**Inter Process Communication**

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task. We will consider:

* Pipes
* Named Pipe(FIFO)
* Message Queues
* Semaphores
* Shared Memory
* Sockets



**PIPE :**   It is the simplest technique to communicate between two processes. The term pipe to mean connecting a data flow from one process to another. Generally you attach, or pipe, the output of one process to the input of another. Lets us understand the pipe concept from our daily life  example, we use pipe fitting from the water tank tank to the wash-room to get water from one end i.e. water tank to the other end i.e. tap in the wash-room. So the pipe is communicating between water tank and the tap in the wash-room. Similarly, pipe in linux is used to communicate between two processes. It is a unidirectional communication i.e. at one time only from one side communication is possible only.

Note: Pipes are used for communication between related processes (parent-child-sibling) on the same linux machine.

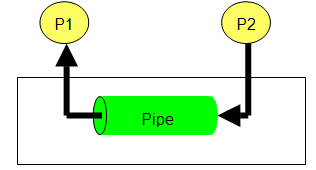
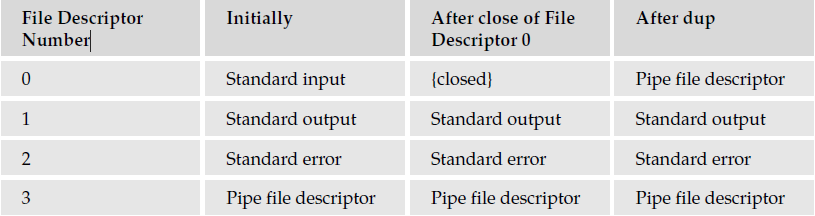
**Syntax :**

**#include <unistd.h>**

**int pipe(int file\_descriptor[2]);**

Pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1.

The two file descriptors returned are connected in a special way. Any data written to file\_descriptor[1] can be read back from file\_descriptor[0]. The data is processed in a first in, first out basis, usually abbreviated to FIFO. This means that if you write the bytes 1, 2, 3 to file\_descriptor[1], reading from file\_descriptor[0] will produce 1, 2, 3. This is different from a stack, which operates on a last in, first out basis, usually abbreviated to LIFO

To create a pipe –

**#mknod my\_pipe\_name p**

Program Example for PIPE communication –

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main()

{

int res,fd[2],s;

char str[25],str1[25];

res=pipe(fd);

if(res==-1)

{

perror("\nPipe Creation Failed...\n");

exit(1);

}

else

{ printf("\nPipe Created Successfully...\n");

printf("\nDescriptors are:%d\t%d\n",fd[0],fd[1]);

switch(fork())

{

case -1:

perror("\nFork Failed....\n");

exit(1);

case 0:

wait(&s);

res=read(fd[0],str1,sizeof(str1));

if(res<0)

{ perror("\nRead Error...\n");

exit(1);

}

else

{ printf("\nChild Process\n");

printf("\nPipe Read Successfully\n");

printf("\nData:%s\n",str1);

}

break;

default:

printf("\nParent Process....\n");

printf("\nEnter Data To Be Written To Pipe\n");

gets(str);

res=write(fd[1],str,sizeof(str));

if(res<0)

{ perror("\nWrite Error Occurred\n");

exit(1);

}

else

{ printf("\nPipe Written Successfully..\n");

}

break;

} } }

Output : –

#gcc -o pipe pipe.c

#./pipe

Pipe Created Successfully…

Descriptors are: 3 4

Parent Process….

Enter Data To Be Written To Pipe

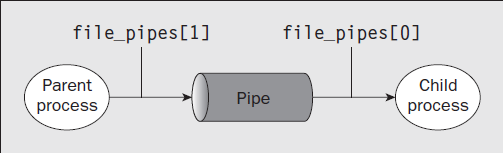
hello

Pipe Written Successfully..

Child Process

Pipe Read Successfully

Data: hello



**Named PIPE (FIFO’s) :**

**Drawbacks of PIPE :**

1.In PIPE communication, only intra-communication between processes is possible i.e. related processes only.

2.Scope of the PIPE is temporary i.e. only upto the program is running.

3.We have to handle file descriptors for communication between two processes.

So, the next inter process communication mechanism is Named PIPE (FIFO).

You can create named pipes from the command line and from within a program.

