NETWORK PROGRAMMING LAB

INDEX

Sl.No.	TITLE	Page No.
1	System Requirements	3
2	Lab Objectives	4
3	Guidelines to Students	5
	LIST OF PROGRAMS	
1	Implement the following forms of IPC. a) Pipes b) FIFO	6
2	Implement file transfer using Message Queue form of IPC.	14
3	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.	18
4	Design TCP iterative Client and Server application to reverse the given input sentence.	23
5	Design TCP concurrent Client and Server application to reverse the given input sentence.	23
6	Design TCP Client and Server application to transfer file.	29
7	Design a TCP concurrent Server to convert a given text into upper case using multiplexing system call “select”.	34
8	Design a TCP concurrent Server to echo given set of sentences using Poll functions.	42
9	Design UDP Client and Server application to reverse the given input sentence.	49
10	Design UDP Client Server to transfer a file.	56
11	Design using Poll Client Server application to multiplex TCP and UDP requests for converting a given text into upper case.	60
12	Design a RPC application to add and subtract a given pair of integers.	67

	ADD ON PROGRAMS	
1	Program to determine the host ByteOrder	**75**
2	Program to set and get socket options	**77**

Reference Books:

1.Advance Unix Programming Richard Stevens, Second Edition Pearson Education

2. Advance Unix Programming, N.B. Venkateswarlu, BS Publication

Signature of the Faculty **Signature of the HOD**

System Requirements

Recommended Systems/Software Requirements:

→ Intel based desktop PC with minimum of 166 MHZ or faster processor with at least 64 MB RAM and 100 MB free disk space LAN Connected

→ Any flavor of Unix / Linux

Lab Objectives

1) To write, execute and debug c programs which use Socket API.
2) To understand the use of client/server architecture in application development
3) To understand how to use TCP and UDP based sockets and their differences.
4) To get acquainted with unix system internals like Socket files, IPC structures.
5) To Design reliable servers using both TCP and UDP sockets

GUIDELINES TO STUDENTS

- Equipment in the lab for the use of student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage is caused is punishable.
- Students are required to carry their observation / programs book with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- Lab can be used in free time / lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- Lab records need to be submitted on or before date of submission.
- Students are not supposed to use cd's and pen drives.

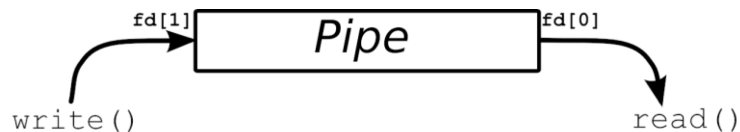
WEEK 1
AIM: Implement the following forms of IPC

a) Pipes **b) FIFO**

a) Pipes:

DESCRIPTION:

There is no form of IPC that is simpler than pipes, Implemented on every flavor of UNIX.



Basically, a call to the **pipe()** function returns a pair of file descriptors. One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end. Anything can be written to the pipe, and read from the other end in the order it came in. On many systems, pipes will fill up after you write about 10K to them without reading anything out.

The following example shows how a pipe is created, reading and writing from pipe.

A pipe provides a one-way flow of data.

A pipe is created by the pipe system call. int pipe (int *filedes) ;

Two file descriptors are returned- filedes[0] which is open for reading , and filedes[1] which is open for writing.

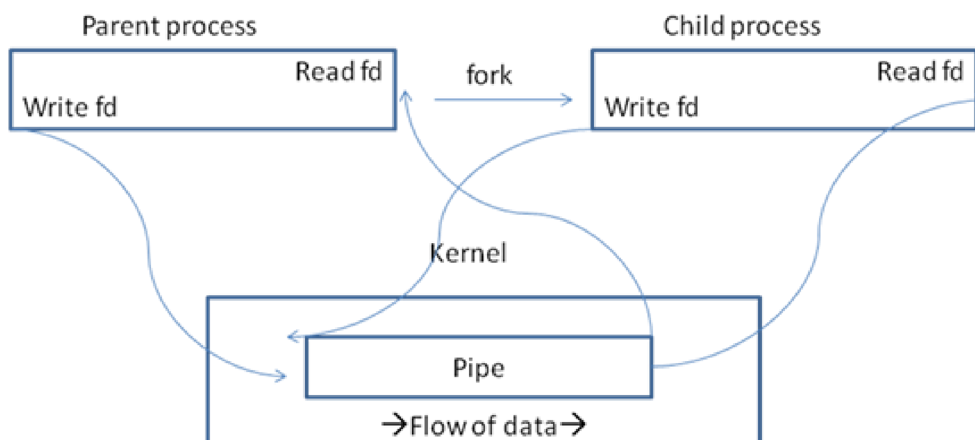


Fig: Pipe in a single process, immediately after fork

Pipes are typically used to communicate between two different processes in the following way. First, a process creates a pipe and then forks to create a copy of itself, as shown above figure.

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.

1) The first return value in the parent process is the process ID of the newly created child process.
2) The second return value in the child process is zero.
   If the fork system call is not successful, -1 is returned

Pseudo code:

START

Store any message in one character array (char *msg=”Hello world”)

Declare another character array

Create a pipe by using pipe() system call

Create another process by executing fork() system call

In parent process use system call write() to write message from one process to another process.

In child process display the message.

END

/* **CREATION OF A ONEWAY PIPE IN A SINGLE PROCESS**. */

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
main()
{
	int pipefd[2],n;
	char buff[100];
	pipe(pipefd);
	printf("\nreadfd=%d",pipefd[0]);
	printf("\nwritefd=%d",pipefd[1]);
```

```
        write(pipefd[1],"helloworld",12);
        n=read(pipefd[1],buff,sizeof(buff));
        printf("\n size of the data%d",n);
        printf("\n data from pipe:%s",buff);
}
```

OUTPUT:

```
readfd=3
writefd=4
 size of the data-1
```

/* **CREATION OF A ONEWAY PIPE BETWEEN TWO PROCESS** */

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
main()
{
        int pipefd[2],n,pid;
        char buff[100];
        pipe(pipefd);
        printf("\n readfd=%d",pipefd[0]);
        printf("\n writefd=%d",pipefd[1]);
        pid=fork();
        if(pid==0)
        {
                close(pipefd[0]);
                printf("\n CHILD PROCESS SENDING DATA\n");
                write(pipefd[1],"hello world",12);
        }


        else
        {
                close(pipefd[1]);
                printf("PARENT PROCESS RECEIVES DATA\n");
                n=read(pipefd[0],buff,sizeof(buff));
                printf("\n size of data%d",n);
                printf("\n data received from child throughpipe:%s\n",buff);
        }
}
```

OUTPUT

readfd=3
writefd=4
CHILD PROCESS SENDING DATA
writefd=4PARENT PROCESS RECEIVES DATA

/***CREATION OF A TWOWAY PIPE BETWEEN TWO PROCESS***/

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
main()
{
        int p1[2],p2[2],n,pid;
        char buf1[25],buf2[25];
        pipe(p1);
        pipe(p2);
        printf("\n readfds=%d %d\n",p1[0],p2[0]);
        printf("\n writefds=%d %d\n",p1[1],p2[1]);
        pid=fork();
        if(pid==0)
        {
                close(p1[0]);
                printf("\n CHILD PROCESS SENDING DATA\n");
                write(p1[1],"where is GEC",25);
                close(p2[1]);
                read(p2[0],buf1,25);
                printf(" reply from parent:%s\n",buf1);
                sleep(2);
        }
        else
        {
                close(p1[1]);
                printf("\n parent process receiving data\n");
                n=read(p1[0],buf2,sizeof(buf2));
                printf("\n data received from child through pipe:%s\n",buf2);
                sleep(3);
                close(p2[0]);
                write(p2[1]," in gudlavalleru",25);
                printf("\n reply send\n");
        }
}
```

OUTPUT:

```
readfds=3 5
writefds=4 6

CHILD PROCESS SENDING DATA

parent process receiving data
data received from child through pipe:where is GEC

reply send
reply from parent: in gudlavalleru
```

b) FIFO:

DESCRIPTION:

A FIFO ("First In, First Out") is sometimes known as a *named pipe*. That is, it's like a pipe, except that it has a name! In this case, the name is that of a file that multiple processes can **open()** and read and write to.

This latter aspect of FIFOs is designed to let them get around one of the shortcomings of normal pipes: you can't get one end of a normal pipe that was created by an unrelated process. See, if I run two individual copies of a program, they can both call **pipe()** all they want and still not be able to communicate to one another. (This is because you must **pipe()**, then **fork()** to get a child process that can communicate to the parent via the pipe.) With FIFOs, though, each unrelated process can simply **open()** the pipe and transfer data through it.

Since the FIFO is actually a file on disk, we have to call **mknod()** with the proper arguments create it.. Here is a **mknod()** call that creates a FIFO:

```
Int mknod ( char *pathname, int mode, int dev ) ;
```

Pathname = is the name of the fifo file . Mode = The mode argument specifies the file mode access mode and is logically or' ed with the S_IFIFO flag.

mknod() returns -1 if unsuccessful and 0 (zero) otherwise

