PART-B

1.AIM: Implement the Inter Process Communication using PIPE’s.

DESCRIPTION:

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.

Pipes:-

A pipe is a technique for passing information from one program process to another.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.

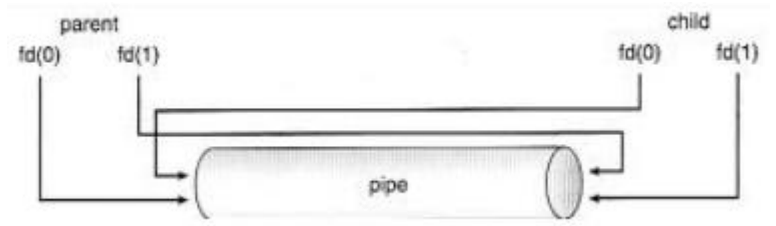
It opens a pipe, which is an area of main memory that is treated as a *“virtual file”*.The pipe can be used by the creating process, as well as all its child processes, for reading and writing.

One process can write to this “virtual file” or pipe and another related process can read from it.

A pipe provides a one-way flow of data. A pipe is created by the pipe system call.

Syntax for creating pipes:

```
int pipe ( int *filedes ) ;
        return 0 upon success and –1 upon failure
        filedes[0] is open for reading
        filedes[1] is open for writin
```



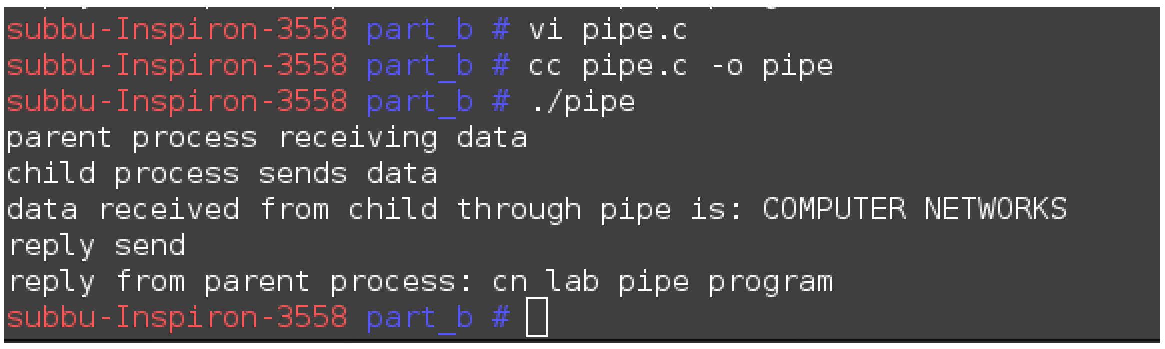
- ➢ First, a process creates a pipe and then forks to create a copy of itself.
- ➢ Next the parent process closes the read end of the pipe and the child process closes the write end of the pipe.
- ➢ The fork system call creates a copy of the process that was executing.

- The process that executed the fork is called the parent process and the new process is called the child process.
- The fork system call is called once but it returns twice.

PROGRAM:-

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int fd1[2],fd2[2],n,pid;
        char buf1[30],buf2[30];

        pipe(fd1);
        pipe(fd2);

        pid=fork();
        if(pid==0)
        {
                close(fd1[0]);
                printf("child process sends data\n");
                write(fd1[1],"COMPUTER NETWORKS ",30);

                close(fd2[1]);
                read(fd2[0],buf1,30);
                printf("reply from parent process: %s\n",buf1);
                sleep(2);
        }
        else
        {
                close(fd1[1]);
                printf("parent process receiving data\n");
                n=read(fd1[0],buf2,sizeof(buf2));
                printf("data received from child through pipe is: %s\n",buf2);
                sleep(3);
                close(fd2[0]);
                write(fd2[1],"cn lab pipe program",30);
                printf("reply send\n");
        }
}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # vi pipe.c
subbu-Inspiron-3558 part_b # cc pipe.c -o pipe
subbu-Inspiron-3558 part_b # ./pipe
parent process receiving data
child process sends data
data received from child through pipe is: COMPUTER NETWORKS
reply send
reply from parent process: cn lab pipe program
subbu-Inspiron-3558 part_b #
```

2.AIM: Implement the Inter Process Communication using PIPE's.

DESCRIPTION:-

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.

FIFO(named pipe):-

It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.

A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Usually a named pipe appears as a file, and generally processes attach to it for inter-process communication.

A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

A FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file.

In order to create a FIFO file, a function calls i.e. mkfifo is used.

Creating a FIFO file(syntax):

intmkfifo(constchar*pathname, mode_t mode);

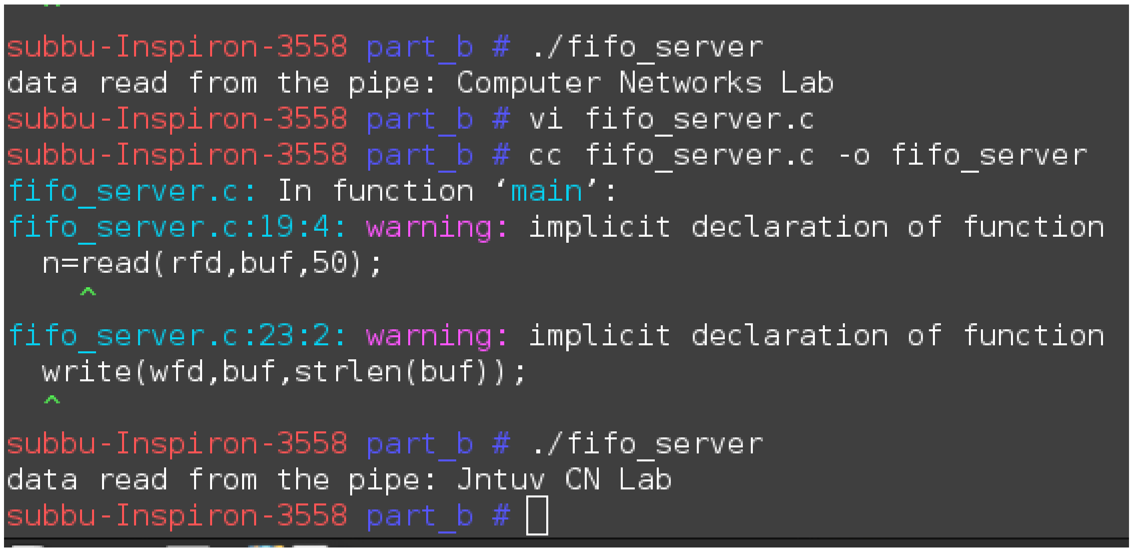
- mkfifo() makes a FIFO special file with name *pathname*.
- Here *mode* specifies the FIFO's permissions.
- It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).
- As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. open, read, write, close.

PROGRAM:-

SERVER PROGRAM:-

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<fcntl.h>
#include<string.h>
#include<sys/stat.h>
#include<sys/types.h>
int main()
{
        int wfd,rfd,count,n,val1,val2;
        charbuf[50];
        val1=mkfifo("fifo1",0666);
        val2=mkfifo("fifo2",0666);
        rfd=open("fifo1",O_RDONLY);
        wfd=open("fifo2",O_WRONLY);
        n=read(rfd,buf,50);
        buf[n]='\0';
        printf("data read from the pipe: %s\n",buf);
        write(wfd,buf,strlen(buf));
}
```