The preferred command-line method is to use-

**# mkfifo filename**

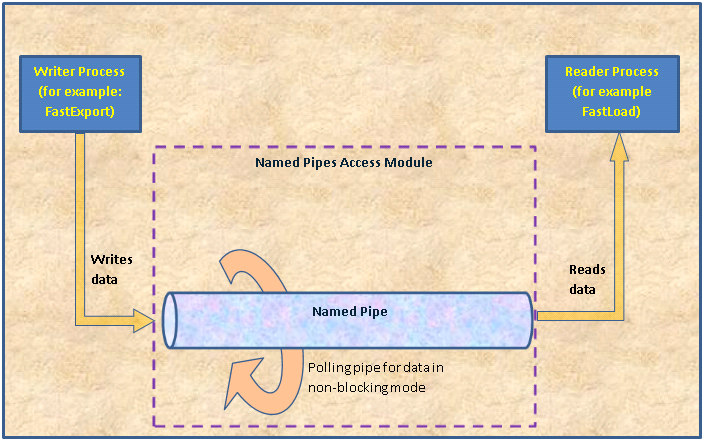
From inside a program, you can use two different calls:

**#include <sys/types.h>**

**#include <sys/stat.h>**

**int mkfifo(const char \*filename, mode\_t mode);**

**Note:**Named Pipes(FIFO)are used for communication between related and unrelated processes on the sane LINUX machine.



**EXAMPLE :**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main()

{

int res,fd;

char str[50],str1[50];

if(access("my\_fifo",F\_OK)==-1)

{ printf("\nFile Permission Are OK\n");

res=mkfifo("my\_fifo",0777);

if(res==-1)

{ perror("\nFifo Can't Created\n");

exit(1);

}

}

printf("\nFifo Created Successfully...\n");

fd=open("my\_fifo",O\_RDWR);

if(fd==-1)

{ perror("\nFifo Can't Open..\n");

exit(1);

}

else

{ printf("\nEnter Data To Be Written..\n");

gets(str);

res=write(fd,str,sizeof(str));

if(res<0)

{ perror("\nWrite Error..\n");

exit(1);

}

else

{ printf("\nWritten Successfully.\n");

}

lseek(fd,0,SEEK\_SET);

res=read(fd,str1,sizeof(str1));

if(res<0)

{ printf("\nRead Error....\n");

exit(1);

}

else

{ printf("\nData From Named Pipe Is:%s\n",str1);

} } }

**OUTPUT :**

File Permissions Are OK

Fifo Created Successfully…

Enter Data To Be Written..

Hello Linux

Written Successfully…

Data From Named Pipe Is:Hello Linux

**Message Queue :**

**Drawbacks of Named-PIPE :**

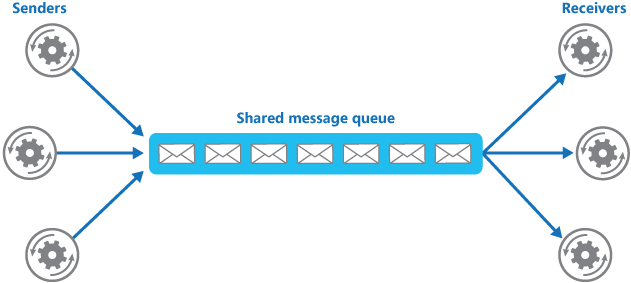
1. In named pipe, communication is only possible if there is a receiver at the opposite to sender side. i.e. receiver must be present to receive message otherwise message will be lost.

So, the next inter process communication mechanism is Message Queue.

**Advantages to use Message Queue :**

1. If the receiver is not present even then also the sender can send a message and the messages will be received  when the receiver is present or online we can say in FIFO manner. i.e. the message send first will be received first.
2. Communication is very fast using message queue. Client-Serve can be implemented easily.

Message queues provide a way of sending a block of data from one process to another. Message queues provide a reasonably easy and efficient way of passing data between two unrelated processes. They have the advantage over named pipes that the message queue exists independently of both the sending and receiving processes, which removes some of the difficulties that occur in synchronizing the opening and closing of named pipes.



**Message Queue Functions :**

|  |  |
| --- | --- |
|  | **#include <sys/msg.h>**  **int msgctl(int msqid, int cmd, struct msqid\_ds \*buff);**  **int msgget(key\_t key, int msgflg);**  **int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);**  **int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);** |

**msgget :**

This function is used to create a message queue and get a key to access this message queue.

**Syntax :**

|  |  |
| --- | --- |
| **1** | **# int msgget(key\_t key, int msgflg);** |

A special bit defined by IPC\_CREAT must be bitwise ORed with the permissions to create a new message queue. The IPC\_CREAT flag is silently ignored if the message queue already exists. The msgget function returns a positive number, the queue identifier, on success or –1 on failure.