```
mknod("myfifo", S_IFIFO | 0644 , 0);
```

In the above example, the FIFO file will be called "*myfifo*". The second argument is the creation mode, which is used to tell **mknod()** to make a FIFO (the S_IFIFO part of the OR) and sets access permissions to that file (octal 644, or rw-r--r--) which can also be set by ORing together macros from *sys/stat.h*. Finally, a device number is passed. This is ignored

when creating a FIFO, so you can put anything you want in there. Once the FIFO has been created, a process can start up and open it for reading or writing using the standard **open()** system call.

Note: a FIFO can also be created from the command line using the Unix**mknod** command.

Here is a small example of FIFO. This is a simulation of Producers and Consumers Problem. Two programs are presented Producer.c and Consumer.c where Producer writes into FIFO and Consumer reads from FIFO.

Pseudo code for FIFO SERVER:

START

Create a fifo is created by the mknod system call.
Initialize a fifo and set its attributes.
wait for the client request, on request establish a connection using accept function.
fork a child process.
Read the message from the client through the connection.
Display the client I message.
send an acknowledgement message to the client .
Exit the child process.

END

Pseudo code for FIFO CLIENT:

START

Initialize the fifo and set its attributes.
sent message to the server.

END

/* INTERPROCESS COMMUNICATION THROUGH FIFO BETWEEN CLIENT AND SERVER */

PROGRAM

SERVER Program

```
#include<stdio.h>
#include<ctype.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
#include<string.h>
main()
{
	int wrfd,rdfd,n,d,ret_val,count;
	char buf[50];
	/*create the first named pipe */
	ret_val=mkfifo("np1",0666);
	/*create the second named pipe */
	ret_val=mkfifo("np2",0666);
	/*open the first named pipe for reading*/
	rdfd=open("np1",O_RDONLY);
	/*open the second named pipe for writing*/
	wrfd=open("np2",O_WRONLY);

	/*read from the first pipe*/
	n=read(rdfd,buf,50);
	buf[n]='\0';//end of line
	printf("full duplex server:read from the pipe:%s\n",buf);

	    /*convert the string to upper class*/
	count=0;
	while(count<n)
	{
			buf[count]=toupper(buf[count]);
			count++;
	}
	/*write the convertor string back to second  pipe*/

	   write(wrfd,buf,strlen(buf));
}
```

FIFO SERVER OUT PUT:

```
[cse09_a3@localhost ~]$ cc server.c -o ser
[cse09_a3@localhost ~]$ ./ser
full duplex server:read from the pipe: hello
```

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>
main()
{
	int wrfd,rdfd,n;
	char buf[50],line[50];
	/*open the first named pipe for writing*/
	wrfd=open("np1",O_WRONLY);
	/*create the second named pipe for reading */
	rdfd=open("np2",O_RDONLY);
	/*write to the pipe*/
	printf("enter line of text");
	gets(line);
	write(wrfd,line,strlen(line));
	/*read from the  pipe*/
	n=read(rdfd,buf,50);
	buf[n]='\0';//end of line
	printf("full duplex client:read from the pipe:%s\n",buf);

}
```

FIFO CLIENT OUT PUT

```
[cse09_a3@localhost ~]$ ./cli
enter line of text hello
full duplex client:read from the pipe: HELLO
```

WEEK-2

AIM: Implement file transfer using Message Queue form of IPC

DESCRIPTION:

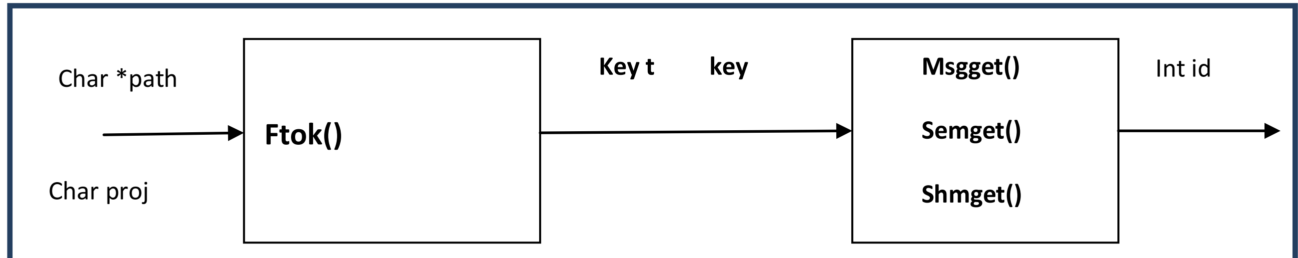


Fig:Generate IPC ids using ftok

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

Key_t   ftok( char   *pathname,   char   proj ) ;
```

The file <sys/types.h> defines the key_t datatype, which is typically a 32-bit integer.

Ftok converts a pathname and a project identifier to a system V IPC key

- ✓ System V IPC keys are used to identify message queues, shared memory, and semaphores.
- ✓ If the pathname does not exist, or is not accessible to the calling process, ftok returns -1.
- ✓ Once the pathname and proj are agreed on by the client and server, then both can call the ftok function to convert these into the same IPC key.

Msgget Ssystem call:

A new message queue is created or an existing message Queueis accesed with the msgget system call

```
Int msgget (key_t key, int msgflag);
```

The value returned by msgget is the message queue identifier, msqid, or -1 if an error occurred.

msgsnd system call:

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

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

msgrcv system call:

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

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

msgctl system call:

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

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

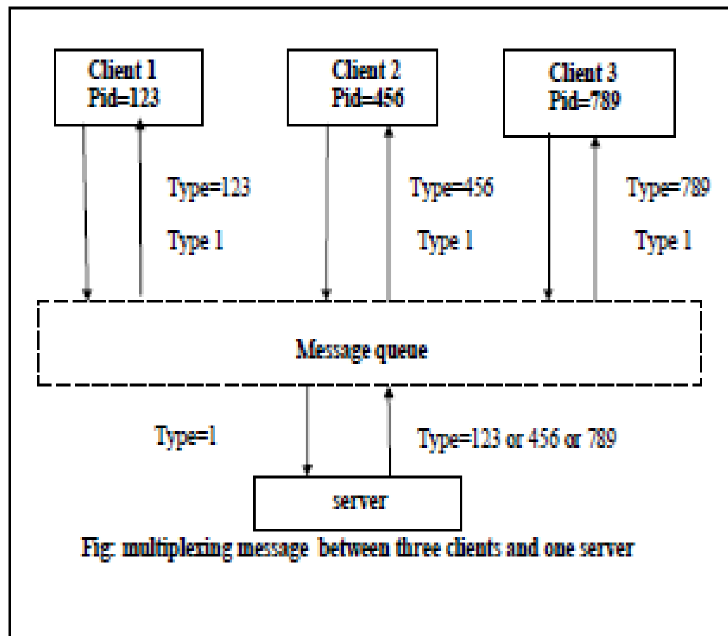


Fig: multiplexing message between three clients and one server

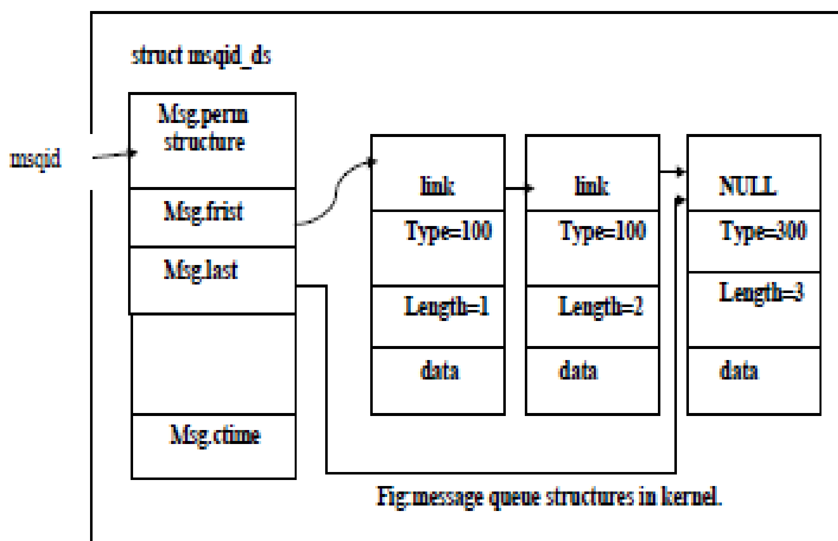


Fig:message queue structures in kernel.

Pseudo code:

START

Initialize character array with string.

Create a message Queue by using msgget() system call.

send a message with msgsnd() system call.

Receive the message by using msgrcv() system call.

Print the message.

kill the message queue using msgctl() system call.

END

Message queues are implemented as linked lists of data stored in shared memory. The message queue itself contains a series of data structures, one for each message, each of which identifies the address, type, and size of the message plus a pointer to the next message in the queue.

To allocate a queue, a program uses the msgget() system call. Messages are placed in the queue by msgsnd() system calls and retrieved by msgrcv() . Other operations related to managing a given message queue are performed by the msgctl() system call.

PROGRAM

SERVER Program

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
main()
{
        int msqid,l;
         struct
         {
                long mtype;
                char fname[20];
        }msgbuf;

        msqid=msgget((key_t)10,IPC_CREAT|0666);
```

```
        msgrcv(msqid,&msgbuf,sizeof(msgbuf),0,0);
        printf("\n Received filename %s \n",msgbuf.fname);
}
```