OUTPUT:

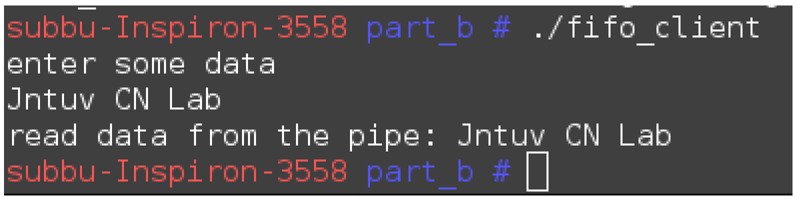


```
subbu-Inspiron-3558 part_b # ./fifo_server
data read from the pipe: Computer Networks Lab
subbu-Inspiron-3558 part_b # vi fifo_server.c
subbu-Inspiron-3558 part_b # cc fifo_server.c -o fifo_server
fifo_server.c: In function ‘main’:
fifo_server.c:19:4: warning: implicit declaration of function
  n=read(rfd,buf,50);
    ^
fifo_server.c:23:2: warning: implicit declaration of function
  write(wfd,buf,strlen(buf));
  ^
subbu-Inspiron-3558 part_b # ./fifo_server
data read from the pipe: Jntuv CN Lab
subbu-Inspiron-3558 part_b #
```

CLIENT PROGRAM:-

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
int main()
{
        int wfd,rfd,n;
        char buf[50],line[50];
        wfd=open("fifo1",O_WRONLY);
        rfd=open("fifo2",O_RDONLY);
        printf("enter some data\n");
        gets(line);
        write(wfd,line,strlen(line));

        n=read(rfd,buf,50);

        buf[n]='\0';

        printf("r data from the pipe: %s\n",buf);

}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # ./fifo_client
enter some data
Jntuv CN Lab
read data from the pipe: Jntuv CN Lab
subbu-Inspiron-3558 part_b #
```

3.AIM: Implement the file transfer using message queue forms of IPC

DESCRIPTION:

- Message queueing is a method by which process (or program instances) can exchange or pass data using an interface to a system-managed queue of messages.
- Messages can vary in length and be assigned different types or usages.
- A message queue can be created by one process and used by multiple processes that read and/or write messages to the queue.
- For example, a server process can read and write messages from and to a message queue created for client processes. The message type can be used to associate a message with a particular client process even though all messages are on the same queue.
- The system calls that we used here are:

int msgget (key_t key, int msgflag);

-- A new message queue is created or an existing message Queueis accesed with the msgget system call. The value returned by msgget is the message queue identifier, msqid, or -1 if an error occurred.

int msgsnd (int msqid , struct msgbuf *ptr, int length);

-- once a message queue is opended with msgget,we put a message otn the queue using the **msgsnd system call**.

int msgrcv(int msqid,struct msgbuf *ptr, int length, long msgtype, int flag);

-- A message is read from a message queue using the msgrcv system call.

int msgctl(int msqid, int cmd, struct msqid_ds *buff);

-- the msgctl system call providea a variety of control operations on a message queue .

PROGRAM:

SERVER PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/ipc.h>
#include<sys/types.h>
#include<sys/msg.h>
#include<unistd.h>
int main()
{
        int msqid;
        key_t key;
        struct
        {
                inttype_field;
                charmsg_text[100];
        }msgbuf;
        key=111999;
        msqid=msgget(key,IPC_CREAT|0666);
        if(msqid<0)
      {
                perror("message queue is not created/opened\n");
                exit(1);
      }
```

```
	msgrcv(msqid,&msgbuf,sizeof(msgbuf),0,0);
	printf("\nreceived filename is %s\n",msgbuf.msg_text);
	msgctl(msqid,IPC_RMID,NULL);
}
```

OUTPUT:

```
Received filename is cnlab
```

CLIENT PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
int main()
{
	int msqid;
	key_t key;
	struct
	{
		inttype_field;
		charmsg_text[100];
	}msgbuf;
	key=111999;
	msqid=msgget(key,IPC_CREAT|0666);
```

```
    if(msqid<0)
   {
           perror("message queue is not created/opened\n");
           exit(1);
   }
    msgbuf.type_field=1;
    printf("enter file name\n");
    scanf("%s",msgbuf.msg_text);
    msgsnd(msqid,&msgbuf,sizeof(msgbuf),0);
    printf("sended data is: %s\n",msgbuf.msg_text);
}
```

OUTPUT:

```
Enter file name
Cnlab
```

4.AIM: Write a program to create an integer variable using shared memory concept and increment the variable simultaneously by two processes. Use semaphores to avoid race conditions.

DESCRIPTION:

- A **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.
- Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems.
- Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.
- **shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.
- Shared memory is an efficient means of passing data between programs.

key_t ftok(char *pathname, char proj);

-- It converts a pathname and a project identifier to a system V IPC key

int semget(key_t key, int nsems, int semflag);

-- semaphore is created or an existing semaphore is accesed with the segment system call.

int semop(int semid, struct sembuf *opstr, unsigned int nops):

--operations are performed on one or more of the semaphore values in the set using semop system call.

int semctl(int semid, int semnum, int cmd, union sem arg);

union semnum{

int val;

struct semid_ds *buff;

ushort *array;

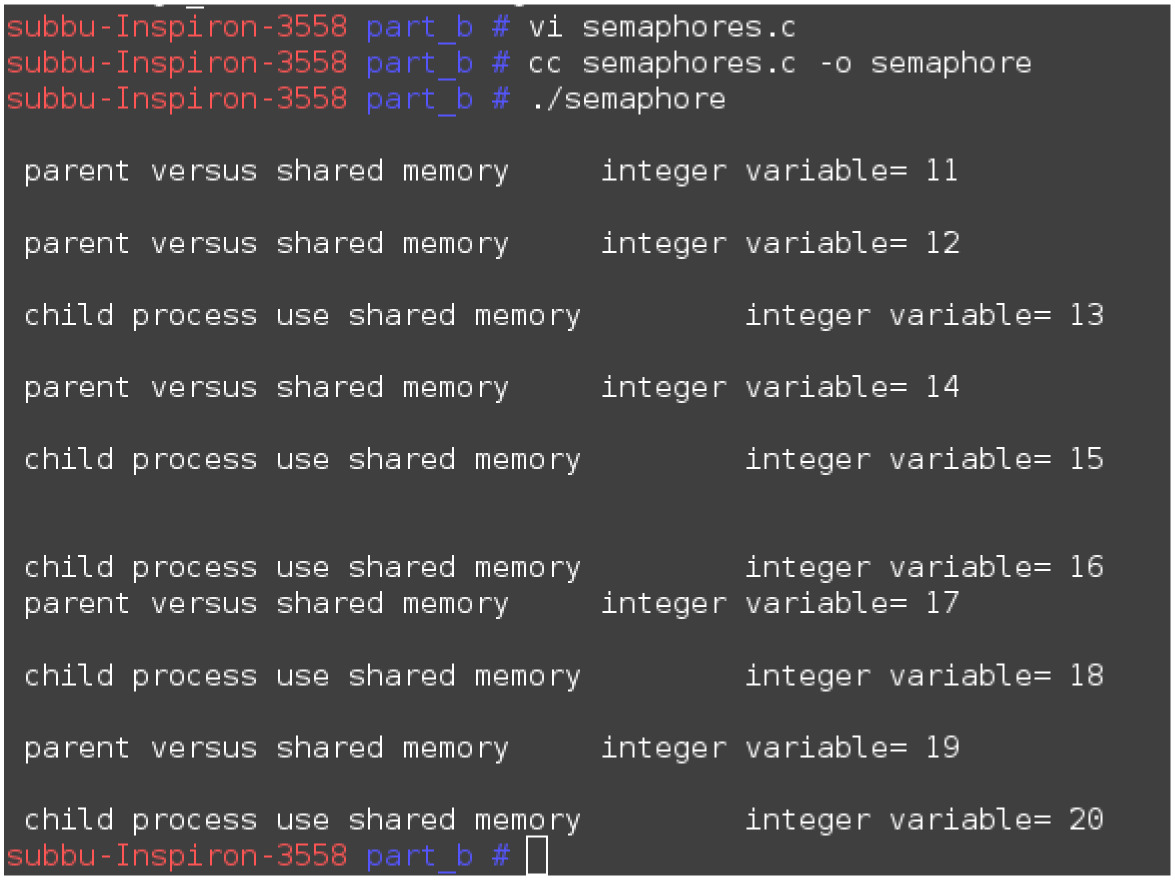
}arg;

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
int *integer=0;
int accessmem(intsemid)
{
	structsembuf sop;
	sop.sem_num=0;
	sop.sem_op=-1;
	sop.sem_flg=0;
	semop(semid,&sop,0);
	(*integer)++;
	printf("\t integer variable= %d",(*integer));
	sop.sem_num=0;
	sop.sem_op=1;
	sop.sem_flg=0;
	semop(semid,&sop,0);
```

```
}
int main()
{
        int shmid,semid,pid;
        char *shm;
        shmid=shmget((key_t)10,30,IPC_CREAT|0666);
        shm=shmat(shmid,NULL,0);
        semid=semget((key_t)10,1,IPC_CREAT|0666);
        integer=(int *)shm;
        pid=fork();
        if(pid==0)
        {
                int i=0;
                while(i<5)
                {
                        sleep(2);
                        printf("\n child process use shared memory ");
                        accessmem(semid);
                        i++;
                }
        }
        else
        {
                int j=0;
                while(j<5)
                {
                        sleep(j);
                        printf("\n parent versus shared memory ");
                        accessmem(semid);
                        j++;
                }
        }
        shmctl(semid,IPC_RMID,0);
}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # vi semaphores.c
subbu-Inspiron-3558 part_b # cc semaphores.c -o semaphore
subbu-Inspiron-3558 part_b # ./semaphore

 parent versus shared memory     integer variable= 11

 parent versus shared memory     integer variable= 12

 child process use shared memory         integer variable= 13

 parent versus shared memory     integer variable= 14

 child process use shared memory         integer variable= 15


 child process use shared memory         integer variable= 16
 parent versus shared memory     integer variable= 17

 child process use shared memory         integer variable= 18

 parent versus shared memory     integer variable= 19

 child process use shared memory         integer variable= 20
subbu-Inspiron-3558 part_b # 
```

5.AIM: Design TCP iterative Client and server application to reverse the given input sentence.

DESCRIPTION:

- The TCP/IP protocol allows systems to communicate even if they use different types of network hardware.
- TCP controls the accuracy of data transmission. IP, or Internet Protocol, performs the actual data transfer between different systems on the network or Internet.
- Using TCP binding, you can create both client and server portions of client/server systems.
- In the client/server type of distributed database system, users on one or more client systems can process information stored in a database on another system, called the server.
- The system calls that we used here are:

#include <sys/socket.h>

int socket int family, int type, int protocol);

The family specifies the protocol family

Family	**Description**
AF_INET	IPV4 protocol
AF_INET6	IPV6 protocol

Type	**Description**
SOCK_STREAM	Stream description
SOCK_DGRAM	Datagram socket

The protocol argument to the socket function is set to zero except for raw sockets.

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

-- The connect function is used by a TCP client to establish a connection with a TCP server.

int bind(int sockfd, const struct sockaddr *myaddr, s ocklen_t addrlen);

-- The bind function assigns a local protocol address to a socket.

#include<strings.h>

void bzero(void *dest,size_t nbytes);

--It sets the specified number of bytes to 0(zero) in the destination. We often use this function to initialize a socket address structure to 0(zero).

int listen(int sockfd, int backlog);

-- **T**he second argument to this function specifies the maximum number of connection that the kernel should queue for this socket.

int accept(int sockfd, struct sockaddr *cliaaddr, socklen_t *addrlen);

-- **T**he cliaddr and addrlen argument are used to ret urn the protocol address of the connected peer processes (client)

int close(int sockfd);

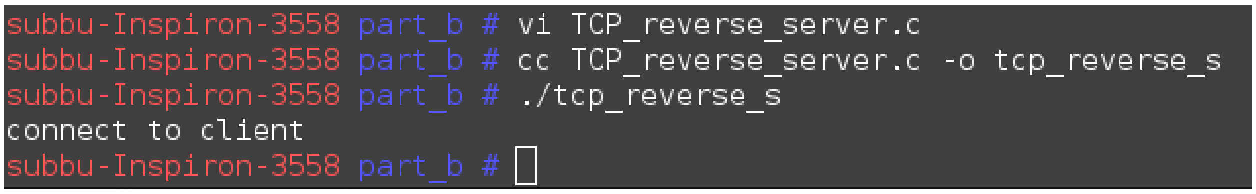
-- **T**he normal UNIX close function is also used to close a socket and terminate a TCP connection.