**msgsnd :**

The **msgsnd** function allows you to add a message to a message queue:

**Syntax :**

**# int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);**

* The first parameter, msqid, is the message queue identifier returned from a msgget function.
* The second parameter, msg\_ptr, is a pointer to the message to be sent, which must start with a long int type as described previously.
* The third parameter, msg\_sz, is the size of the message pointed to by msg\_ptr. This size must not include the long int message type.
* The fourth parameter, msgflg, controls what happens if either the current message queue is full or the systemwide limit on queued messages has been reached. If msgflg has the IPC\_NOWAIT flag set, the function will return immediately without sending the message and the return value will be –1.  
  If the msgflg has the IPC\_NOWAIT flag clear, the sending process will be suspended, waiting for space to become available in the queue.
* On success, the function returns 0, on failure –1. If the call is successful, a copy of the message data has been taken and placed on the message queue.

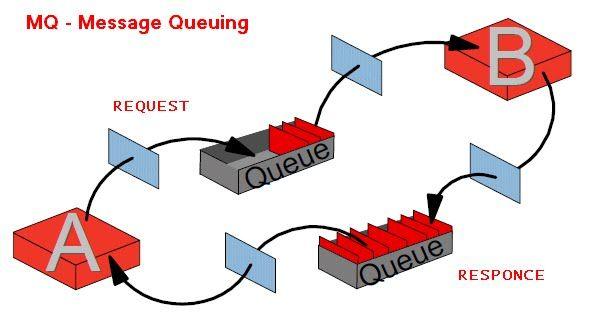
**msgrcv :**

The msgrcv function retrieves messages from a message queue:

**Syntax :**

**int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);**

* The first parameter, msqid, is the message queue identifier returned from a msgget function.
* The second parameter, msg\_ptr, is a pointer to the message to be received, which must start with a long int type as described previously in the msgsnd function.
* The third parameter, msg\_sz, is the size of the message pointed to by msg\_ptr, not including the long int message type.
* The fourth parameter, msgtype, is a long int, which allows a simple form of reception priority to be implemented.



**Example :**

To under stand the message queue let  us take an example with a sender and a receiver.

**sender.c –**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/types.h>

#include<sys/msg.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

struct msgqbuf

{ long mtype;

char mtext[50];

};

typedef struct msgqbuf msgque;

int main()

{ int msgqid,siz,siz1,i;

char ch;

msgque q,q1;

msgqid=msgget(1234,0666|IPC\_CREAT);

if(msgqid==-1)

{ perror("\nMsg Queue Can't Create...\n");

exit(1); }

else

{ printf("\nMsg Queue Created Successfully...\n");

q.mtype=0;

do

{ printf("\nEnter Data To Be sent\n");

ch=getchar();

         gets(q.mtext);

siz=msgsnd(msgqid,q.mtext,strlen(q.mtext),0);

if(siz==-1)

{ perror("\nMsg Sending Failed....\n");

exit(1);

}

else if(siz==0)

{ printf("\nMsg Sent Successfully...\n");

}

printf("Do u want to continue sending msg..\nY:Yes\nN:No\n");

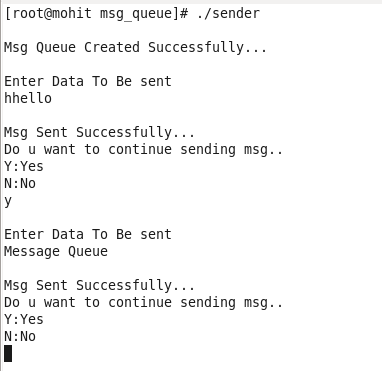
        scanf(" %c",&ch);

        }while(ch=='Y'||ch=='y');

        }}

**Compile it –**

# gcc -0 sender sender.c



**receiver.c –**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/types.h>

#include<sys/msg.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

struct msgqbuf

{ long mtype;

char mtext[50];

};

typedef struct msgqbuf msgque;

int main()

{ int msgqid,siz,siz1;

msgque q,q1;

char ch,ch1;

send: msgqid=msgget(1234,0666|IPC\_CREAT);

if(msgqid==-1)

{ perror("\nMsg Queue Can't Create...\n");

exit(1);

}

else

{ printf("\nMsg Queue Created Successfully...\n");

                 printf("\nWaiting to receive message ...\n");

q.mtype=0;

siz=msgrcv(msgqid,q.mtext,BUFSIZ,0,0);

if(siz==-1)