CLIENT PROGRAM

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
main()
{
        int msqid,l;
        struct
        {
                long mtype;
                char fname[20];
        }msgbuf;

        msqid=msgget((key_t)10,IPC_CREAT|0666);
        printf("Enter file name");
        scanf("%s",msgbuf.fname);
        msgsnd(msqid,&msgbuf,sizeof(msgbuf),0);
}
```

MESSAGE SENDER OUTPUT:

```
[student@localhost ~]$ cc msgsndQ.c -o msgsndQ
[student@localhost ~]$ ./msgsndQ pipe.c
msgid=0
Send Msg Success : return 0
```

MESSAGE RECIEVER OUTPUT:

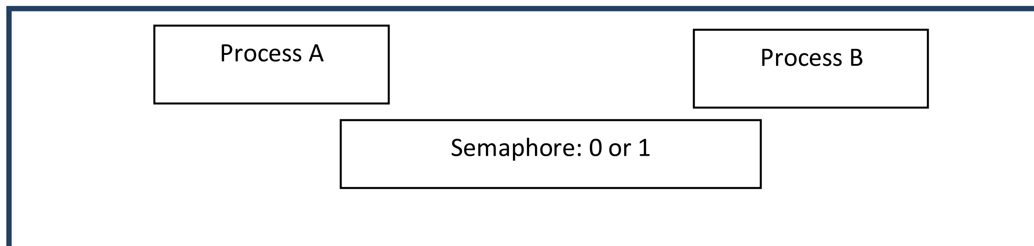
```
[student@localhost ~]$ cc msgrecvQ.c -o msgrecvQ
[student@localhost ~]$ ./msgrecv recvpipe.c
msgid=0
Rec Bytes : 415
```

WEEK-3

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.

DECRIPTION:

Semaphores are synchronization primitive. If we have one resource say a file that is shared ,then the valid semaphore values are zero and one. Semaphore is used to provide resource synchronization between different processes the actual semaphore value must be stored in the kernel.



To obtained a resource that is controlled by a semaphore a process needs to test its current value, and if the current value is greater than zero, decrement the value by one.

0=wait 1=enter

If the current value is zero the processes must wait until the value it greater than zero.

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

```
Key_t  ftok(char *pathname,  char  proj );
```

Pathname =name of a file, name of a server or name of a client.
Project identifier =name of the IPC channel.

Semget:a semaphore is created or an existing semaphore is accesed with the segment system call.

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

Semctl system call:

```
int semctl(int  semid,  int  semnum, int  cmd, union  sem arg);
        union semnum
        {
            int val;
            struct semid_ds *buff;
            ushort   *array;
         }
        arg;
```

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

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

The pointer opstr points to an array of the following structure.

```
Struct sembuf
{
      Ushort  sem_num;
      Short   sem_op;
      Short   sem_flg;
};
```

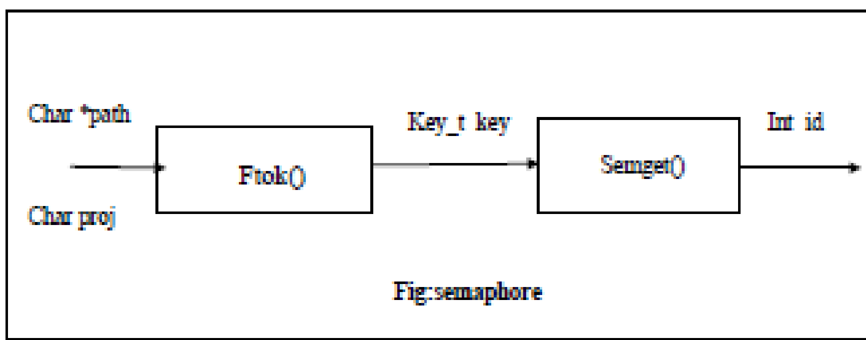


Fig:semaphore

semid

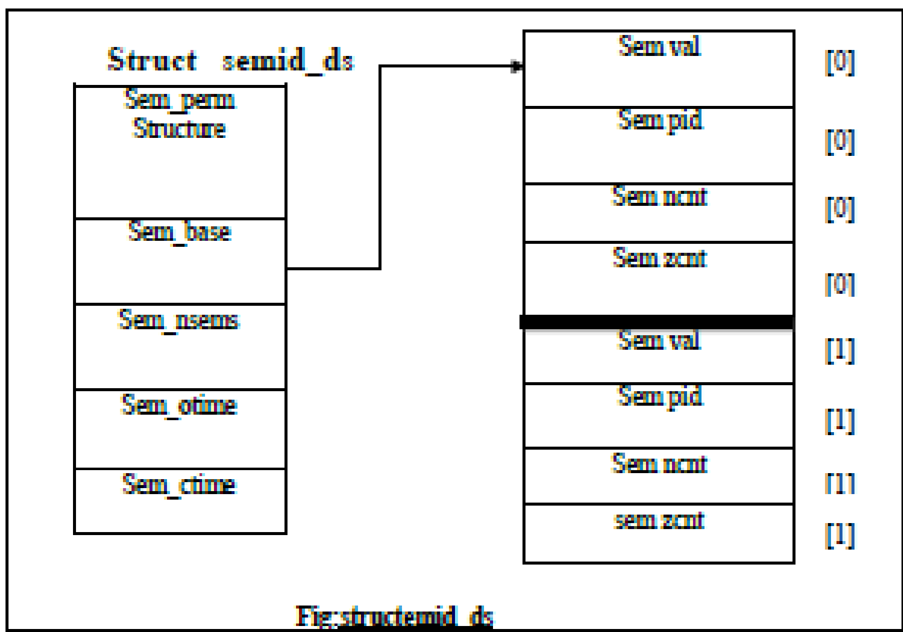


Fig:structemid_ds

Shared memory concept using Semaphores

Shared memory is perhaps the most powerful of the SysV IPC methods, and it is the easiest to implement. As the name implies, a block of memory is shared between processes. Listing 7 shows a program that calls fork(2) to split itself into a parent process and a child process, communicating between the two using a shared memory segment.

Pseudo code:

START

Create a shared memory using mhmget().

store integer value in shared memory. (shmat())

create a child process using fork().

get a semaphore on shared memory using semget().

increase the value of shared variable

release the semaphore

repeat step 4,5,6 in child process also.

remove shared memory.

END

PROGRAM

```
#include<sys/stat.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/shm.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<string.h>
#define SIZE 10
int *integer=0;
main()
{
        int shmid;
        key_t key_10;
        char *shm;
        int semid,pid;
        shmid=shmget((key_t)10,SIZE,IPC_CREAT|0666);
        shm=shmat(shmid,NULL,0);
        semid = semget(0x20,1,IPC_CREAT|0666);
        integer=(int *)shm;
```

```
        pid=fork();
        if(pid==0)
        {
                int i=0;
                while(i<10)
                {
                        sleep(2);
                        printf("\n child process use shared memory");
                        accessmem(semid);
                        i++;
                }
        }
        else
        {
                int j=0;
                while(j<10)
                {
                        sleep(j);
                        printf("\n parent versus shared memory");
                        accessmem(semid);
                        j++;
                }
        }
        shmctl(semid,IPC_RMID,0);
}
int accessmem(int semid)
{
        struct sembuf sop;
        sop.sem_num=0;
        sop.sem_op=-1;
        sop.sem_flg=0;
        semop(semid,&sop,1);
        (*integer)++;
        printf("\t integer variable=%d",(*integer));
        sop.sem_num=0;
        sop.sem_op=1;
        sop.sem_flg=0;
        semop(semid,&sop,1);
}
```

OUTPUT:

parent process uses shared memory   Integer variable=631

parent process uses shared memory   Integer variable=633

child process uses shared memory   Integer variable=632

parent process uses shared memory Integer variable=634

parent process uses shared memory Integer variable=636

parent process uses shared memory Integer variable=638

child process uses shared memory Integer variable=635

child process uses shared memory Integer variable=641

child process uses shared memory Integer variable=645

child process uses shared memory Integer variable=649

WEEK-4 &5

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

DECRIPTION:

Socket function:

```
#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
AF_LOCAL	unix domain protocol
AF_ROUTE	routing sockets
AF_KEY	key socket

Type	Description
SOCK_STREAM	Stream description
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

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

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

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

Bind function: The bind function assigns a local protocol address to a socket.

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

Bzero: 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).

```
#include<strings.h>
void bzer(void  *dest,size_t  nbytes);
```

Memset: It sets the specified number of bytes to the value c in the destination.

```
#include<string.h>
void  *memset(void  *dest, int  c, size_t  len);
```

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

```
#include<unistd.h>
int close(int  sockfd);
```

Return 0 if ok, -1 on error.

