# DISTRIBUTED COMPONENT LABARATORY – 15I117 PRELAB – 1

1. **What are the various methods of Inter Process Communication? Pipes:**

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

# Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

* + Send (message)- message size fixed or variable
  + Received (message)

# Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

# Direct Communication:

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

# Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

# Shared Memory:

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

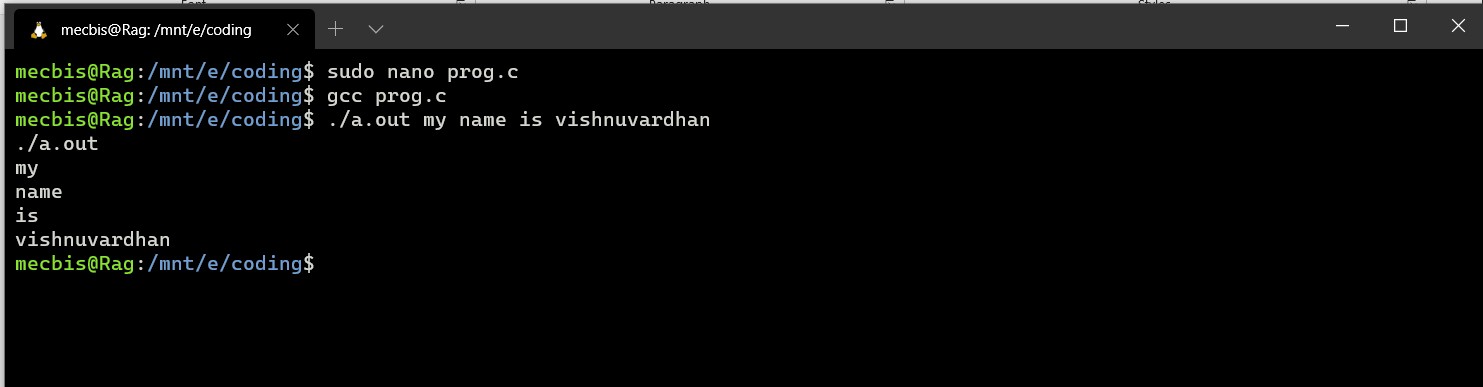
# FIFO:

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.

# Execute any command line argument program in C Code:



**Output:**



1. **What are sockets? Explain Types of sockets**

**Sockets** are seen as the end of the two-way communication between two processes; they allow processes to communicate with each other using a file descriptor and are commonly used in client-server applications that allow for communication between multiple applications.

There are four types of sockets available to the users.

* **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** − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications.

# List out the client and server-side system calls used in socket programming along with explanations

**Socket**

socket (struct proc ∗p, struct socket\_args ∗uap, int retval) In the socket system call:

* + p is a pointer to the proc structure of the process that makes the socket call.
  + uap is a pointer to the socket\_args structure that contains the arguments passed to the process in the socket system call.
  + retval is the return value of the system call.

The socket system call creates a new socket by assigning a new descriptor. The new descriptor is returned to the calling process. Any subsequent system calls are identified with the created socket. The socket system call also assigns the protocol to the created socket descriptor.

# Bind

bind (struct proc ∗p, struct bind\_args ∗uap, int ∗retval) In the bind system call function:

* + s is the socket descriptor.
  + name is the pointer to the buffer that contains the network transport address.
  + namelen is the size of the buffer.

The bind system call associates a local network transport address with a socket. For a client process, it is not mandatory to issue a bind call. The kernel takes care of doing an implicit binding when the client process issues the [connect](https://developer.ibm.com/technologies/systems/articles/au-tcpsystemcalls/#connect) system call. It is often necessary for a server

process to issue an explicit bind request before it can accept connections or start communication with clients.

# Connect

connect (struct proc ∗p, struct connect\_args ∗uap, int ∗retval); In the connect system call:

* + s is the socket descriptor.
  + name is the pointer to the buffer that has the foreign IP/port address pair.
  + namelen is the length of the buffer.

The connect system call is normally called by the client process to connect to the server process. If the client process has not explicitly issued a bind system call before initiating the connection, implicit binding on the local socket is taken care of by the stack.

# Accept

accept(struct proc ∗p, struct accept\_args ∗uap, int ∗retval); In the accept system call:

* + s is the socket descriptor.
  + name is a buffer (an OUT parameter), which contains the network transport address of the foreign host.
  + anamelen is the size of the name buffer.