{ perror("\nMsg Recieve Error\n");

exit(1);

}

else

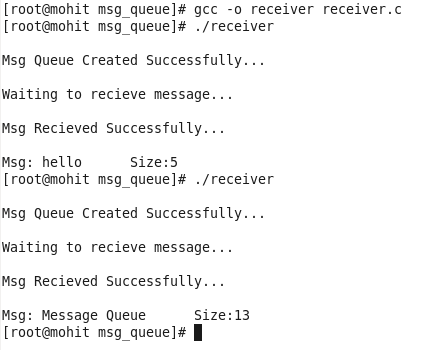
{ printf("\nMsg Recieved Successfully...\n");

printf("\nMsg:%s\tSize:%d\n",q.mtext,siz);

} } }

**Explanation –**When we compile the sender.c a message queue is created with key 1234 and write data to enter into queue and then compile receiver.c that has the same key 1234. The data send from the sender.c will be received from the receiver.c using mssgrcv function.

**Output :**

****

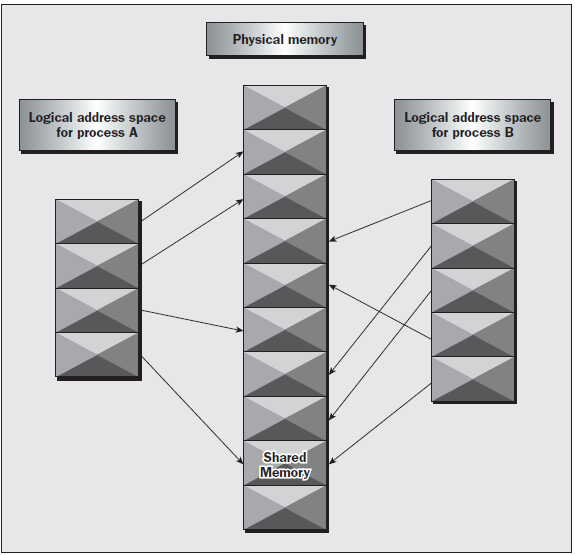
**Disadvantage of Message Queue :-**

1. The only disadvantage of message queue is that we can send data only of 1024 bytes. But in shared memory the more size of data can be transfer between two processes and that depends on the physical memory of the system.

So to overcome this problem we use shared memory concept.

**Shared Memory :**

In this concept two unrelated processes can access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes. Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process. Other processes can then “attach” the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.



**Functions for shared memory**

**#include <sys/shm.h>**

**void \*shmat(int shm\_id, const void \*shm\_addr, int shmflg);**

**int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);**

**int shmdt(const void \*shm\_addr);**

**int shmget(key\_t key, size\_t size, int shmflg);**

**shmget :**

This function is defined in sys/shm.h header file and used to create shared memory.

**Syntax :**

**int shmget(key\_t key, size\_t size, int shm\_flag);**

It returns a shared memory identifier and flay used is IPC\_PRIVATE that creates a private shared memory to the processes and IPC\_CREAT flag to create memory. The second argument is size which tells the amount of memory required for processes in bytes.

**shmat :**

To enable access to the memory that is created we have to attach it to the address space of a process, so this is done with the help of shmat function.

Syntax :

**void \*shmat(int shm\_id, const void \*shm\_addr, int shm\_flag);**

**shmdt:**

The shmdt function detaches the shared memory from the current process. It takes a pointer to the address returned by shmat. On success, it returns 0, on error –1.

**EXAMPLE :**

First at the sender side we will create a logical memory by using shmget function of size say 1024 bytes.

**sender.c –**

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{ int shmid,run=1;

char \*shm\_mem,str[50],i,ch;

void \*ptr;

shmid=shmget(1234,1024,0666|IPC\_CREAT);

if(shmid==-1)

{ perror("\nShared Memmory Can't Be Allocated....\n");

exit(1);

}

else

{ printf("\nShared Memory Allocated Successfully...\n");

ptr=shmat(shmid,(void \*)0,0);

if(ptr==(void \*)-1)

{ perror("\nMemory Attachment Failed..\n");

exit(1);

}

else

{ printf("\nMemory Attached Successfully At Add:%ld\n",(int \*)ptr);

while(run)

{ i=0;

ch=getchar();

do

{ str[i]=ch;

ch=getchar();

i++;

}while(ch!='\n');

str[i]='\0';

if(strcmp(str,"bye")==0)

{ run=0;

}

else

{ shm\_mem=(char \*)ptr;

strcpy(shm\_mem,str);

printf("\nData U written :%s\n",shm\_mem);

sleep(2);

}}}}}

**OUTPUT :**



Here the shared memory is created to communication between two processes and the memory address is returned by a pointer. Second process is say receiver.c that will receive the data from that shared memory.