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

```
int listen(int sockfd, int backlog);
```

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

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

IPv4 Socket Address Structure:

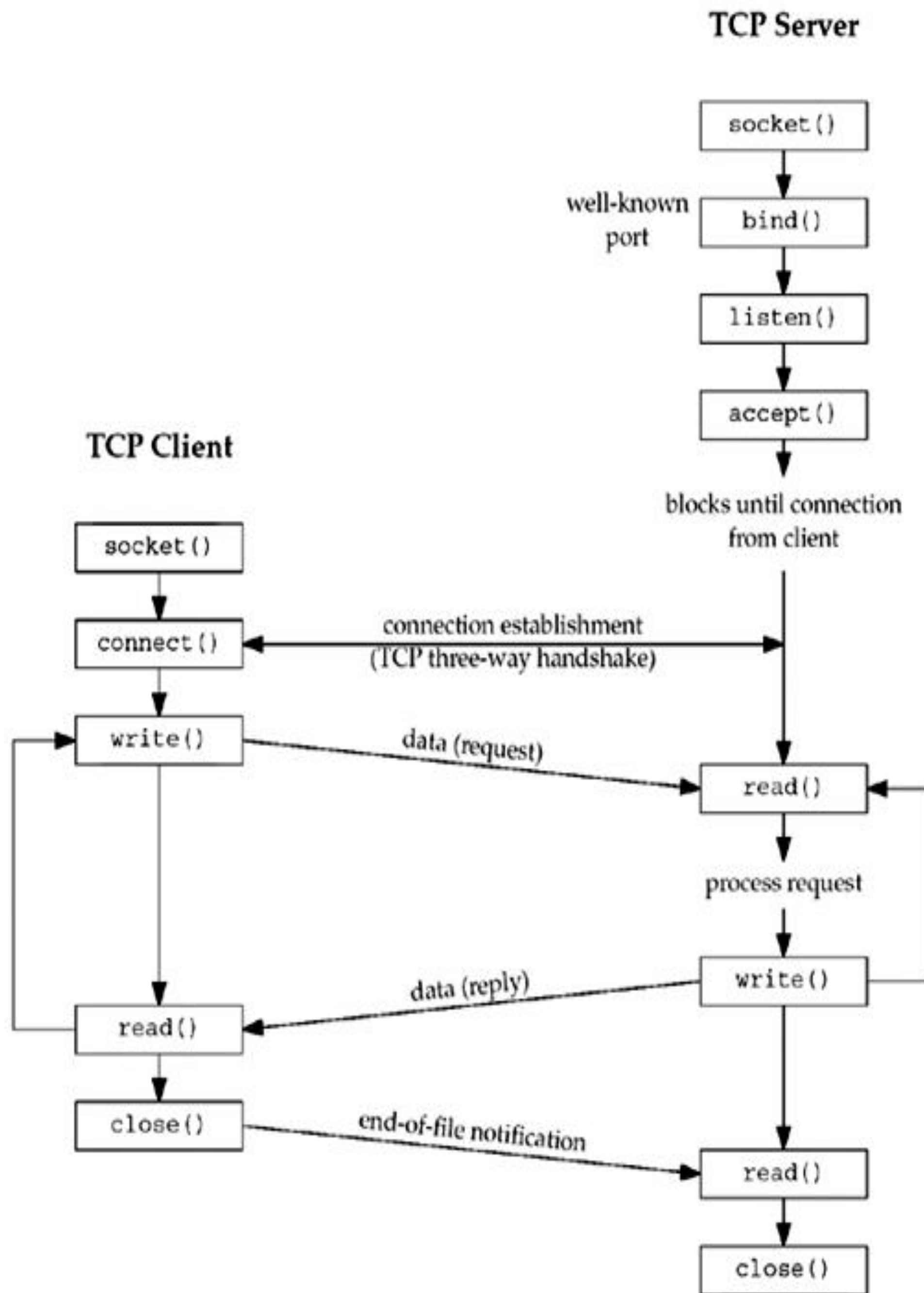
An IPv4 socket address structure, commonly called an " Internet socket address structure, " is named sockaddr_in and defined by including the **<netinet/in.h>** header.

```
struct in_addr
 {
         in_addr_t    s_addr;        /* network byte ordered */
};
struct sockaddr_in
{
        uint8_t          sin_len;                /* length of structure(16) */
        sa_family_t    sin_family;                   /* AF_INET */
        in_port_t       sin_port;             /* 16-bit TCP or UDP port number*/
                                              /* network byte ordered */
       struct in_addr     sin_addr;                  /* 32-bit IPv4 address */
                                              /*newtork byte ordered */
       char  sin_zero[8];                     /* unused */
};
```

Address Conversion functions

```
#include<netinet/in.h>
Uint16_t      htons( uint16_t    host16bitvalue);
Uint32_t      htonl( uint32_t     host32bitvalue);
Uint16_t      ntohs( uint16_t     net16bitvalue);
Uint32_t      ntohl( uint32_t      net32bitvalue);
```

Socket functions for elementary TCP client/server



Pseudo code:

START

Client sends message to server using sent functions.

Server receives all the messages, server ignores all the consonants in the message.

All the vowels in the message are converted into upper case.

Server returns the entire message to clients (with toggled vowel cases).

END

For example: "This is a test and sample message." to server will be sent back to client as "ThIs Is A tEst And sAmplE mEssAgE."

When client closes the connection server should close the communication with that client (socket). And once again wait for new clients to connect. Server program never exits.

Using fork function rewrite the programs, such that this server can handle multiple client connections at one time. To test this you need to run simultaneously multiple copies of client executions. Please log on server machine number of clients it is handled at **this** time.

PROGRAM

CLIENTPROGRAM

```
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#define MAXLINE 20
#define SERV_PORT 5777
main(int argc,char *argv)
{
	char sendline[MAXLINE],revline[MAXLINE];
	int sockfd;
	struct sockaddr_in servaddr;

	sockfd=socket(AF_INET,SOCK_STREAM,0);
	bzero(&servaddr,sizeof(servaddr));
	servaddr.sin_family=AF_INET;
	servaddr.sin_port=ntohs(SERV_PORT);
	connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
	printf("\n enter the data to be send");
```

```
        while(fgets(sendline,MAXLINE,stdin)!=NULL)
        {
                write(sockfd,sendline,strlen(sendline));
                printf("\n line send");
                read(sockfd,revline,MAXLINE);
                printf("\n reverse of the given sentence is : %s",revline);
                printf("\n");
        }
        exit(0);
}
```

SERVER PROGRAM

```
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#define MAXLINE 20
#define SERV_PORT 5777
main(int argc,char *argv)
{
        int i,j;
        ssize_t n;
        char line[MAXLINE],revline[MAXLINE];
        int listenfd,connfd,clilen;
        struct sockaddr_in servaddr,cliaddr;

        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);

        for( ; ; )
        {
          clilen=sizeof(cliaddr);
          connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&clilen);
          printf("connect to client");
          while(1)
          {
                if((n=read(connfd,line,MAXLINE))==0)
                    break;
```

```
                line[n-1]='\0';
                j=0;

                for(i=n-2;i>=0;i--)
                        revline[j++]=line[i];
                revline[j]='\0';
                write(connfd,revline,n);
        }
    }
}
```

OUTPUT

Enter the data to be send: cse
Line send
Reverse of the given sentence: esc

WEEK-6

AIM: Design TCP client and server application to transfer file

DESCRIPTION:

Socket function:

```
#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
AF_LOCAL	unix domain protocol
AF_ROUTE	routing sockets
AF_KEY	key socket

Type	Description
SOCK_STREAM	Stream description
SOCK_DGRAM	Datagram socket
SOCK_RAW	Raw socket

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

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

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

Bind function: The bind function assigns a local protocol address to a socket.

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

Bzero: 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).

```
#include<strings.h>
void bzer(void  *dest,size_t  nbytes);
```

Memset: It sets the specified number of bytes to the value c in the destination.

```
#include<string.h>
void  *memset(void  *dest, int  c, size_t  len);
```

Close function: The normal UNIX close function is also used to close a socket and terminate a TCP connection.

```
#include<unistd.h>
int close(int  sockfd);
```

Return 0 if ok, -1 on error.

Listen function: The second argument to this function specifies the maximum number of connection that the kernel should queue for this socket.

```
int listen(int sockfd, int backlog);
```

Accept function: The cliaddr and addrlen argument are used to ret urn the protocol address of the connected peer processes (client)

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

IPv4 Socket Address Structure:

An IPv4 socket address structure, commonly called an " Internet socket address structure, " is named sockaddr_in and defined by including the **<netinet/in.h>** header.

