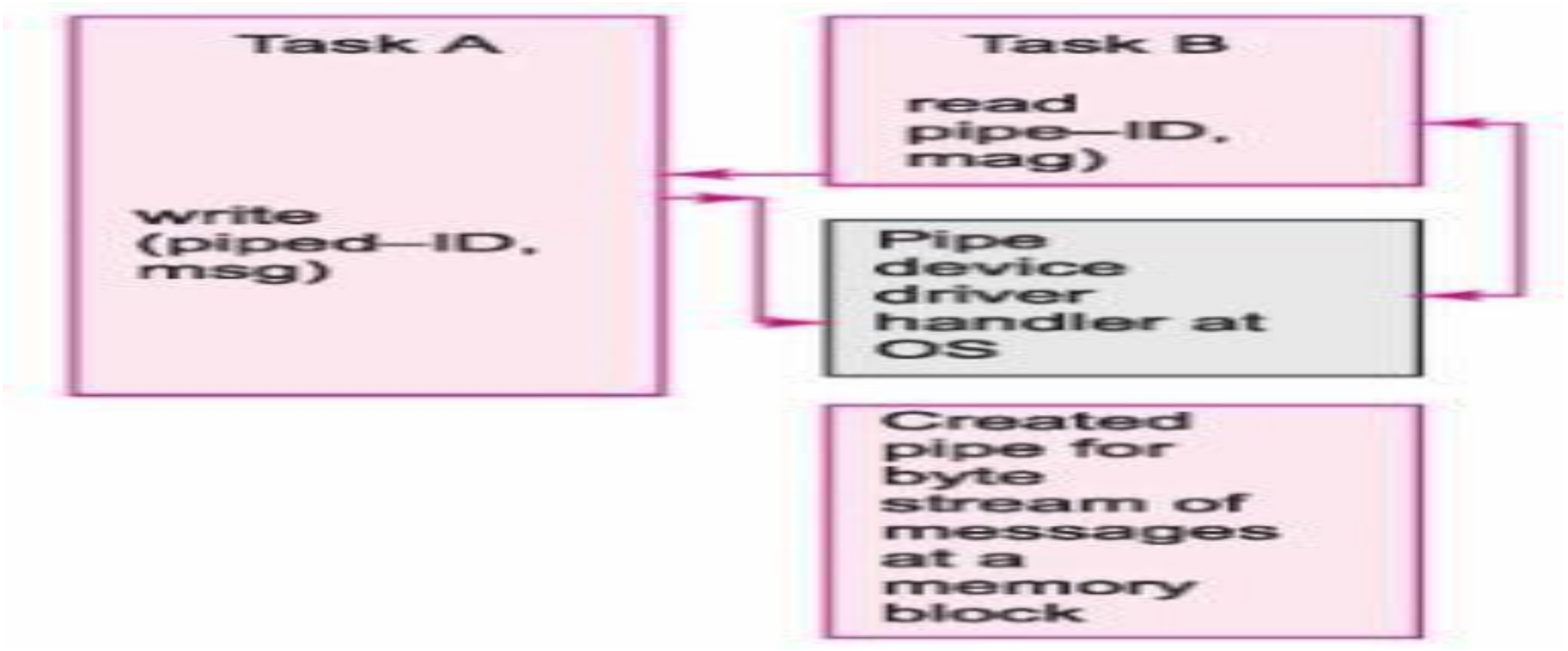

Task A
write (piped—ID. msg)
Task B
read pipe—ID. mag)
Pipe device driver handler at OS
Created pipe for byte stream of messages at a memory block

Figure (b) shows the functions for the pipe in the OS.