**receiver.c –**

#include<sys/ipc.h>

#include<sys/shm.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{ int shmid,run=1;

char \*shm\_mem,str[50],i,ch;

void \*ptr;

shmid=shmget(1234,1024,0666|IPC\_CREAT);

if(shmid==-1)

{ perror("\nShared Memmory Can't Be Allocated....\n");

exit(1);

}

else

{ printf("\nShared Memory Allocated Successfully...\n");

ptr=shmat(shmid,(void \*)0,0);

if(ptr==(void \*)-1)

{ perror("\nMemory Attachment Failed..\n");

exit(1);

}

else

{ printf("\nMemory Attached Successfully At Add:%ld\n",(int \*)ptr);

shm\_mem=(char \*)ptr;

strcpy(str,shm\_mem);

while(1)

         { if(run)

{ printf("\nData U written :%s\n",str);

sleep(1);

run=0;

}

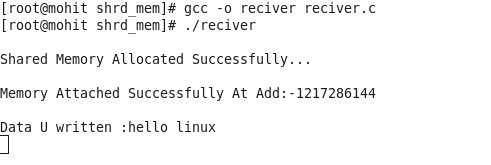
else if(str=="bye")

{ run=0;

exit(0);

} }}}}

**OUTPUT :**

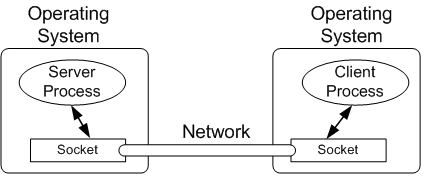


**Disadvantage of Shared Memory :**

By itself, shared memory doesn’t provide any synchronization facilities. So you usually need to use some other mechanism to synchronize access to the shared memory. Typically, you might use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory. There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It’s the responsibility of the programmer to synchronize access

# ****SOCKETS**** ****:****

A socket is a communication mechanism that allows client/server systems to be developed either locally, on a single machine, or across networks. Linux functions such as printing, connecting to databases, and serving web pages as well as network utilities such as rlogin for remote login, http for hyper text transfer protocol and ftp for file transfer usually use sockets to communicate.  
Sockets are created and used differently from pipes because they make a clear distinction between client and server. The socket mechanism can implement multiple clients attached to a single server.



## **Socket Connections**

First, a server application creates a socket, which like a file descriptor is a resource assigned to the server process and that process alone. The server creates it using the system call socket, and it can’t be shared with other processes.

Next, the server process gives the socket a name. For example, a web server typically creates a socket on port 80, an identifier reserved for the purpose. Web browsers know to use port 80 for their HTTP connections and port 443 for HTTPS connections.

* A socket is named using the system call bind.
* The system call, listen, creates a queue for incoming connections.
* The server can accept them using the system call accept.

When the server calls accept, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client.

The client side of a socket-based system is more straightforward. The client creates an unnamed socket by calling socket.

## **Socket Attributes**

Sockets are characterized by three attributes: domain, type, and protocol.

**Socket Domains :**Domains specify the network medium that the socket communication will use. The most common socket domain is AF\_INET, which refers to Internet networking that’s used on many Linux local area networks and, of course, the Internet itself.

**Protocols :** Internet protocols provide two communication mechanisms with distinct levels of service: streams and datagrams.

* **Stream Sockets :**Stream sockets (in some ways similar to standard input/output streams) provide a connection that is a sequenced and reliable two-way byte stream. Thus, data sent is guaranteed not to be lost, duplicated, or  
  reordered without an indication that an error has occurred.
* **Datagram Sockets :**  A datagram socket, specified by the type SOCK\_DGRAM, doesn’t establish and maintain a connection. There is also a limit on the size of a datagram that can be sent. It’s transmitted as a single network message that may get lost, duplicated, or arrive out of sequence.

### **Creating a Socket**

The socket system call creates a socket and returns a descriptor that can be used for accessing the socket.

Syntax :

#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

**Naming a Socket**

To make a socket (as created by a call to socket) available for use by other processes, a server program needs to give the socket a name. The bind system call assigns the address specified in the parameter, address, to the unnamed socket associated with the file descriptor socket. The length of the address structure is passed as address\_len.