```
struct in_addr
 {
        in_addr_t    s_addr;        /* network byte ordered */
};
struct sockaddr_in
{
        uint8_t          sin_len;               /* length of structure(16) */
        sa_family_t    sin_family;                   /* AF_INET */
        in_port_t       sin_port;               /* 16-bit TCP or UDP port number*/
                                                /* network byte ordered */
       struct in_addr     sin_addr;                  /* 32-bit IPv4 address */
                                                /*newtork byte ordered */
       char  sin_zero[8];                       /* unused */
};
```

Address Conversion functions

```
#include<netinet/in.h>
Uint16_t      htons( uint16_t    host16bitvalue);
Uint32_t      htonl( uint32_t     host32bitvalue);
Uint16_t      ntohs( uint16_t      net16bitvalue);
Uint32_t      ntohl( uint32_t      net32bitvalue);
```

<u>Pseudo code:</u>

Server side Filer Transfer TCP Pseudo code:

START

Start the program.

Declare the variables and structure for the socket.

Create a socket using socket functions

The socket is binded at the specified port.

Using the object the port and address are declared.

After the binding is executed the file is specified.

Then the file is specified.

Execute the client program.

END

Client side File Transfer TCP Pseudo code:

START

Start the program.

Declare the variables and structure.

Socket is created and connects function is executed.

If the connection is successful then server sends the message.

The file name that is to be transferred is specified in the client side.

The contents of the file is verified from the server side.

END

SERVER PROGRAM

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#define SERV_PORT 5576
main(int argc,char **argv)
{
        int i,j;
        ssize_t n;
  FILE *fp;
  char s[80],f[80];
  struct sockaddr_in servaddr,cliaddr;
  int listenfd,connfd,clilen;
  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);
  clilen=sizeof(cliaddr);
  connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&clilen);
  printf("\n clinet 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));
  }
}
```

SERVER OUTPUT:

```
 clinet connected
 name of the file :  samplehai
This is the argument file name
for the server
enjoy the np lab......
```

CLIENT PROGRAM

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#define SERV_PORT 5576
main(int argc,char **argv)
{
 int i,j;
 ssize_t n;
 char filename[80],recvline[80];
 struct sockaddr_in servaddr;
 int sockfd;
 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("enter the file name");
 scanf("%s",filename);
 write(sockfd,filename,sizeof(filename));
 printf("\n data from server: \n");

 while(read(sockfd,recvline,80)!=0)
 {
         fputs(recvline,stdout);
 }
}
```

CLIENT OUTPUT

```
enter the file namesample
data from server:
hai
this is the argument file name
for the server
enjoy the np lab......
```

WEEK-7

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 onlywhen 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.

This process can significantly increase execution efficiency of a program. (If you are more concerned with performance than portability, we discuss some alternatives *to select() and poll()toward the end of the article.)*
select( ) description The Single UNIX Specification, version 2 (SUSv2) defines select() as follows:

int select(int nfds,fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout); It takes these parameters:
· *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

The return value indicates the number of file descriptors (fds) whose request event has been satisfied. 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*

Upon return from *select()*, *FD_ISSET()* can be called for each fd in a given set to identify whether its condition has been met. With the *timeout* value, you can specify how long *select()* will wait for an event. If*timeout* is *NULL*, *select()* will wait indefinitely for an event. If timeout's *timeval*structures are set to 0, *select()* will return immediately rather than wait for any event to occur. Otherwise, *timeout* defines how long *select()* will wait.

Pseudo code for SERVER:

START

```
        Declare sockfd,connfd as integer variables
        Declare clint as integer array
        Declare len newfd,maxfd,max and I as integer variables
        Declare character arrays named as recv_bufand send_buf
        Declare rset,allset ae fd_set type
        Declare variables named server_addr and client_addr for sockaddr_in structure
        Declare n as ssize_t type
        If socket system call returns -1
                then perror socket
                Exit
        Call memeset system call to set the no of bytes to the value  in the destination
        Set server_addr.sin_family=AF_INET
        Set server_addr.sin_port=htons(50000)
        Set server_addr.sin_addr.s_addr=htonl(INADDR_ANY)
        Call bzero system call to set the specified no of bytes to 0
        If bind system call returns -1
                then
        Perror unable to bind
        Exit
```

```
End if
Call listen system call
Set maxfd = sockfd
Set maxi=-1
Loop
        form 0 to less than fd_setsize
                Set client[i]=-1
                Call FD_ZERO( &all) to initialize the set all bits off
                Call FD_SET(sockfd,&all) to turn on sockfd
                Print tcp server waiting
While true
Set rset=allset
Call select system call tomonitor multiple file descriptors and assign it to nready
If FD_ISSET system call returns true
Then
        Set len=sizeof(client_addr)
        Call accept system call to accept the client request and assign it to the connfd
        Print I got connection from client

        Loop  from 0 to less than FD_SETSIZE
                If client[i] is less than zero
                then
                        Set client[i]=connfd
                        Break
                End if
                If I is equal to FD_SETSIZE
                        Print too many clients
                        Exit
                End if
                Call FD_SET system call to set all bits on
                If connfd is grater than maxfd
                then
                        Set maxfd=connfd
                        If I is grater than maxi
                        then
                                Set maxi=i
                                 If –nready <=0 then
                                        Continue
End loop
Loop from 0 to less than or equal to maxi
```

If newfd =client[i] is grater than zero
then
    Continue
If FD_ISSET returns true
then
    If recv system call returns-1
    then
        Close newfd
        Call FD_CLR system call toclear the bits
        Set client[i]=-1
    End if
Else
    Print text from the client
    Set j=string lenth of received buffer
    Declare a integer variable k
    Loop from 0 to less than j
        Call toupper(recv_buff[k]) function and assign it to the send_buf[k]
    End loop
    Set send_buf to NULL
    Print upper case text send_buf
    Send the upper case text to client
End if

If –nready is less than or equal to zero
then
    Break
End if
End loop
End if
Return 0
END

<u>Pseudo code for CLIENT</u>

START

    Declare sock as integer variable
    Declare character arryas named fname and op
    Declare a file pointer variable named fp
    Declare variables named server_addr for sockaddr_in structure

If socket system call returns -1
Then
Perror socket
Exit
Call memeset system call to set the no of bytes to the value cin the destination
Set server_addr.sin_family=AF_INET
Set server_addr.sin_port=htons(40000)
Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")
Call bzero system call to set the specified no of bytes to 0
If connect system call returns -1
Then
Perror connect
Exit
While true
Print enter file name
Read fname
Send file to socket
Receive file from the socket
Print the contents in the file
Open file in write mode
Write contents to file
Print file sent successfully
Close file
Break
Close socket
Return 0
END

PROGRAM

SERVER 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
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
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?

WEEK-8

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 first argument is a pointer to the first element of an array of structures. Each element of the array is a pollfd structure that specifies the condition to be tested for a given descriptor fd.

```
Structure pollfd
{
    Int fd;
    Short events;
    Short revents;
}
```

The number of elements in the array of structures is specified by the nfds argument.

The conditions to be tested are specified by the events member, and the function returns the status for that, descriptor in the corresponding revents member.

Constants	Inputnto events ?	Result from revents	description
POLLIN	•	•	Normal or priority band normal date
POLLRDNORM	•	•	normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High_ Priority data can be read
POLLOUT	•	•	normal data can be written
POLLWRNORM	•	•	normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	An error has can occurred
POLLHUP		•	An error has can occurred
POLLNVAL		•	Descriptor is not an open file

Fig: input events and returned revents for poll

The timeout argument specifies how long the function is to wait before returning.
A positive value specifies the number of milliseconds to wait.

Timeout value	Description
INFTIM	Wait forever
0	Return immediately, do not block
>0	Wait specified number of milliseconds

Fig: time out values for poll

Pseudo code for SERVER:

```
START
        Declare structure  variables for Server socket data
        take character buffers to store data
        create IPV4 socket by calling socket() system call
        if socket system call returns -1
        then
                perror
                exit
        Initialize server socket
        Bind server to an IP address
        If bind system call returns -1
        Then
                Perror unable to bind
                Exit
        Listen for clients on port
        While true
                Poll for client descriptors
                Accept connections from client
                If recv less than zero
                        Print error no
                else
                        Accept data from client and store in character buffers
                        Print  received data
                        Send data received from client again to client
                Close the connection
    END
```

Pseudo code for CLIENT