PROGRAM:

SERVER SIDE PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<unistd.h>

#include<netinet/in.h>

#include<sys/socket.h>

int main()

{

        char senddata[30],reversedata[30];

        int listenfd,connfd,clilen,i,j;

        int n,len;

        struct sockaddr_inservaddr,cliaddr;
        listenfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));

        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(20154);
        bind(listenfd,(structsockaddr*)&servaddr,sizeof(servaddr));
        listen(listenfd,1);
        clilen=sizeof(cliaddr);
        connfd=accept(listenfd,(structsockaddr*)&cliaddr,&clilen);
        printf("connect to client\n");
        while(1)
        {
                if((n=read(connfd,senddata,30))==0)
                        break;
                if(strcmp(senddata,"quit\n") == 0)
                        break;
                len=strlen(senddata);
                for(i=0;senddata[i]!='\0';i++)
                        reversedata[len-1-i]=senddata[i];
                reversedata[i]='\0';
                write(connfd,reversedata,n);
        }
}
```

OUTPUT:

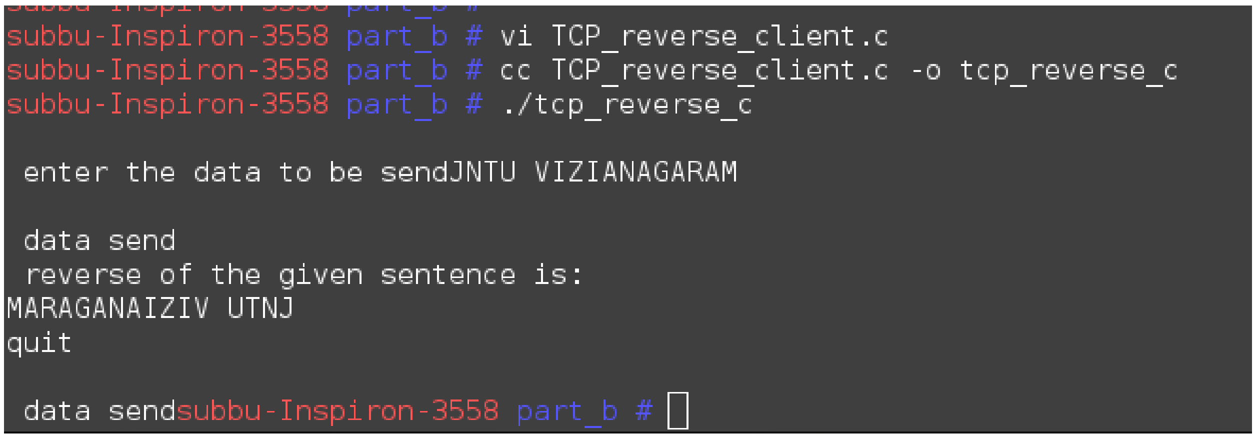


```
subbu-Inspiron-3558 part_b # vi TCP_reverse_server.c
subbu-Inspiron-3558 part_b # cc TCP_reverse_server.c -o tcp_reverse_s
subbu-Inspiron-3558 part_b # ./tcp_reverse_s
connect to client
subbu-Inspiron-3558 part_b # 
```

CLIENT SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
int main()
{
        char senddata[30],reversedata[30];
        int sockfd;
        struct sockaddr_inservaddr;
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=ntohs(20154);
        connect(sockfd,(structsockaddr*)&servaddr,sizeof(servaddr));
        printf("\n enter the data to be send");
        while(fgets(senddata,30,stdin)!=NULL)
        {
                write(sockfd,senddata,30);
                printf("\n data send");
                if(strcmp(senddata,"quit\n") == 0 )
                        break;
                read(sockfd,reversedata,30);
                printf("\n reverse of the given sentence is:%s",reversedata);
                printf("\n");
        }
        exit(0);
}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # vi TCP_reverse_client.c
subbu-Inspiron-3558 part_b # cc TCP_reverse_client.c -o tcp_reverse_c
subbu-Inspiron-3558 part_b # ./tcp_reverse_c

 enter the data to be sendJNTU VIZIANAGARAM

 data send
 reverse of the given sentence is:
MARAGANAIZIV UTNJ
quit

 data sendsubbu-Inspiron-3558 part_b #
```

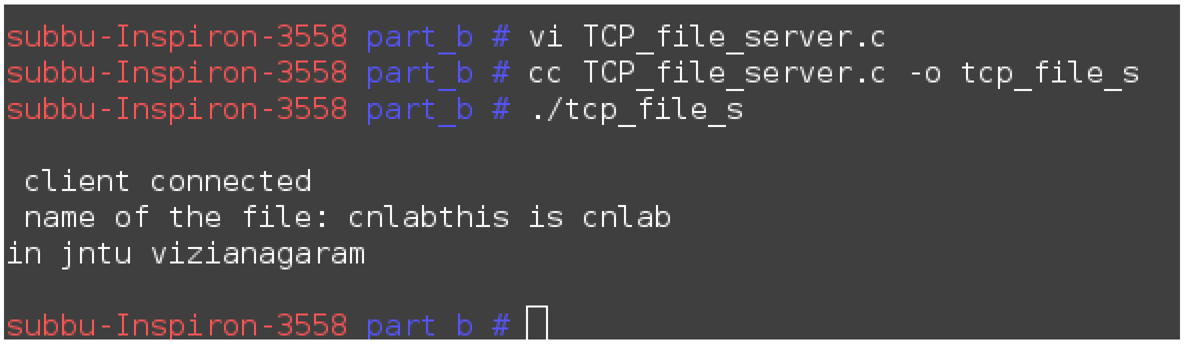
6.AIM: Design TCP client and server application to transfer file.

PROGRAM:

SERVER SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
int main()
{
        inti,j,listenfd,connfd,clilen;
        structsockaddr_inservaddr,cliaddr;
        char s[80],f[80];
        FILE *fp;
        listenfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(5678);
        bind(listenfd,(structsockaddr*)&servaddr,sizeof(servaddr));
        listen(listenfd,1);
        clilen=sizeof(cliaddr);
        connfd=accept(listenfd,(structsockaddr*)&cliaddr,&clilen);
        printf("\n client connected");
        read(connfd,f,80);
        fp=fopen(f,"r");
        printf("\n name of the file: %s",f);
        while(fgets(s,80,fp)!=NULL)
        {
                printf("%s",s);
                write(connfd,s,sizeof(s));
        }
}
```

OUTPUT:

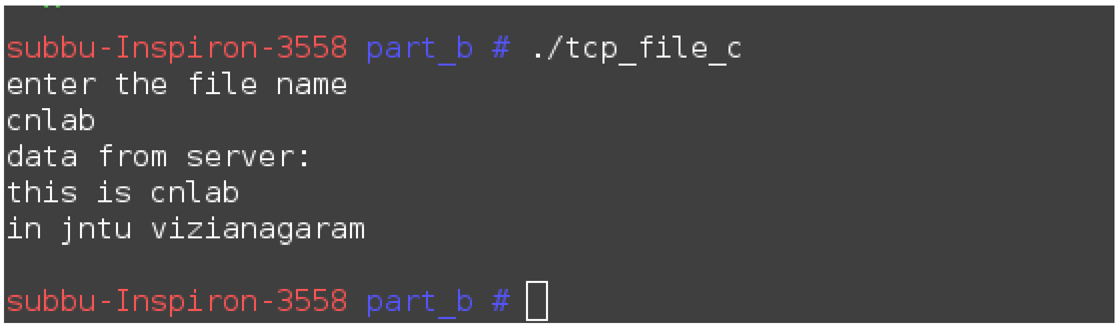