Syntax :

**#include <sys/socket.h>**

**int bind(int socket, const struct sockaddr \*address, size\_t address\_len);**

**Creating a Socket Queue**

To accept incoming connections on a socket, a server program must create a queue to store pending requests. It does this using the listen system call.

Syntax :

**#include <sys/socket.h>**

**int listen(int socket, int backlog);**

A value of 5 for backlog is very common.

**Accepting Connections**

Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the accept system call.

Syntax :

**#include <sys/socket.h>**

**int connect(int socket, const struct sockaddr \*address, size\_t address\_len)**

**Closing a Socket**

You can terminate a socket connection at the server and client by calling close, just as you would for low level file descriptors.

### **Example of SOCKET Communication in LINUX**

Let us take a server-client example to illustrate socket communication between two processes.

* **client.c**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<netdb.h>

#include<unistd.h>

#define DATA "Hello World"

int main(int argc,char \*argv[])

{ int sock,i;

struct hostent \*hp;

struct sockaddr\_in server;

char buff[1024],data[100],ch;

while(1)

{ sock=socket(AF\_INET,SOCK\_STREAM,0);

if(sock<0)

{ perror("\nSocket failed\n");

exit(1);

}

server.sin\_family=AF\_INET;

hp=gethostbyname(argv[1]);

if(hp==0)

{ perror("\ngethostbyname failed......\n");

exit(1);

}

memcpy(&server.sin\_addr,hp->h\_addr,hp->h\_length);

server.sin\_port=htons(6000); //converts unsigned short integer from host byte order to network bbyte order.

if(connect(sock,(struct sockaddr \*)&server,sizeof(server))<0)

{ perror("Connect failed\n");

exit(1);

}

printf("Enter data to send:");

// scanf("%s",data);

i=0;

ch=getchar();

do

{ data[i]=ch;

ch=getchar();

i++;

}while(ch!='\n');

data[i]='\0';

if(send(sock,data,sizeof(data),0)<0)

{ perror("send failed\n");

close(sock);

exit(1);

}

printf("\nSent:%s\n",data);

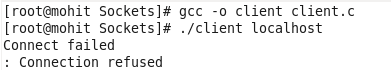
close(sock);

}

return 0;

}

**OUTPUT :**



On compiling client.c, we provide localhost as a command line argument to connect the system with localhost i.e. 127.0.0.1, But the connection was refused because the server is unavaliable.

* **server.c**

#include<stdio.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<string.h>

#include<stdlib.h>

int main(int argc,char \*agrv[])

{ /\*Variables\*/

int sock;

struct sockaddr\_in server;

int mysock;

char buff[1024];

int rval;

/\*Create Socket\*/

sock=socket(AF\_INET,SOCK\_STREAM,0);

if(sock<0)

{ perror("\nFailed to create socket\n");

exit(1);

}

server.sin\_family=AF\_INET;

server.sin\_addr.s\_addr=INADDR\_ANY;

server.sin\_port=htons(6000);

/\*Call Bind\*/

if(bind(sock,(struct sockaddr \*)&server,sizeof(server)))

{ perror("\nBind Failed\n");

exit(1);

}

/\*Listen\*/

listen(sock,5);

/\*Accept\*/

do

{ printf("\nWaiting....\n");

mysock=accept(sock,(struct sockaddr \*)0,0);

if(mysock==-1)

{ perror("\nAccept failed\n");

exit(1);

}

else

{ memset(buff,0,sizeof(buff));

if((rval=recv(mysock,buff,sizeof(buff),0))<0)

{ perror("\nReading stream message error\n");

exit(1);

}

else if(rval==0)

printf("\nEnding Connection\n");

else

{ printf("MSG:%s\n",buff);

}

printf("Got the message(rval=%d)",rval);

close(mysock);

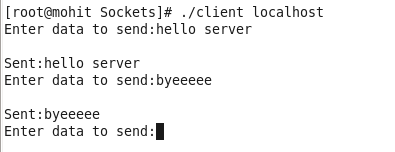
}

}while(1);

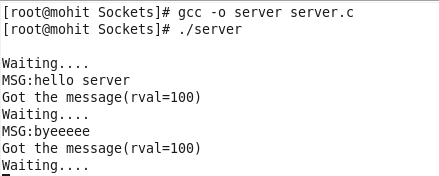
return 0;}

**Output :**

**Now again run client .c  –**

****

**compile server.c and run simultaneously ,**

****