```
START
        Declare sock  as integer variable
        Declare character arryas named fname and op
        Declare a file pointer variable named fp
        Declare variables named server_addr  for sockaddr_in structure
        If socket system call returns -1
        Then
                Perror socket
                Exit
        Call memeset system call to set the no of bytes to the value  cin the destination
        Set server_addr.sin_family=AF_INET
        Set server_addr.sin_port=htons(40000)
        Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")
        Call bzero system call to set the specified no of bytes to 0
        If connect system call returns -1
        Then
                Perror connect
                Exit
        While true
                Print enter file name
                Read fname
                Send file to socket
                Receive file from the socket
                Print the contents in the file
                Open file in write mode
                Write contents to file
                Print file sent successfully
                Close file
                Break
                Close socket
                Return 0
END
```

PROGRAM

SERVER 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 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
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

WEEK-9

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

DESCRIPTION:

UDP provides a connectionless service as there need not be any long-term relationship between a UDP client and server.

The User Datagram Protocol

The TCP/IP protocol suite provides two transport protocols, the *User Datagram Protocol* (UDP) described in this chapter, and the *Transmission Control Protocol* (TCP).There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers:
UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. UDP is less complex and easier to understand.
The characteristics of UDP are given below.
End-to-end: UDP can identify a specific process running on a computer.
Connectionless: UDP follows the connectionless paradigm (see below).
Message-oriented: Processes using UDP send and receive individual messages called *segments*.
Best-effort: UDP offers the same best-effort delivery as IP.
Arbitrary interaction: UDP allows processes to send to and receive from as many other processes as it chooses.
Operating system independent: UDP identifies processes independently of the local operating system.

The Connectionless Paradigm

UDP uses a *connectionless* communication setup. A process using UDP does not need to establish a connection before sending data and when two processes stop communicating there are no additional, control messages. Communication consists only of the data segments themselves.

Message-Oriented Interface

UDP provides a *message-oriented* interface. Each message is sent as a single UDP segment, which means that data boundaries are preserved. However, this also means that the maximum size of a UDP segment depends on the maximum size of an IP datagram. Allowing large UDP segments can cause problems. Processes sending large segments can result in IP fragmentation, quite often on the sending computer.

UDP offers the same best-effort delivery as IP, which means that segments can be lost, duplicated, or corrupted in transit. This is why UDP is suitable for applications such as voice or video that can tolerate delivery errors. See below for more on UDP problems.

UDP Datagram Format

UDP provides a way for applications to send encapsulated IP datagram without having to establish a connection. UDP transmits *segments* consisting of an 8-byte header followed by the payload. The format is shown in Figure

UDP header

The *SOURCE PORT* field identifies the *UDP process* which sent the datagram.
The *DESTINATION PORT* field identifies the *UDP process* that will handle the payload.
The *MESSAGE LENGTH* field includes the 8-byte header and the data, measured on octets.
The *CHECKSUM* field is optional and stored as zero if not computed (a computed zero is stored as all ones).
Note that UDP does not provide flow control, error control, or retransmission on receipt of a bad segment. All it provides is demultiplexing multiple processes using the port numbers.
The UDP Checksum
The 16-bit *CHECKSUM* field is optional. The sender can choose to compute a checksum or set the field to zero. The receiver only verifies the checksum if the value is non-zero. Note that UDP uses ones-complement arithmetic, so a computed zero value is stored as all-ones.

UDP Problems

Since UDP provides only a simple delivery service, almost all of the problems with UDP are related to delivery problems.
UDP-based applications are prone to failures in a congested or loss-intensive network because a lost UDP datagram has to be handled by the application.
As an extreme example, consider the Network File System (NFS) which uses UDP for remote file system access, since it benefits from the low-overhead nature of UDP. NFS typically writes data in large chunks (often 8 KB blocks), which are then split into IP fragments depending on the MTU of the underlying topology.

Only when *all* the fragments have been received at the destination is the IP datagram reassembled and passed via UDP to the NFS application. If the underlying network loses 10% - 20% of its datagram’s, then NFS will encounter problems, resulting in retransmission of data and thus providing a sluggish and poor performance.

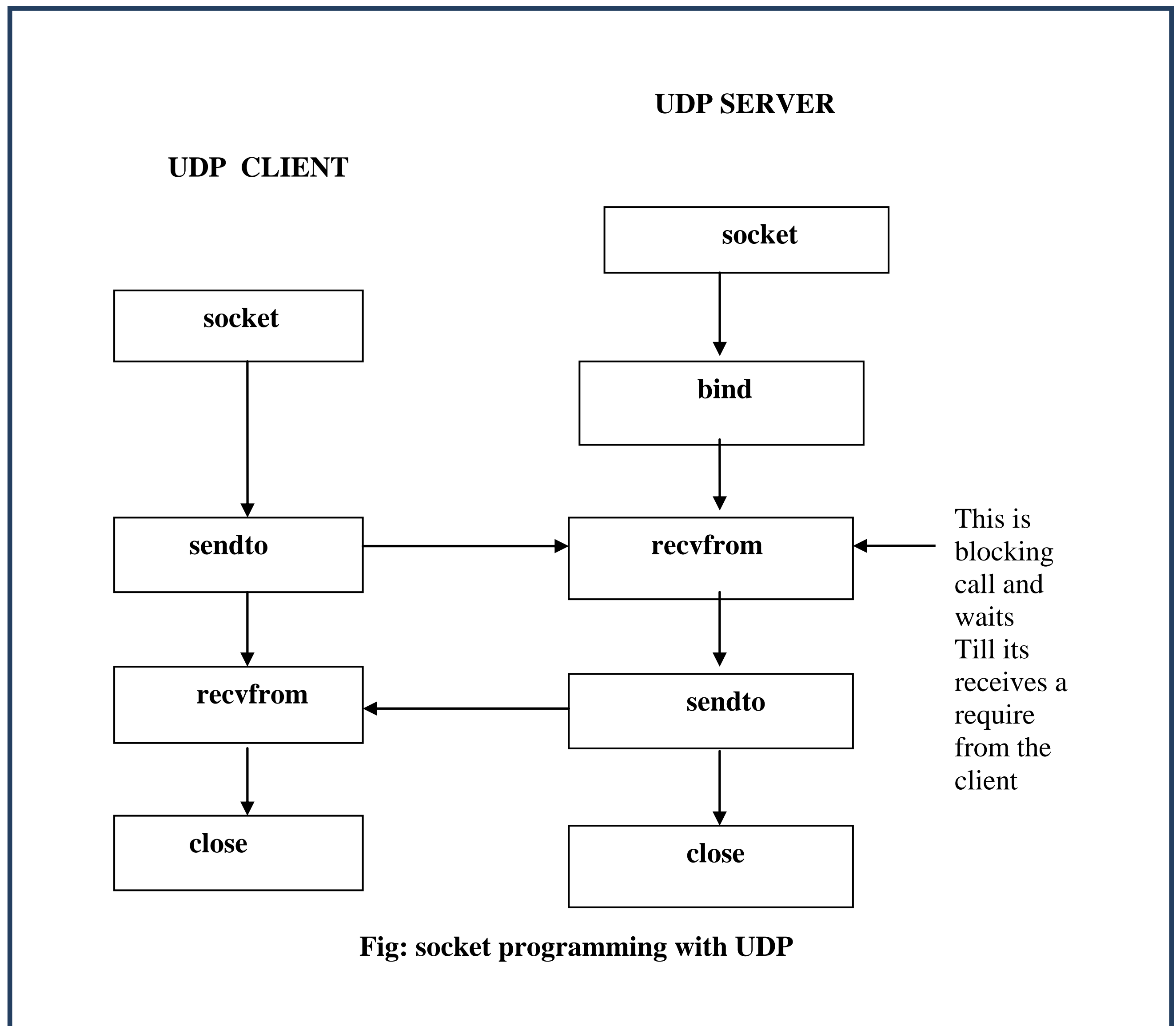


Fig: socket programming with UDP

1)The client does not establish a connection with the server.

2)The client just sends a datagram to the server using the sendto function, which requires the address of the destination as a parameter.
Similarly, the server does not accept a connection from a client.

3)Instead, the server just calls the recvfrom function, which waits until data arrives from some client.

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

We can create UDP socket by specifying the second argument to socket function as SOCK_DGRAM.

sendto and recvfrom functions used to send and receive datagrams

```
include<sys/socket.h>
ssize_t   recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                        struct sockaddr  *form , socklen_t  *addrlen);
```

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

Pseudo code for SERVER

START
    Define LOCAL_SERVER_PORT 1500
    Define MAX_MSG 3000
    Declare structure variables for Server socket data
    take character buffers to store data
    create IPV4 socket by using socket system call
    Initialize server socket
    if socket system call return -1
    then
        perror socket
        exit
    Call memeset system call to set the no of bytes to the value cin the destination
    Set server_addr.sin_family=AF_INET
    Set server_addr.sin_port=htons(50000)
    Set server_addr.sin_addr.s_addr=htonl(INADDR_ANY)
    Call bzero system call to set the specified no of bytes to 0
    If bind system call returns -1
    Then
        Perror unable to bind
        Exit
    End if
    bind local server port
    server infinite loop
    receive message
    reading file contents
    reading data to msg
    closing stream
    print received message
    Send data received from client again to client by reversing it
    Close connection
    end of server infinite loop
END

Pseudo code for CLIENT

```
START

        Declare sock  as integer variable
        Declare character arryas named fname and op
        Declare a file pointer variable named fp
        Declare variables named server_addr  for sockaddr_in structure
        If socket system call returns -1
        then
                Perror socket
                Exit
        Call memeset system call to set the no of bytes to the value  cin the destination
        Set server_addr.sin_family=AF_INET
        Set server_addr.sin_port=htons(40000)
        Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")
        Call bzero system call to set the specified no of bytes to 0
        If connect system call returns -1
                then
                Perror connect
        Exit
        While true
                Print enter file name
                Read fname
                Send file to socket
                Receive file from the socket
                Print the contents in the file
                Open file in write mode
                Write contents to file
                Print file sent successfully
                Close file
                Break
                Close socket
                Return 0
END
```

PROGRAM

SERVER PROGRAM

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

```
#define SERV_PORT 5839
#define MAXLINE 20
main(int argc,char **argv)
{
   int i,j;
   ssize_t n;
   char line[MAXLINE],recvline[MAXLINE];
   struct sockaddr_in servaddr,cliaddr;
   int sockfd,clilen;
   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(SERV_PORT);
   bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
   for( ; ; )
   {
         clilen=sizeof(cliaddr);
         while(1)
         {
                 if((n=recvfrom(sockfd,line,MAXLINE,0,(struct
         sockaddr*)&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,(struct sockaddr*)&cliaddr,clilen);
         }
     }
}
```

CLIENT PROGRAM

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<sys/types.h>
#include<stdlib.h>
#define SERV_PORT 5839
```

```
#define MAXLINE 20
main(int argc,char **argv)
{
   ssize_t n;
   struct sockaddr_in servaddr;
   char sendline[MAXLINE],recvline[MAXLINE];
   int sockfd;
   if(argc!=2)
   {
          printf("usage:<IPADDRESS>");
          exit(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);
   sockfd=socket(AF_INET,SOCK_DGRAM,0);
   printf("enter the data to be send");
   while(fgets(sendline,MAXLINE,stdin)!=NULL)
   {
          sendto(sockfd,sendline,strlen(sendline),0,(struct
sockaddr*)&servaddr,sizeof(servaddr));
          printf("line sent");
          n=recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
          recvline[n]='\0';
          fputs(recvline,stdout);
          printf("\n reverse of the sentense is %s",recvline);
          printf("\n");

   }
   exit(0);
}
```

OUTPUT

Enter the data to be send: cse

Line sent

Reverse of the sentence is:esc

WEEK-10

AIM: Design UDP Client server to transfer a file

DESCRIPTION:

UDP Client and Server

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.