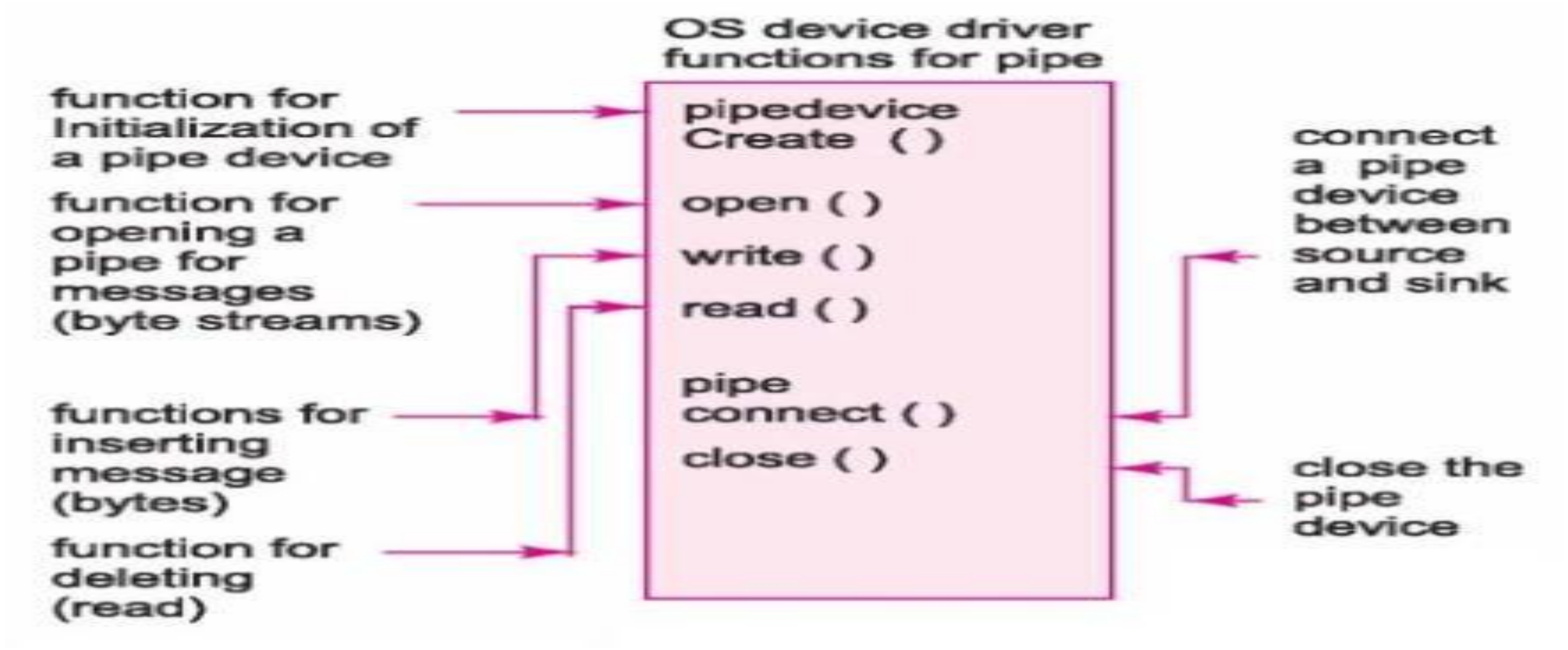
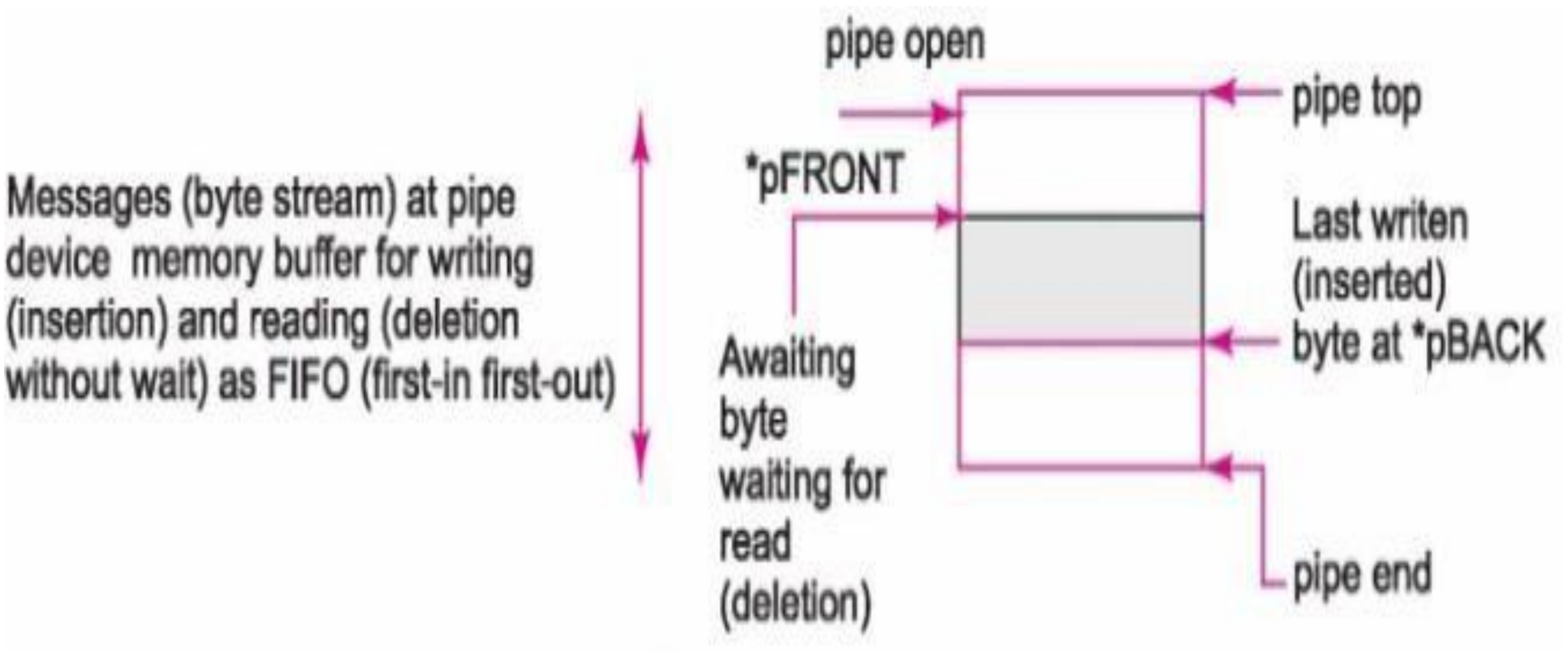