```
subbu-Inspiron-3558 part_b # vi TCP_file_server.c
subbu-Inspiron-3558 part_b # cc TCP_file_server.c -o tcp_file_s
subbu-Inspiron-3558 part_b # ./tcp_file_s

 client connected
 name of the file: cnlabthis is cnlab
in jntu vizianagaram

subbu-Inspiron-3558 part_b #
```

CLIENT SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
int main()
{
        int sockfd,i,j;
        char filename[80],recvline[80];
        struct sockaddr_inservaddr;
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=ntohs(5678);
        inet_pton(AF_INET,"127.0.0.1",&servaddr.sin_addr);
        connect(sockfd,(structsockaddr*)&servaddr,sizeof(servaddr));
        printf("enter the file name\n");
        scanf("%s",filename);
        write(sockfd,filename,sizeof(filename));
        printf("data from server:\n");
        while(read(sockfd,recvline,80)!=0)
        {
                fputs(recvline,stdout);
        }
}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # ./tcp_file_c
enter the file name
cnlab
data from server:
this is cnlab
in jntu vizianagaram

subbu-Inspiron-3558 part_b #
```

8.AIM:-Design a TCP concurrent server to convert a given text into upper case using multiplexing system call “select”.

DESCRIPTION:

- ➢ Client sends message to server using sent functions. Server receives all the messages.
- ➢ The select function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed.
- ➢ The select () and poll () methods can be a powerful tool when you’re multiplexing network sockets. Specifically, these methods will indicate when a procedure will be safe to execute on an open file descriptor without any delays.
- ➢ For instance, a programmer can use these calls to know when there is data to be read on a socket. By delegating responsibility to select() and poll(), you don’t have to constantly check whether there is data to be read.
- ➢ Instead, select() and poll() can be placed in the background by the operating system and woken up when the event is satisfied or a specified timeout has elapsed.
    - int select(int nfds,fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
    - int nfds - The highest file descriptor in all given sets plus one ·
    - fd_set *readfds - File descriptors that will trigger a return when data is ready to be read ·
    - fd_set *writefds - File descriptors that will trigger a return when data is ready to be written to.
    - fd_set *errorfds - File descriptors that will trigger a return when an exception occurs.
    - struct timeval *timeout - The maximum period select() should wait for an event.
- ➢ You can’t modify the fd_set structure by changing its value directly. The only portable way to either set or retrieve the value is by using the provided FD_* macros:
    - FD_ZERO(fd_set *) - Initializes an fd_set to be empty.
    - FD_CLR(int fd, fd_set *) - Removes the associated fd from the fd_set.
    - FD_SET(int fd, fd_set *) - Adds the associated fd to the fd_set.

- FD_ISSET(int fd, fd_set *) - Returns a nonzero value if the fd is in fd_set

PROGRAMS:

SERVER SIDE PROGRAM:

```
        #include<stdio.h>
        #include<netinet/in.h>
        #include<sys/types.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
 #include<sys/select.h>
#include<unistd.h>
#define MAXLINE 20
#define SERV_PORT 7134
Int main(int argc,char **argv)
{
 int i,j,maxi,maxfd,listenfd,connfd,sockfd;
 int nread,client[FD_SETSIZE];
ssize_t n;
fd_set rset,allset;
char line[MAXLINE];
socklen_t clilen;
struct sockaddr_in cliaddr,servaddr;
listenfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_port=htons(SERV_PORT);
bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
listen(listenfd,1);
maxfd=listenfd;
maxi=-1;
for(i=0;i<FD_SETSIZE;i++)
client[i]=-1;
FD_ZERO(&allset);
FD_SET(listenfd,&allset);
for(; ;)
{
rset=allset;
nread=select(maxfd+1,&rset,NULL,NULL,NULL);
if(FD_ISSET(listenfd,&rset))
{
        clilen=sizeof(cliaddr);
        connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&clilen);
        for(i=0;i<FD_SETSIZE;i++)
        if(client[i]<0)
        {
                client[i]=connfd;
                break;
        }
        if(i==FD_SETSIZE)
        {
                 printf("too many clients");
                exit(0);
```

```
            }
            FD_SET(connfd,&allset);
            if(connfd>maxfd)
                        maxfd=connfd;
            if(i>maxi)
            maxi=i;
            if(--nread<=0)
                        continue;
    }
for(i=0;i<=maxi;i++)
{
            if((sockfd=client[i])<0)
                        continue;
            if(FD_ISSET(sockfd,&rset))
            {
                        if((n=read(sockfd,line,MAXLINE))==0)
                        {
                                    close(sockfd);
                                    FD_CLR(sockfd,&allset);
                                    client[i]=-1;
                        }
                        else
                        {
                                    printf("line recieved from the client :%s\n",line);
                                    for(j=0;line[j]!='\0';j++)
                                    line[j]=toupper(line[j]);
                                    write(sockfd,line,MAXLINE);
```

```
                }
                if(--nread<=0)
                        break;
        }
}
}
}
```

OUTPUT:

```
line recieved from the client: what is u r name?
```

CLIENT PROGRAM:

```
#include<netinet/in.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/select.h>
#include<unistd.h>
#define MAXLINE 20
#define SERV_PORT 7134
Int main(int argc,char **argv)
{
        int maxfdp1;
        fd_set rset;
        char sendline[MAXLINE],recvline[MAXLINE];
        int sockfd;
        struct sockaddr_in servaddr;
        if(argc!=2)
        {
                printf("usage tcpcli <ipaddress>");
                return;
        }
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(SERV_PORT);
        inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
        connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
        printf("\n enter data to be send");
        while(fgets(sendline,MAXLINE,stdin)!=NULL)
        {
                write(sockfd,sendline,MAXLINE);
                printf("\n line send to server is %s",sendline);
                read(sockfd,recvline,MAXLINE);
                printf("line recieved from the server %s",recvline);
        }
        exit(0);
}
```

OUTPUT:

```
        Enter data to be send :what is u r name?
        line send to server is : what is u r name?
        line recieved from the server :  WHAT IS U R NAME?
```

9.AIM: Design a TCP concurrent server to echo given set of sentences using poll functions

DESCRIPTION:

- Poll provides functionality that is similar to select, but poll provides additional information when dealing with streams devices.

  #include<poll.h>

  int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);
  returns : count of ready descriptors, 0 on timeout, -1 on error.