Pseudo code for SERVER

```
START
        Declare structure  variables for Server socket data
        take character buffers to store data
        create IPV4 socket by using socket system call
        Initialize server socket
        if socket system call return -1
        then
                perror socket
                exit
        Call memeset system call to set the no of bytes to the value  cin the destination
        Set server_addr.sin_family=AF_INET
        Set server_addr.sin_port=htons(50000)
        Set server_addr.sin_addr.s_addr=htonl(INADDR_ANY)
        Call bzero system call to set the specified no of bytes to 0
        If bind system call returns -1
        Then
                Perror unable to bind
                Exit
        End if
        bind local server port
        server infinite loop
        receive message
```

reading file contents
reading data to msg
closing stream
print received message
Send data received from client again to client by reversing it
Close connection
end of server infinite loop
END

Pseudo code for CLIENT

START

Declare sock as integer variable
Declare character arryas named fname and op
Declare a file pointer variable named fp
Declare variables named server_addr for sockaddr_in structure
If socket system call returns -1
Then
    Perror socket
    Exit
Call memeset system call to set the no of bytes to the value cin the destination
Set server_addr.sin_family=AF_INET
Set server_addr.sin_port=htons(40000)
Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")
Call bzero system call to set the specified no of bytes to 0
If connect system call returns -1
then
    Perror connect
    Exit
While true
    Print enter file name
    Read fname
    Send file to socket
    Receive file from the socket
    Print the contents in the file
    Open file in write mode
    Write contents to file
    Print file sent successfully
    Close file
    Break
    Close socket
    Return 0
END

PROGRAM

CLIENT PROGRAM

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#define SERV_PORT 6349
main(int argc,char **argv)
{
        char filename[80];
        int sockfd;
        struct sockaddr_in servaddr;
        sockfd=socket(AF_INET,SOCK_DGRAM,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);
        printf("enter the file name");
        scanf("%s",filename);
       sendto(sockfd,filename,strlen(filename),0,(structsockaddr*)&servaddr,sizeof(servad
       dr))
}
```

OUTPUT OF CLIENT

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

SERVER PROGRAM

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netinet/in.h>
#define SERV_PORT 6349
main(int argc,char **argv)
{
  char filename[80],recvline[80];
  FILE *fp;
  struct sockaddr_in servaddr,cliaddr;
  int clilen,sockfd;
  sockfd=socket(AF_INET,SOCK_DGRAM,0);
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family=AF_INET;
  servaddr.sin_port=htons(SERV_PORT);
  bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
  clilen=sizeof(cliaddr);
  recvfrom(sockfd,filename,80,0,(struct sockaddr*)&cliaddr,&clilen);
  printf("\n date in the file is \n ");
  fp=fopen(filename,"r");
  while(fgets(recvline,80,fp)!=NULL)
  {
          printf("\n %s\n ",recvline);
  }
  fclose(fp);
}
```

OUTPUT OF SERVER

```
Server:

date in the file is:

          hai this is np lab

          something intresting
```

WEEK: 11

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.

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

Pseudo code for Server

Define LOCAL_SERVER_PORT 1500

Define MAX_MSG 100

START

    Declare structure variables for Server socket data

    take character buffers to store data

    create IPV4 socket by using socket system call

    Initialize server socket

    if socket system call return -1

    then

        perror socket

        exit

bind local server port

server infinite loop

while true

init buffer

receive message

end of server infinite loop

return 0

END

Pseudo code for TCP Client:

START

Declare sock as integer variable

Declare character arryas named fname and op

Declare a file pointer variable named fp

Declare variables named server_addr for sockaddr_in structure

If socket system call returns -1

then

Perror socket

Exit

Call memeset system call to set the no of bytes to the value cin the destination

Set server_addr.sin_family=AF_INET

Set server_addr.sin_port=htons(40000)

Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")

Call bzero system call to set the specified no of bytes to 0

If connect system call returns -1

then

Perror connect

Exit

While true

Print enter file name

Read fname

Send file to socket

Receive file from the socket

Print the contents in the file

Open file in write mode

Write contents to file

Print file sent successfully

Close file

Break

Close socket

Return 0

END

Pseudo code for UDP Client

START

Declare sock as integer variable

Declare character arryas named fname and op

Declare a file pointer variable named fp

Declare variables named server_addr for sockaddr_in structure

If socket system call returns -1

Then

Perror socket

Exit

Call memeset system call to set the no of bytes to the value cin the destination

Set server_addr.sin_family=AF_INET

Set server_addr.sin_port=htons(40000)

Set server_addr.sin_addr.s_addr=inet_addr("127.0.0.1")

Call bzero system call to set the specified no of bytes to 0

If connect system call returns -1

Then

Perror connect

Exit

While true

Print enter file name

Read fname

Send file to socket

Receive file from the socket

Print the contents in the file

Open file in write mode

Write contents to file

Print file sent successfully

Close file

Break

Close socket

Return 0

END

PROGRAM

CLIENT 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
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 of CLIENT

```
cc selcli.c -o cli
./cli localhost
Enter data to be send:gec-cse
line send to server :gec-cse
line received from the server : GEC-CSE
```

SERVER 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
```

```
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;
           }
        }
    }
}
```

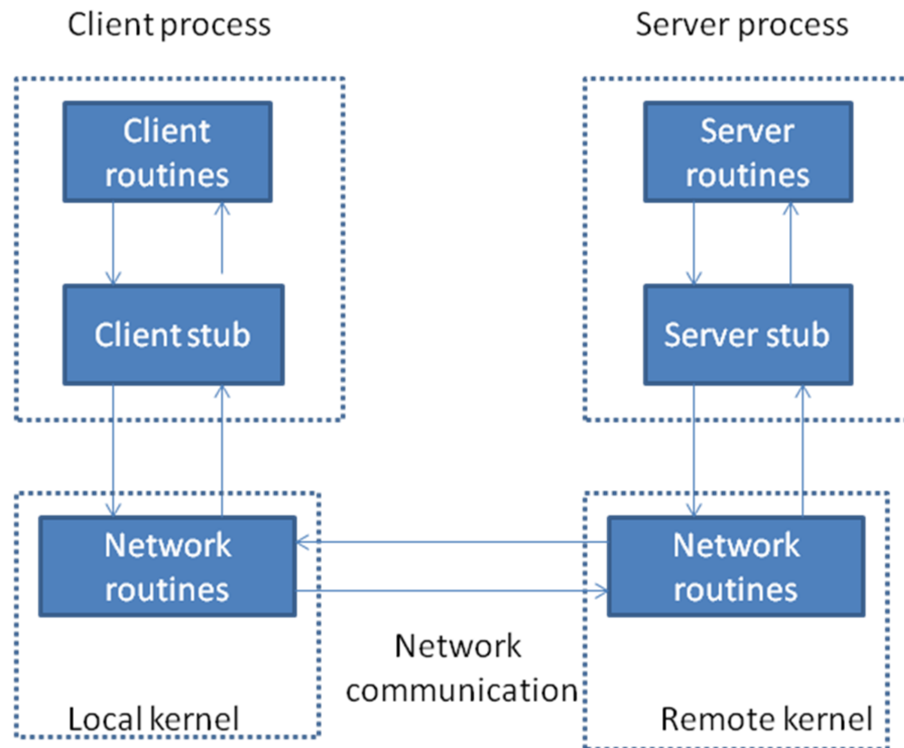
OUTPUT OF SERVER:

```
cc selser.c -o ser
./ser
line received from client:gec-cse
```

WEEK-12

Design a RPC application to add and subtract a given pair of integers

DESCRIPTION:



Remote Procedure Call model

The steps in the Figure Remote Procedure Call (RPC) Model are.

1) 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.

2) These network messages are sent to the remote system by the client stub. This requires a system call into the kernel.

3) The network messages are transferred to the remote system. Either a connection-oriented or a connectionless protocol is used.

4) 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.
5) 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.
6) When the server procedure is finished, it returns to the server stub, returning whatever its return values are.
7) 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.
8) To message get transferred back across the net work to client stub.
9) The client stub reads the network message from the local kernel.
10) 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.

Pseudo code

START

First create RPC specification file with .x extension which defines the server procedure along with their arguments and results. the following program shows the contents of Filename simp.x

Specification file to define server procedure and arguments.

The definition of the data type that will be passed to both of the remote procedures add() and sub().