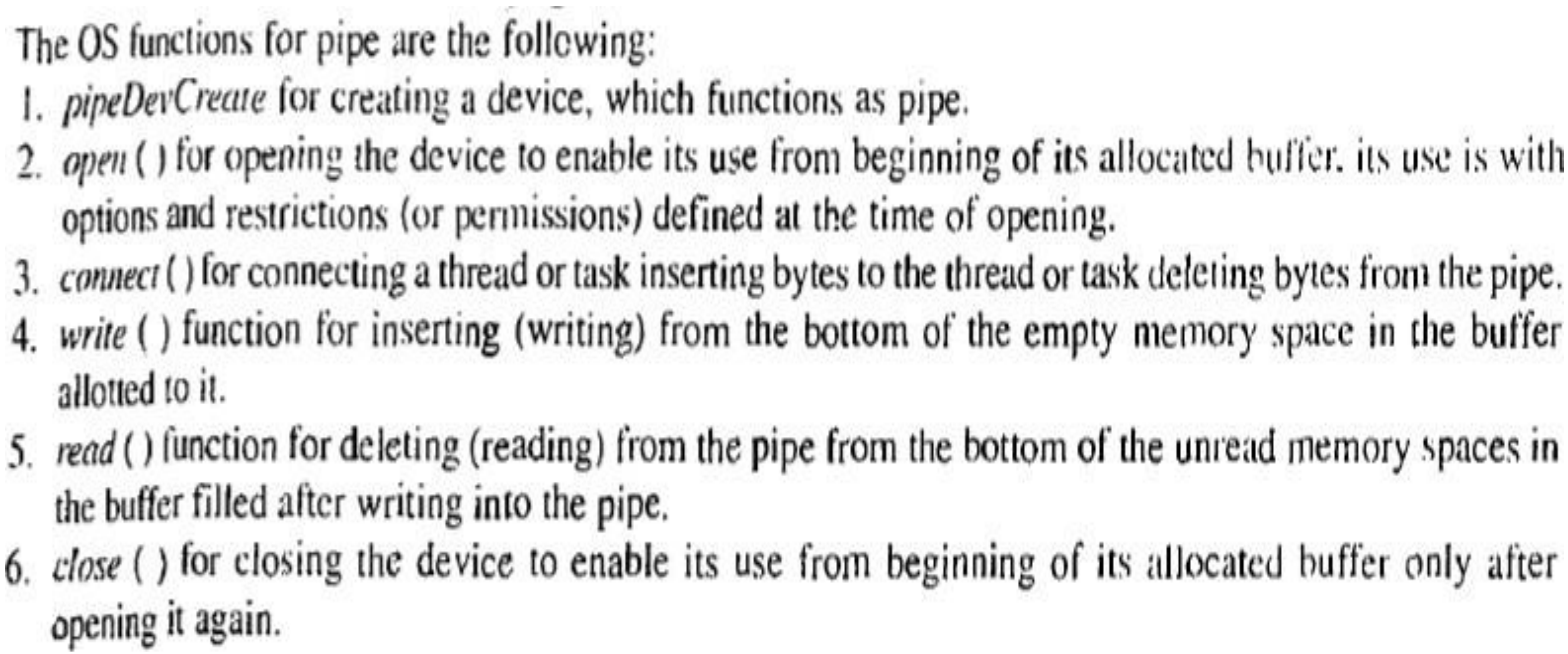