The accept system call is a blocking call that waits for incoming connections. Once a connection request is processed, a new socket descriptor is returned by accept. This new socket is connected to the client and the other socket s remains in LISTEN state to accept further connections.

# Listen

listen (struct proc ∗p, struct listen\_args ∗uap, int ∗retval) In the listen system call:

* + s is the socket descriptor.
  + backlog is the queue limit for the number of connections on a socket.

The listen call indicates to the protocol that the server process is ready to accept any new incoming connections on the socket. There is a limit on the number of connections that can be queued up, after which any further connection requests are ignored.

# Close

soo\_close(struct file ∗fp , struct proc ∗p); In the close system call:

* + fp is the pointer to the file structure.
  + p is the pointer to the proc structure of the calling process.

The close system call closes or aborts any pending connections on the socket.

# Explain TCP and UDP

**Transmission Control Protocol (TCP)** is a transport layer protocol used to create a connection between applications so that they can exchange data over a communication network.

* + TCP provides a process for communication. Port numbers are appended in the header of the packet to enable this.
  + A three-way handshaking process takes place to open a connection in TCP; hence, it is called a connection-oriented protocol.
  + A packet (called a segment in case of TCP) contains a checksum field used for error control.
  + TCP prevents the sender from sending in more data than the receiver can handle with the window size provided by the receiver.
  + It guarantees the reliable delivery of segments. Incase of segment loss or corruption, segments are re-transmitted.
  + It reassembles the packets in the correct order at the receiver side.
  + It uses a congestion avoidance algorithm to avoid the network from being congested.
  + TCP provides communication between an application program and the Internet Protocol (they are frequently written as TCP/IP.) An application does not need to required packet fragmentation on the transmission medium or other mechanisms for sending data in order to be sent via TCP. [While IP handles actual delivery of the data,](https://en.wikipedia.org/wiki/Transmission_Control_Protocol) TCP keeps track of 'segments' - the individual units of data transmission that a message is divided into for efficient routing through the network.
  + Due to unpredictable network behaviour, IP packets can be lost or delivered out of order; TCP detects and minimizes these issues by reordering packet data or requesting redelivery. This accuracy comes with a trade-off in speed. TCP is known more for reliability, but this accuracy comes from trading speed, sometimes coming with a delay of several seconds.

# User Datagram Protocol (UDP)

* + **User Datagram Protocol (UDP)** is a transport layer protocol that is used to create a connection between applications running on hosts that are connected via a network.
  + UDP provides a process to process communication. Port numbers are appended in the header of the packet to enable this.
  + There is no handshaking required before sending a message; hence, UDP is referred to as a connection-less protocol.
  + UDP is a best-effort delivery service – it does not transmit any lost or corrupt packets.
  + A packet (called a datagram in the case of UDP) contains a checksum field that is used to detect a corrupted datagram. However, it has relatively limited error detection abilities.

# What is called a socket descriptor?

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access a file, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor.

To create a socket, we call the socket function.

# #include <sys/socket.h>

**int socket(int *domain*, int *type*, int *protocol*); Returns: file (socket) descriptor if OK, 1 on error**

1. **Give detailed explanations on**

**Multiple clients involved in socket Programming**

**Socket programming where client and server running in same and different systems Multiple clients involved in socket Programming:**

The simple way to handle multiple clients would be to spawn new thread for every new client connected to the server. This method is strongly not recommended because of various disadvantages, namely:

* + Threads are difficult to code, debug and sometimes they have unpredictable results.
  + Overhead switching of context
  + Not scalable for large number of clients
  + Deadlocks can occur

# Select()

A better way to handle multiple clients is by using **select()** linux command.

* + Select command allows to monitor multiple file descriptors, waiting until one of the file descriptors become active.
  + For example, if there is some data to be read on one of the sockets select will provide that information.
  + **Select** works like an interrupt handler, which gets activated as soon as any file descriptor sends any data.

# Socket programming where client and server running in same and different systems

When Server daemon is running on a host, it is listening to its port and handles all requests sent by clients to the port on the host(server socket). A client must know IP and port of the server (Server socket) to send a request to it. Client’s port is often provided by OS kernel when client starts communication with the server and is freed when communication is over.