```
#define VERSION_NUMBER 1
struct operands
{
    int x
    int y
}

Program, version and procedure definitions
Program SIMP_PROG
{
Version SIMP_VERSION
{
int ADD(operands)=1; // Procedure number 1
int SUB(operands)=2; // Procedure number 2
```

```
}=VERSION_NUMBER
}=0x28976543 // Program numbe
```

Program name simp_server.c, definition of the remote add and subtract procedure used by simple RPC example, rpcgen will create a template for you that contains much of the code, needed in this file is you give it the “-Ss” command line arg.

SERVER

```
#include<stdio.h>
#include<rpc/rpc.h>  //always needed
#include “simp.h”  //generated by rpcgen
Here is the actual remote procedure
The return value of this procedure must be a pointer to int.
We declare the variable result as static so we can return a pointer to it
int *add_l_svc(operands *argp, struct svc_req *rqstp)
{
    static int result
    printf(“Got request: adding %d, %d\n”,grgp->x,argp->y)
    result=argp->x + argp->y
    return (&result)
}
int *sub_l_svc(operands *argp, struct svc_req *rqstp)
{
    static int result
    printf(“Got request: subtracting %d, %d\n”,grgp->x,argp->y)
    result=argp->x + argp->y
    return (&result)
}
```

CLIENT

Program name simp_client.c RPC client for simple addition and subtraction example.

```
#include<stdio.h>
#include<rpc/rpc.h> // always needed
#include “simp.h” // created for us by rpcgen – has everything we need
Wrapper function takes care of calling the RPC procedure
int add(CLIENT *clnt, int x, int y)
{
    operands ops
    int *result
    Gather everything into a single data structure to send to the server
    ops.x=x
    ops.y=y
    Call the client stub created by rpcgen
```

```
    result=add_l(&ops, clnt)
    if(result==NULL)
    {
        fprintf(stderr,"Trouble calling remote procedure\n");
        exit(0)
    }
    return(*result)
}
```

Wrapper function takes care of calling the RPC procedure

```
int sub(CLIENT *clnt, int x, int y)
{
    operands ops
    int *result
    Gather everything into a single data structure to send to the server
    ops.x=x
    ops.y=y
    Call the client stub created by rpcgen
    result=sub_l(&ops, clnt)
    if(result==NULL)
    {
        fprintf(stderr,"Trouble calling remote procedure\n");
        exit(0)
    }
    return(*result)
}
int main(int argc, char *argv{})
{
    CLIENT *clnt
    int x,y
    if(argc!=4)
    {
        fprintf(stderr,"Usage: %s hostname num1 num \n",argv[0])
        exit(0)
    }
```

Create a CLIENT data structure that reference the RPC procedure SIMP_PROG, version SIMP_VERSION running on the host specified by the 1st command line arg.

```
clnt=clnt_create(argv[1], SIMP_PROG, SIMP_VERSION, "udp")
```

Make sure the create worked

```
if(clnt==(CLIENT*)NULL)
{
    clnt_pcreateerror(argv[1])
    exit(1)
}
```

```
get the  2 numbers that should be added
x = atoi(argv[2])
y=atoi(argv[3])
printf("add = %d + %d = %d \n",x,y,add(clnt,x,y))
printf("sub = %d – %d = %d \n",x,y,sub(clnt,x,y))
return(0)
}
END
```

SERVER 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));
```

```
52
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");
53
exit (1);
}
```

CLIENT PROGRAM

```
#include "rpctime.h"
void
rpctime_1(char *host)
{
CLIENT *clnt;
47
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 */
48
#include "rpctime.h"
/* Default timeout can be changed using clnt_control() */
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);
}
```

49

Execution procedure and Result:

```
Step 1:     $rpcgen –C –a simp.x
//This creates simp.h, simp_clnt.c, simp_svc.c simp_xdr.c files in the folder //
Step 2: $cc –o client simp_client.c simp_clnt.c simp_xdr.c –lrpcsvc –lnsl
Step 3: $ cc –o server simp_server.c simp_svc.c simp_xdr.c –lrpcsvc –lnsl
Step 4: $ ./server &
$./client 10.0.0.1   10  5
Add = 10 + 5 = 15
Sub = 10 – 5 = 5
```

ADDITIONAL PROGRAMS

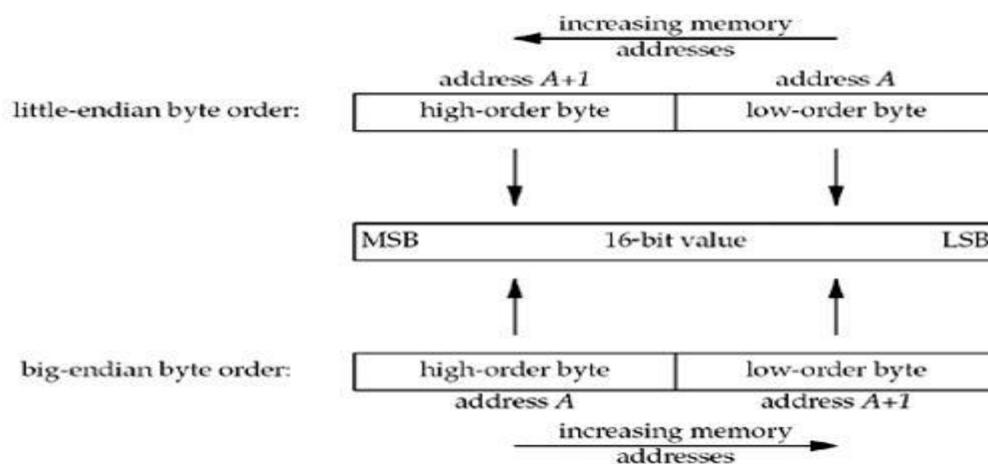
Aim: Program to determine the host ByteOrder

DESCRIPTION:

BYTE ORDERING

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as little-endian byte order, or with the high-order byte at the starting address, known as big-endian byte order. We show these two formats in

Figure 3.9. Little-endian byte order and big-endian byte order for a 16-bit integer.



In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.

The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Unfortunately, there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the host byte order. The program prints the host byte order.

PROGRAM

```
#include     "unp.h"
int
main(int argc, char **argv)
{

union
{
        short   s;
        char    c[sizeof(short)];
} un;

un.s = 0x0102;
printf("%s: ", CPU_VENDOR_OS);

if (sizeof(short) == 2)
{
        if (un.c[0] == 1 && un.c[1] == 2)
                printf("Host follows big-endian Byte order\n");
        else if (un.c[0] == 2 && un.c[1] == 1)
                printf("Host follows little-endian Byte order\n");
        else
                printf("unknown\n");
}
else
        printf("sizeof(short) = %d\n", sizeof(short));
exit(0);
}
```

We store the two-byte value 0x0102 in the short integer and then look at the two consecutive bytes, c[0] (the address A in Figure) and c[1] (the address A+1 in Figure), to determine the byte order.

The string CPU_VENDOR_OS is determined by the GNU autoconf program when the software in this book is configured, and it identifies the CPU type, vendor, and OS release. We show some examples here in the output from this program when run on the various systems.

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian
macosx % byteorder
powerpc-apple-darwin6.6: big-endian
```

OUTPUT:

Host follows little endian Byte order

Aim: Program to set and get socket options

DESCRIPTION:

getsockopt and setsockopt Functions

These two functions apply only to sockets.

#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, const void *optval socklen_t optlen);
Both return: 0 if OK,–1 on error

sockfd must refer to an open socket descriptor. level specifies the code in the system that interprets the option: the general socket code or some protocol-specific code (e.g., IPv4, IPv6, TCP, or SCTP).

optval is a pointer to a variable from which the new value of the option is fetched by setsockopt, or into which the current value of the option is stored by getsockopt. The size of this variable is specified by the final argument, as a value for setsockopt and as a value-result for getsockopt.

Pseudo code

START

Create socket using socket function

Get the TCP maximum segment size using getsockopt function

Print the TCP maximum segment size

Set the socket sendbuffer size using setsockopt function

Get the socket sendbuffer size using getsockopt function

Print the socket sendbuffer size

END

PROGRAM

```
 #include<stdio.h>
 #include<sys/types.h>
 #include<sys/socket.h>
 #include<string.h>
 #include<netinet/in.h>
 #include<netinet/tcp.h>
 main()
{
        int sockfd,maxseg,sendbuff,optlen;
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        optlen=sizeof(maxseg);
        if(getsockopt(sockfd,IPPROTO_TCP,TCP_MAXSEG,(char *)&maxseg,&optlen)<0)
                printf("Max seg error");
        else
                printf("TCP max seg=%d\n",maxseg);
        sendbuff=2500;

if(setsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,(char*)&sendbuff,sizeof(sendbuff))<0)
                printf("set error");
        optlen=sizeof(sendbuff);
        getsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,(char *)&sendbuff,&optlen);
        printf("send buff size=%d\n",sendbuff);
}
```

OUTPUT

TCP max seg=512

Send buff size=5000