OS device driver functions for pipe
function for Initialization of a pipe device
function for opening a pipe for messages (byte streams)
functions for inserting message (bytes)
function for deleting (read)
pipedevice Create ( )
open ( )
write ( )
read ( )
pipe connect ( )
close ( )
connect a pipe device between source and sink
close the pipe device

Figure (c) shows a pipe messages in the message buffer.


Messages (byte stream) at pipe device memory buffer for writing (insertion) and reading (deletion without wait) as FIFO (first-in first-out)
pipe open
*pFRONT
Awaiting byte waiting for read (deletion)
pipe top
Last writen (inserted) byte at *pBACK
pipe end

The OS functions for the pipe are:



The OS functions for pipe are the following:

1. *pipeDevCreate* for creating a device, which functions as pipe.
2. *open* ( ) for opening the device to enable its use from beginning of its allocated buffer. its use is with options and restrictions (or permissions) defined at the time of opening.
3. *connect* ( ) for connecting a thread or task inserting bytes to the thread or task deleting bytes from the pipe.
4. *write* ( ) function for inserting (writing) from the bottom of the empty memory space in the buffer allotted to it.
5. *read* ( ) function for deleting (reading) from the pipe from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. *close* ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

Sockets

- Provides a device like mechanism for bi-direction communication.
- Sockets allow communication between two different processes on the same or different machines.
- **Need for Sockets for the Inter Process Communication**
- A pipe could be used for inserting the byte steam by a process and deleting the bytes from the stream by another process.
- However, for example, we need that the card information to be transferred from a process A as byte stream to the host machine process B and the B sends messages as byte stream to the A
- There is need for bi-directional communication between A and B.

- We need that the A and B ID or address information when communicating must also be specified either for the destination alone or for the source and destination both. [The messages in a letter are sent along with address specification.]
- We need to use a protocol for communication
- A protocol, for example, provides along with the byte stream information of the address or port of the destination or addresses or ports of source and destinationboth or the protocol may provide for the addresses and ports of source and destination in case of the remote processes
- For example, UDP (user datagram protocol) is used as connectionless protocol
- UDP header contains source port (optional) and destination port numbers, length of the datagram and checksum

- Port means a process or task for specific application.
- Port number specifies the process.
- Datagram means a data, which is independent need not in sequence with the previously sent data.
- Checksum is sum of the bytes to enable the checking of the erroneous data transfer.
- TCP (transport control protocol) is used as connection oriented protocol.

- **Features:**
- Socket Provides for a bi-directional pipe like device, which also use a protocol between source and destination processes for transferring the bytes.

- Provides for establishing and closing a connection between source and destination processes using a protocol for transferring the bytes.
- May provide for listening from multiple sources or multicasting to multiple destinations.
- Two tasks at two distinct places or locally interconnect through the sockets.
- Multiple tasks at multiple distinct places interconnect through the sockets to a socket at a server process.
- The client and server sockets can run on same CPU or at distant CPUs on the Internet
- Sockets can be using a different domain. For example, a socket domain can be TCP (transport control protocol) , another socket domain may be UDP(transport control protocol), the card and host example socket domain is different.
- .