- The return value from poll is -1 if an error occurred, 0 if no descriptors are ready before the time expires,otherwise it is the number of descriptors that have a nonzero revents member.
- The number of elements in the array of structures is specified by the nfds argument.
- The timeout argument specifies how long the function is to wait before returning.

```
Structure pollfd

{

        Int fd;

        Short events;

        Short revents;

}
```

PROGRAM:

SERVER SIDE PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

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

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<poll.h>
#include<errno.h>
#define MAXLINE 100
#define SERV_PORT 5939
#define POLLRDNORM 5
#define INFTIM 5
#define OPEN_MAX 5
int main(int argc,char **argv)
{
	int k,i,maxi,listenfd,connfd,sockfd,nready;
	ssize_t n;
	char line[MAXLINE];
	socklen_t clilen;
	struct pollfd client[OPEN_MAX];
	struct sockaddr_in cliaddr,servaddr;
	listenfd=socket(AF_INET,SOCK_STREAM,0);
	bzero(&servaddr,sizeof(servaddr));
	servaddr.sin_family=AF_INET;
	servaddr.sin_port=htons(SERV_PORT);
	servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
	bind(listenfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
	listen(listenfd,5);
	client[0].fd=listenfd;
	client[0].events=POLLRDNORM;
```

```
for(i=1;i<OPEN_MAX;i++)
{
        nready=poll(client,maxi+1,INFTIM);
        if(client[0].revents&POLLRDNORM)
        {
        clilen=sizeof(cliaddr);
        connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&clilen);
        for(i=1;i<OPEN_MAX;i++)
        if(client[i].fd<0)
        {
                client[i].fd=connfd;
                break;
        }
        if(i==OPEN_MAX)
        {
                printf("too many client requests");
                exit(0);
        }
        client[i].events=POLLRDNORM;
        if(i>maxi)
                maxi=i;
        if(--nready<=0)
                continue;
        }
        for(i=1;i<=maxi;i++)
```

```
{
        if((sockfd=client[i].fd)<0)
                continue;
        if(client[i].revents&(POLLRDNORM|POLLERR))
        {
                if((n=read(sockfd,line,MAXLINE))<0)
                {
                        if(errno==ECONNRESET)
                        {
                                close(sockfd);
                                client[i].fd=-1;
                        }
                        else
                                printf("read line error");
                }
                else if(n==0)
                {
                        close(sockfd);
                        client[i].fd=-1;
                }
                else
                {
                        printf("\n data from the client is %s",line);
                        write(sockfd,line,n);
                }
```

```
                                        if(--nready<=0)
                                                break;
                                }
                        }
                }
 }
```

CLIENT SIDE PROGRAM:

```
        #include<stdio.h>
        #include<stdlib.h>
        #include<unistd.h>
        #include<sys/types.h>
        #include<sys/socket.h>
        #include<netinet/in.h>
        #include<poll.h>
        #include<errno.h>
        #define MAXLINE 100
        #define SERV_PORT 5939
        int main(int argc,char **argv)
        {
                int sockfd,fd;
                struct sockaddr_in servaddress;
                char sendline[100],recvline[100];
                int i=0;
                sockfd=socket(AF_INET,SOCK_STREAM,0);
                bzero(&servaddress,sizeof(servaddress));
        servaddress.sin_family=AF_INET;
        servaddress.sin_port=htons(SERV_PORT);
        servaddress.sin_addr.s_addr=inet_addr(argv[1]);
        connect(sockfd,(struct sockaddr*)&servaddress,sizeof(servaddress));
        printf("Enter sentence to send");
        while(fgets(sendline,MAXLINE,stdin)!=NULL)
        {
            write(sockfd,sendline,MAXLINE);
            printf("line send:%s",sendline);
            read(sockfd,recvline,MAXLINE);
            printf("echoed sentence%s",recvline);
        }
        close(sockfd);
        return 0;
    }
```

OUTPUT:

```
Enter the sentence to send: cse
Line send: cse
Echoed sentence: cse
```

10.AIM: Design UDP Client and server application to reverse the given input sentence.

DESCRIPTION:

- The UDP client and server are created with the help of DatagramSocket and Datagram packet classes.
- If the UDP protocol is used at transport, then the unit of data at the transport layer is called a datagram and and not a segment.
- In UDP, no connection is established. It is the responsibility of an application to encapsulate data in datagrams (using Datagram classes) before sending it.
- If TCP is used for sending data, then the data is written directly to the socket (client or server) and reaches there as a connection exists between them.
- The datagram sent by the application using UDP may or may not reach the UDP receiver.

  The system calls used here are:

  ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,struct sockaddr *form,socklen_t *addrlen);

  The client just sends a datagram to the server using the sendto function, which requires the address of the destination as a parameter.

ssize-t sendto(int sockfd const void *buff, size_t nbytes, int flags, const structsockaddr *to, socklen_t addrlen);

recvfrom returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

int bind(int sockfd, const struct sockaddr *myaddr, s ocklen_t addrlen);

-- The bind function assigns a local protocol address to a socket.

#include<strings.h>

void bzero(void *dest,size_t nbytes);

--It sets the specified number of bytes to 0(zero) in the destination. We often use this function to initialize a socket address structure to 0(zero).

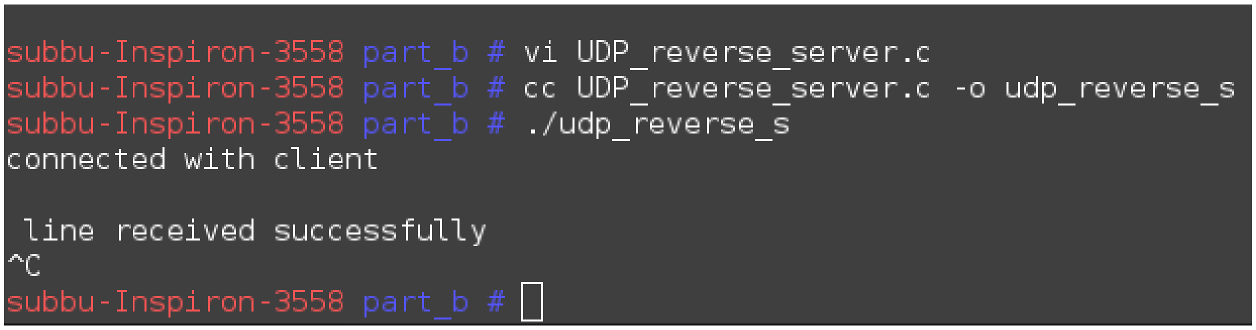
PROGRAM:

SERVER SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
int main()
{
        inti,j,sockfd,clilen;
        size_t n;
        char line[30],recvline[30];
        structsockaddr_inservaddr,cliaddr;
        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
        servaddr.sin_port=htons(7778);
        bind(sockfd,(structsockaddr*)&servaddr,sizeof(servaddr));
        printf("connected with client\n");
        clilen=sizeof(cliaddr);
        while(1)
        {
        if((n=recvfrom(sockfd,line,30,0,(structsockaddr*)&cliaddr,&clilen))==0)
                        break;
```

```
                printf("\n line received successfully");

                line[n-1]='\0';
                j=0;
                for(i=n-2;i>=0;i--)
                {
                        recvline[j++]=line[i];
                }
                recvline[j]='\0';
                sendto(sockfd,recvline,n,0,(structsockaddr*)&cliaddr,clilen);
        }
}
```