- A pipe does not have protocol based inter-processor communication, while socket provides that.
- A socket can be a client-server socket. Client Socket and server socket functions are different.
- A socket can be a peer-to-peer socket IPC. At source and destination sockets have similar functions.

socket device
create_socket ( )

unlink ( )

bind ( )

listen ( )
accept ( )

receive ( )
send ( )

write ( )
read ( )

close ( )
shutdown ( )

socket error ( )

server

server socket
device

create_socket ( )

bind ( )

connect ( )

send ( )
receive ( )

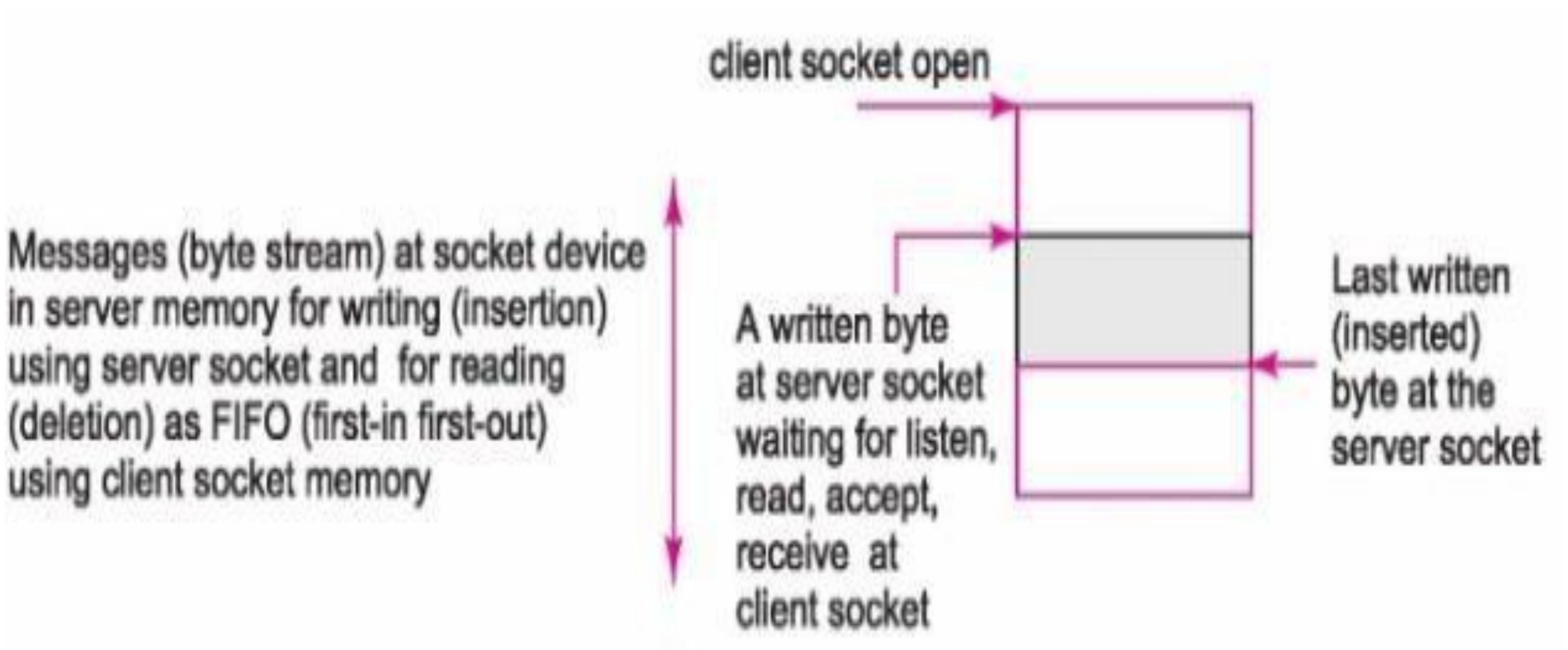
read ( )
write ( )

unlink ( )

close ( )
shutdown ( )

socket error ( )

client


client socket open
Messages (byte stream) at socket device in server memory for writing (insertion) using server socket and for reading (deletion) as FIFO (first-in first-out) using client socket memory
A written byte at server socket waiting for listen, read, accept, receive at client socket
Last written (inserted) byte at the server socket