OUTPUT:



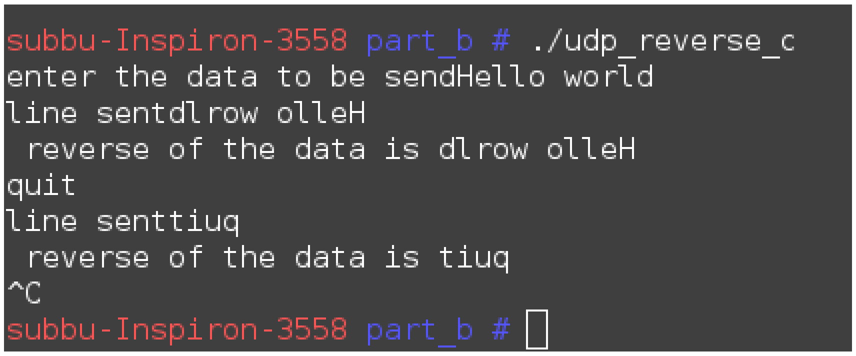
```
subbu-Inspiron-3558 part_b # vi UDP_reverse_server.c
subbu-Inspiron-3558 part_b # cc UDP_reverse_server.c -o udp_reverse_s
subbu-Inspiron-3558 part_b # ./udp_reverse_s
connected with client

 line received successfully
^C
subbu-Inspiron-3558 part_b #
```

CLIENT SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
```

```
#include<netinet/in.h>
int main()
{
        size_t n;
        struct sockaddr_inservaddr;
        char sendline[30],recvline[30];
        int sockfd;

        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=ntohs(7778);

        inet_pton(AF_INET,"127.0.0.1",&servaddr.sin_addr);
        sockfd=socket(AF_INET,SOCK_DGRAM,0);

        printf("enter the data to be send");
        while(fgets(sendline,30,stdin)!=NULL)
        {
        sendto(sockfd,sendline,strlen(sendline),0,(structsockaddr*)&servaddr,sizeof(
servaddr));
                printf("line sent");
                n=recvfrom(sockfd,recvline,30,0,NULL,NULL);
                recvline[n]='\0';
                fputs(recvline,stdout);
                printf("\n reverse of the data is %s",recvline);
                printf("\n");
        }
}
```

OUTPUT:



```
subbu-Inspiron-3558 part_b # ./udp_reverse_c
enter the data to be sendHello world
line sentdlrow olleH
 reverse of the data is dlrow olleH
quit
line senttiuq
 reverse of the data is tiuq
^C
subbu-Inspiron-3558 part_b # 
```

11.AIM: Design UDP Client server to transfer a file.

PROGRAM:
(SERVER SIDE PROGRAM)

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<unistd.h>

int main()

{

        char filename[80],recvline[80];

        intsockfd,clilen;

        structsockaddr_inservaddr,cliaddr;

        FILE *fp;

        sockfd=socket(AF_INET,SOCK_DGRAM,0);

        bzero(&servaddr,sizeof(servaddr));

        servaddr.sin_family=AF_INET;

        servaddr.sin_port=htons(5999);

        bind(sockfd,(structsockaddr*)&servaddr,sizeof(servaddr));

        clilen=sizeof(cliaddr);

        recvfrom(sockfd,filename,80,0,(structsockaddr*)&cliaddr,&clilen);

        printf("data in the file is:\n");

        fp=fopen(filename,"r");

        while(fgets(recvline,80,fp)!=NULL)
        {
                printf("%s",recvline);
        }
        fclose(fp);
}
```

OUTPUT:

```
date in the file is:
hai this is np lab
something intresting
```

CLIENT SIDE PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<unistd.h>
int main()
{
        char filename[80];
        int sockfd;
        struct sockaddr_inservaddr;
        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=ntohs(5999);

        inet_pton(AF_INET,"127.0.0.1",&servaddr.sin_addr);

        printf("enter the file name\n");

        scanf("%s",filename);

        sendto(sockfd,filename,strlen(filename),0,(structsockaddr*)&servaddr,sizeof
(servaddr));

}
```

OUTPUT:

```
enter the file name: npfile
```

12.AIM: Design using poll client server application to multiplex TCP and UDP requests for converting a given text into upper case.

DESCRIPTION:

Poll is used for multiplexing tcp & udp requests.

```
#include<poll.h>

int  poll ( struct  pollfd   *fdarray,  unsigned   long   nfds,  int  timeout);
```

getsockopt and setsockopt Functions:-

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

int   getsockopt (int  sockfd, int level, int optname, void  *optval, socklen_t *optlen);

int setsockopt (int sockfd, int  level, int optname, void  *optval, socklen_t *optlen);
```

Both return : 0 if ok, -1 on error.

- Sockfd from socket descriptor.
- The level specifies the code in the system to interpret the option.
- The optval is a pointer to a variable, can be set true(non-zero) or false(zero).
- Size of the third argument variable.

PROGRAM:

SERVER SIDE PROGRAM:

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/socket.h>

#include<unistd.h>

#include<netinet/in.h>

#define MAXLINE 20
```

```
#define SERV_PORT 8114
int main(int argc,char **argv)
{
        int maxfdp1;
        fd_set rset;
        char sendline[MAXLINE],recvline[MAXLINE];
        int sockfd;
        struct sockaddr_in servaddr;
        if(argc!=2)
        {
            printf("usage tcpcli <ipaddress>");
            return;
        }
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        bzero(&servaddr,sizeof(servaddr));
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(SERV_PORT);
        inet_pton(AF_INET,argv[1],&servaddr.sin_addr);
        connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
        printf("\nenter data to be send:");
        while(fgets(sendline,MAXLINE,stdin)!=NULL)
        {
            write(sockfd,sendline,MAXLINE);
            printf("\nline send to server :%s ",sendline);
            read(sockfd,recvline,MAXLINE);
            printf("line received from the server : %s",recvline);
        }
        exit(0);
}
```

OUTPUT:

```
$./ser
line received from client: gec-cse
```

CLIENT SIDE PROGRAM:

```
#include<stdio.h>
#include<netinet/in.h>
#include<sys/types.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/select.h>
#include<unistd.h>
#define MAXLINE 20
#define SERV_PORT 8114
int main(int argc,char **argv)
{
    int i,j,maxi,maxfd,listenfd,connfd,sockfd;
    int nready,client[FD_SETSIZE];
    ssize_t n;
    fd_set rset,allset;
    char line[MAXLINE];
    socklen_t clilen;
    struct sockaddr_in cliaddr,servaddr;
```