Types of Socket

- There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.
- Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.
- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.
- **Sequenced Packet Sockets** − It provides a two-way sequenced reliable connectin for datagrams of a fixed maximum length . No protocol for this type has been implemented for any protocol family

Socket Functions

- **Socket()**

  It creates a new socket of a certain socket type ,identifeid by an integer number and allocates system resources to it

- **Connect()**

  This function is used to connect a TCP client with a TCP server

- **Bind()**

  The bind function makes the sockets have its own address information

- **Listen()**

  To set the socket to listen for incoming connections

- **Accept()**

This function is used to accept for an incoming TCP connection

- **Close()**

This function closes the TCP socket

- **Send()**

This function is used to send data to a socket

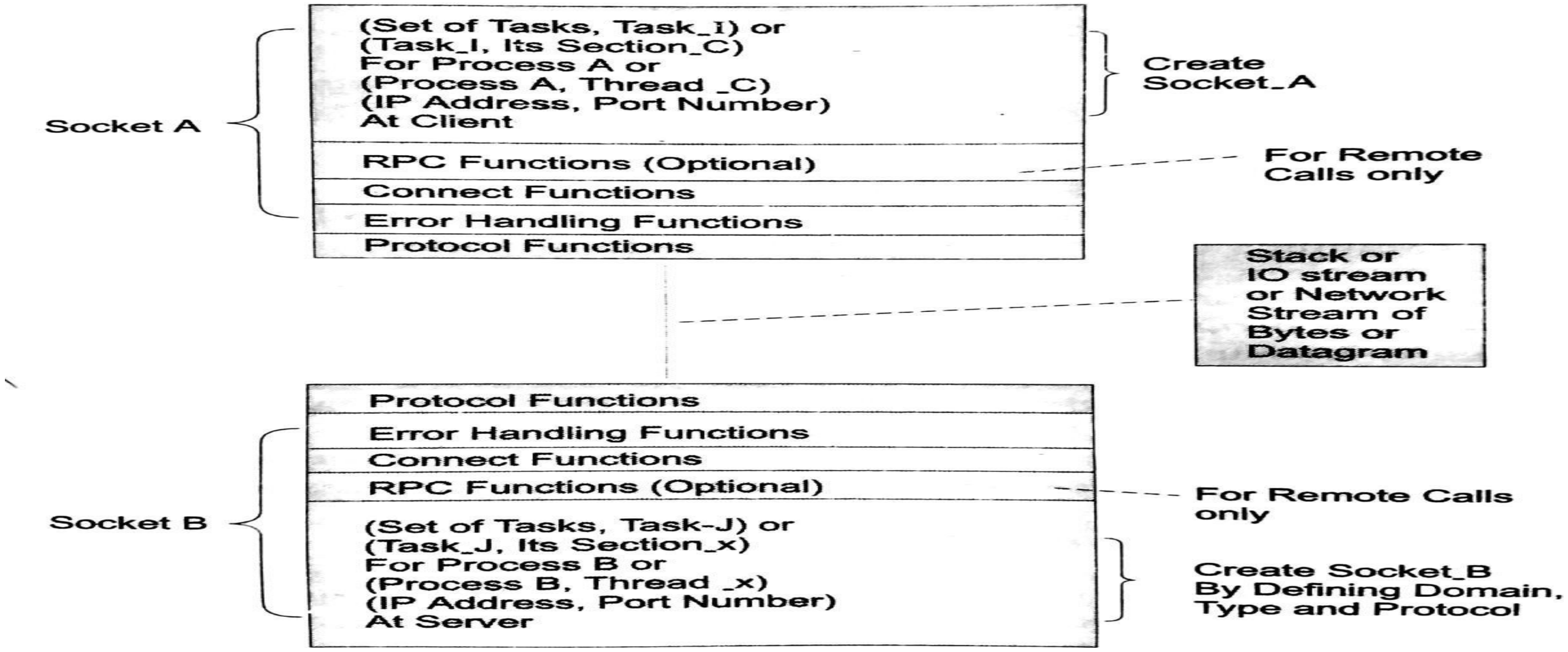
- **Read() and Write()**

A read and write on a stream socket

Remote Procedure Calls

- RPC is used for connecting two remotely placed functions by first using a protocol for connecting the processes.
- It is used in the cases of distributed tasks.
- The RTOS can provide for the use of RPCs. These permits distributed environment for the embedded systems.
- The OS IPC function allows a function or method to run at another address space of shared network or other remote computer.
- The client makes the call to the function that is local or remote and the server response is either remote or local in the call.
- Both systems work in the peer-to-peer communication mode. Each system in peer- to-peer mode can make an RPC.
- An RPC permits remote invocation of the processes in the distributed systems.

Figure below shows the initialised virtual sockets between the client set of tasks and a server set of tasks at RTOS.



Shared Data Problem

```
Static int iTemperatures[2];
Void interrupt vReadTemperatures (void)
{
            iTemperatures[0] = !! read in value from hardware
             iTemperatures[1] = !! read in value from hardware
}
  void main (void)
{
             int iTemp0, iTemp1;
            while (TRUE)
             {
                         iTemp0 = iTemperatures[0];
                        iTemp1 = iTemperatures[1];
                        if (iTemp0 != iTemp1)
                        !! Set off howling alarm;
            }
}
```

- The shared data problem occurs when several functions (or ISRs or tasks) share a variable. Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable before the completion of previous task operations.
- **Steps to eliminate shared data problem:**
- i) Use **modifier volatile** with declaration from variable that returns from interrupt. It warns the compiler that certain variables may modify.
- ii) Use **reentrant function** with atomic instructions that needs its complete execution before it can be interrupted. This part is called critical section.
- iii) Put a **shared variable in a circular queue**. The function that requires the value takes it from queue front. The function that inserts (writes) the value takes it at the queue back.

- iv) Disable the interrupts before critical section starts executing and enable the interrupts on its completion. Even the high priority interrupts than the critical section gets disabled.
- **Application of semaphores in shared data problem**
- A semaphore is special kind of shared program variable.
- A semaphore is a **kernel object that one or more tasks can acquire or release** for the purpose of synchronization or mutual exclusion.
- Mutual exclusion is a provision by which only one task at a time can access a shared resource (port, memory block, etc.)
- The value of a semaphore is a non-negative integer.

- Mutex is semaphore that provides at an instance two tasks mutually exclusive access to resources and is used in solving shared data problem.
- Semaphores does not eliminate shared data problem completely.
- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Types of Semaphores

- There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:
- **Counting Semaphores**These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores**The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

- **Semphore functions**
- **OSSem create**

A semaphore function to create a semaphore and initialize it with an initial value

- **OSSemPend**

Acquire a semaphore or wait.