```
listenfd=socket(AF_INET,SOCK_STREAM,0);
bzero(&servaddr,sizeof(servaddr));

servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(SERV_PORT);

bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));

listen(listenfd,1);  maxfd=listenfd;  maxi=-1;

for(i=0;i<FD_SETSIZE;i++)

        client[i]=-1;

FD_ZERO(&allset);

FD_SET(listenfd,&allset);

for(;;)

{

        rset=allset;

        nready=select(maxfd+1,&rset,NULL,NULL,NULL);
        if(FD_ISSET(listenfd,&rset))

        {

                clilen=sizeof(cliaddr);

                connfd=accept(listenfd,(struct sockaddr *)&cliaddr,&clilen);

                for(i=0;i<FD_SETSIZE;i++)

                if(client[i]<0)

                {

                        client[i]=connfd;

                        break;

                }

                if(i==FD_SETSIZE)

                {
```

```
                printf("too many clients");
                exit(0);
            }
            FD_SET(connfd,&allset);
            if(connfd>maxfd)
                maxfd=connfd;
            if(i>maxi)
                maxi=i;
            if(--nready<=0)
                continue;
        }
        for(i=0;i<=maxi;i++)
        {
            if((sockfd=client[i])<0)
                continue;
            if(FD_ISSET(sockfd,&rset))
            {
                if((n=read(sockfd,line,MAXLINE))==0)
                {
                    close(sockfd);
                    FD_CLR(sockfd,&allset);
                    client[i]=-1;
                }
                else
                {
```

```
                                        printf("line received from client:%s\n",line);
                                        for(j=0;line[j]!='\0';j++)
                                        line[j]=toupper(line[j]);
                                        write(sockfd,line,MAXLINE);
                                }
                                if(--nready<=0)
                                        break;
                        }
                }
        }
 }
```

OUTPUT:

```
$./cli localhost
Enter data to be send: gec-cse
line send to server : gec-cse
line received from the server : GEC-CS
```

13.AIM: Design a RPC application to add and subtract a given pair of integers

DESCRIPTION:

- The client calls a local procedure, called the clients stub. It appears to the client that the client stub is the actual server procedure that it wants to call.
- The purpose of the stub is to package up the arguments to the remote procedure, possibly put them into some standard format and then build one or more network messages.
- The packaging of the clients arguments into a network message is termed marshaling.
- These network messages are sent to the remote system by the client stub. This requires a system call into the kernel.
- The network messages are transferred to the remote system. Either a connection oriented or a connectionless protocol is used.
- A server stub procedure is waiting on the remote system for the client's request. It unmarshals the arguments from the network messages and possibly converts them.
- The server stub executes a local procedure call to invoke the actual server function, passing it the arguments that it received in the network messages from the client stub.
- When the server procedure is finished, it returns to the server stub, returning whatever its return values are.
- The server stub converts the return values, if necessary and marshals them into one or more network messages to send back to the client stub.
- To message get transferred back across the network to client stub.
- The client stub reads the network message from the local kernel.
- After possibly converting the return values the client stub finally returns to the client functions this appears to be a normal procedure returns to the client.

PROGRAM:

SERVER SIDE PROGRAM:

```
#include "rpctime.h"

#include <stdio.h>

#include <stdlib.h>

#include <rpc/pmap_clnt.h>
```

```
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
static void rpctime_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        union
        {
                int fill;
        } argument;
        char *result;
        xdrproc_t _xdr_argument, _xdr_result;
        char *(*local)(char *, struct svc_req *);
        switch (rqstp->rq_proc)
        {
                case NULLPROC:
                 (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
                return;
                case GETTIME:
                _xdr_argument  =  (xdrproc_t)  xdr_void;  _xdr_result  =  (xdrproc_t)
xdr_long;
                local = (char *(*)(char *, struct svc_req *)) gettime_1_svc;
                break;
        default: svcerr_noproc (transp);
        return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument))
    {
        svcerr_decode (transp);
        return;
    }
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result,
result))
    {
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument))
    {
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
    }
    return;
}
int main (int argc, char **argv)
{
    register SVCXPRT *transp;
    pmap_unset (RPCTIME, RPCTIMEVERSION);
```

```
	transp = svcudp_create(RPC_ANYSOCK);
	if (transp == NULL)
	{
		fprintf (stderr, "%s", "cannot create udp service.");
		exit(1);
	}
	if (!svc_register(transp, RPCTIME, RPCTIMEVERSION, rpctime_1, IPPROTO_UDP))
	{
		fprintf (stderr, "%s", "unable to register (RPCTIME, RPCTIMEVERSION, udp).");
		exit(1);
	}
	transp = svctcp_create(RPC_ANYSOCK, 0, 0);
	if (transp == NULL)
	{
		fprintf (stderr, "%s", "cannot create tcp service.");
		exit(1);
	}
	if (!svc_register(transp, RPCTIME, RPCTIMEVERSION, rpctime_1, IPPROTO_TCP))
	{
		fprintf (stderr, "%s", "unable to register (RPCTIME, RPCTIMEVERSION, tcp).");
		exit(1);
	}
	svc_run ();
```

```
        fprintf (stderr, "%s", "svc_run returned");
        exit (1);
}
```

CLIENT SIDE PROGRAM:

```
#include "rpctime.h"
void rpctime_1(char *host)
{
        CLIENT *clnt;
        long *result_1;
        char *gettime_1_arg;
        #ifndef DEBUG
        clnt = clnt_create (host, RPCTIME, RPCTIMEVERSION, "udp");
        if (clnt == NULL)
        {
                clnt_pcreateerror (host);
                exit (1);
        }
        #endif /* DEBUG */
        result_1 = gettime_1((void*)&gettime_1_arg, clnt);
        if (result_1 == (long *) NULL)
        {
                clnt_perror (clnt, "call failed");
        }
        else
                printf("%d |%s", *result_1, ctime(result_1));
        #ifndef DEBUG
        clnt_destroy (clnt);
        #endif /* DEBUG */
}
int main (int argc, char *argv[])
{
        char *host;
        if (argc < 2)
        {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        rpctime_1 (host);
        exit (0);
        }
        rpctime_cntl.c
        #include <memory.h> /* for memset */
        #include "rpctime.h"
        static struct timeval TIMEOUT = { 25, 0 };
        long * gettime_1(void *argp, CLIENT *clnt)
        {
                static long clnt_res;
                memset((char *)&clnt_res, 0, sizeof(clnt_res));
                if (clnt_call (clnt, GETTIME, (xdrproc_t) xdr_void, (caddr_t) argp,
(xdrproc_t) xdr_long, (caddr_t) &clnt_res, TIMEOUT) != RPC_SUCCESS)
```

```
          {
                    return (NULL);
          }
          return (&clnt_res);
}
```

OUTPUT:

```
$ ./server
$./client 10.0.0.1   10  5
Add = 10 + 5 = 15
Sub = 10 – 5 = 5
```