- **OSSemPost()**

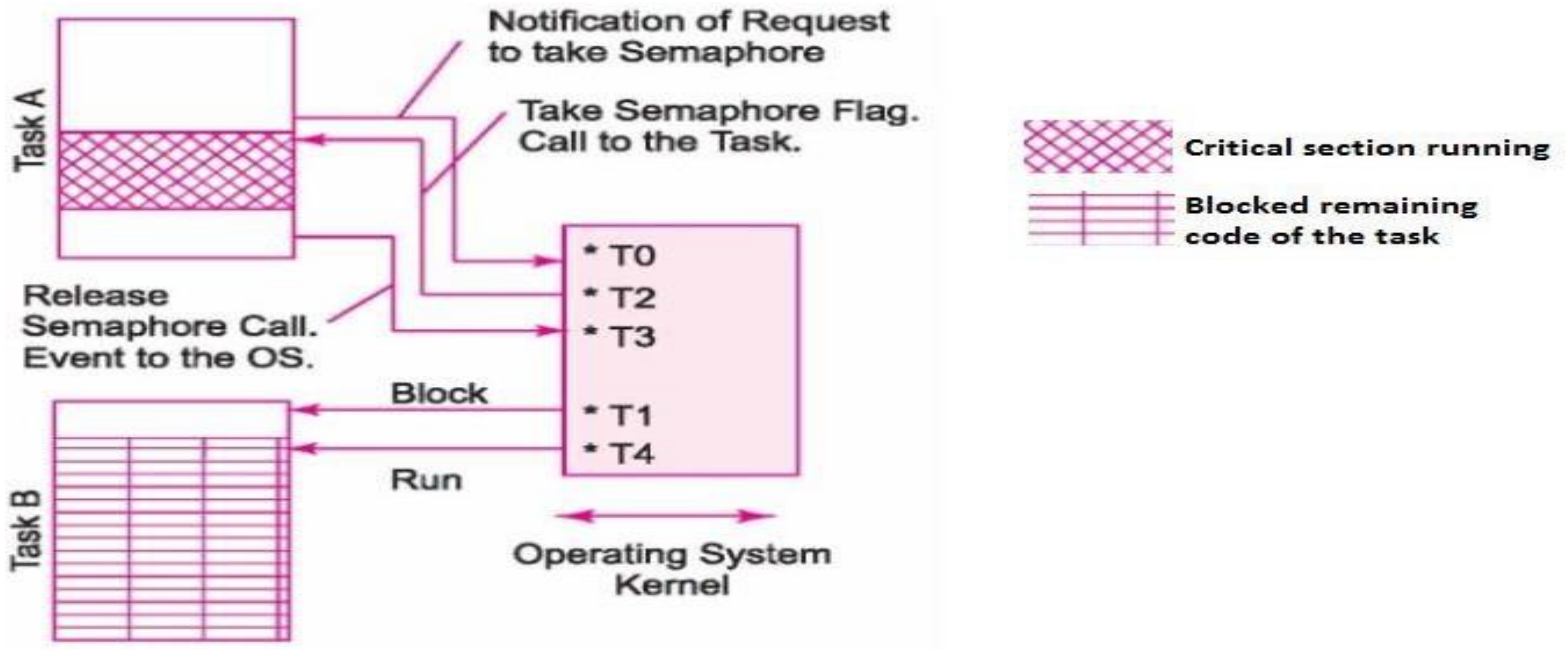
Signals a semaphore

- **OSSemAccept**

Tries to acquire a semaphore without waiting

- **Semaphore as a resource key and for critical sections having shared for critical sections having shared resource (s):**
- Shared Resources like shared memory buffer are to be used only by one task (process or thread) at an instance.
- OS Functions provide for the use of a semaphore resource key for running of the codes in critical section.
- Let a task A, when getting access to a resource/critical section notifies to the OS to have taken the semaphore (take notice)That is, an OS function *OSSemPend()* runsto notify. The OS returns the semaphore as taken (accepted) by decrementing the semaphore from 1 to 0
- Now the task A accesses the resource

- The task A, after completing the access to a resource/critical section it notifies tothe OS to have posted that semaphore (post notice). That is an OS function *OSSemPost()* runs to notify. The OS returns the semaphore as released by incrementing the semaphore from 0 to 1.
- Figure below shows the use of semaphore between A and B. It shows the fivesequential actions at five different times.


Task A
Notification of Request to take Semaphore
Take Semaphore Flag. Call to the Task.
Release Semaphore Call. Event to the OS.
* T0
* T2
* T3
* T1
* T4
Block
Run
Task B
Operating System Kernel
Critical section running
Blocked remaining code of the task

- **<u>Mutex Semaphore for use as resource key:</u>**
- Mutex means mutually exclusive key.
- Mutex is a binary semaphore usable for protecting use of resource by other tasksection at an instance
- Let the semaphore sm has an initial value = 1
- When the semaphore is taken by a task the semaphore sm decrements from 1 to 0 and the waiting task codes starts.
- Assume that the sm increments from 0 to 1 for signalling or notifying end of use of the semaphore that section of codes in the task.
- When sm = 0 ─assumed that it has been taken (or accepted) and other task code section has not taken it yet and using the resource

- When sm = 1─assumed that it has been released (or sent or posted) and other taskcode section can now take the key and use the resource.
- **<u>P and V semaphores</u>**
- An efficient synchronisation mechanism
- P and V semaphores represent by integers in place of binary or unsigned integers.
- The semaphore, apart from initialization, is accessed only through two standard atomic operations - P and V.
- P semaphore function signals that the task requires a resource and if not available waits for it.