

R20 Autonomous Lab Cycle LAB CYLCE -I

A. Practice the following Network related Utilities Commands.

write, wall, mail, talk, ftp, telnet, rlogin, ping, netstat, ipcs, ifconfig etc.

B. Practice the following low-level C Programs that use System calls

- i) to copy contents of a source file to target file
- ii) to concatenate given files in to a new file
- iii) to move a file from one location to another location in file system
- iv) to print the statistics of a given file
- v) to enlist the number of errors identified by your system
- vi) to enlist the number of signals identified by your system
- vii) to create a new file that contains the source file contents in reverse order

LAB CYLCE -II

A. Practice the following local IPC programs.

- i) implement a half duplex pipe
- ii) implement a full duplex pipe
- iii) implement a uni directional FIFO
- iv) implement a bi directional FIFO
- v) implement a message queue
- vi) implement shared memory
- vii) demonstrate semaphore

B. Practice the following Network IPC programs.

- i) develop an Iterative Client Server set up for DayTime service
- ii) develop an Iterative Client Server set up for echo service
- iii) develop a concurrent Client Server set up for DayTime service
- iv) develop a concurrent Client Server set up for echo service
- v) develop a concurrent Client Server set up for reversing the given string
- vi) develop a concurrent Client Server set up for File service

Additional Experiments:

1. Develop C program to determine host byte order
2. Develop C program to handle socket options

LAB CYLCE -I

Network related Utilities Commands.

The ping Utility

The ping command sends an echo request to a host available on the network. Using this command, you can check if your remote host is responding well or not.

The ping command is useful for the following –

- Tracking and isolating hardware and software problems.
- Determining the status of the network and various foreign hosts.
- Testing, measuring, and managing networks.

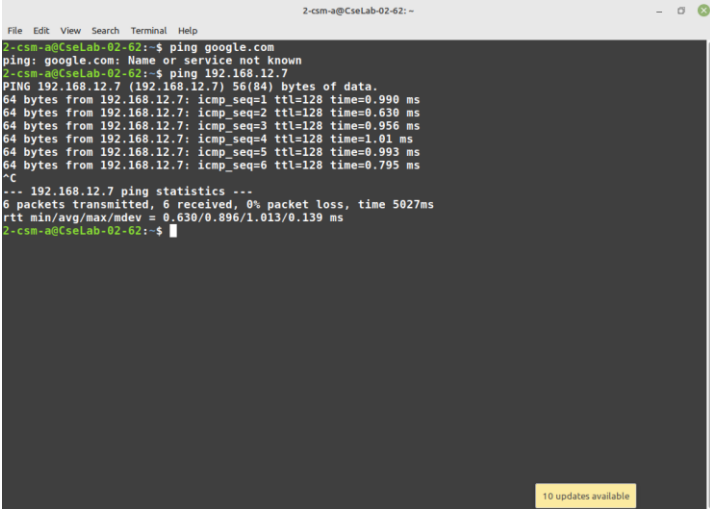
Syntax

Following is the simple syntax to use the ftp command –

\$ping hostname or ip-address

The above command starts printing a response after every second. To come out of the command, you can terminate it by pressing CNTRL + C keys

output:

A screenshot of a terminal window titled '2-csm-a@CseLab-02-62: ~'. The terminal shows the execution of the 'ping google.com' command. The output indicates that the service is not known. Then, the user runs 'ping 192.168.12.7', which shows six successful ping requests with varying response times (0.990 ms to 1.01 ms). The user then presses Ctrl+C, which displays the ping statistics: 6 packets transmitted, 6 received, 0% packet loss, and a total time of 5027ms. The statistics also show the minimum, average, maximum, and standard deviation of the response times. The terminal window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. A yellow notification bar at the bottom right says '10 updates available'.

The ftp Utility

Here, ftp stands for File Transfer Protocol. This utility helps you upload and download your file from one computer to another computer.

The ftp utility has its own set of Unix-like commands. These commands help you perform tasks such as –

- Connect and login to a remote host.
- Navigate directories.
- List directory contents.
- Put and get files.
- Transfer files as ascii, ebcdic or binary.

Syntax

Following is the simple syntax to use the ftp command –

`$ftp hostname or ip-address`

Sr.No.	Command & Description
1	<code>put filename</code> Uploads filename from the local machine to the remote machine.
2	<code>get filename</code> Downloads filename from the remote machine to the local machine.
3	<code>mput file list</code> Uploads more than one file from the local machine to the remote machine.
4	<code>mget file list</code> Downloads more than one file from the remote machine to the local machine.
5	<code>prompt off</code> Turns the prompt off. By default, you will receive a prompt to upload or download files using <code>mput</code> or <code>mget</code> commands.

6	prompt on Turns the prompt on.
7	dir Lists all the files available in the current directory of the remote machine.
8	cd dirname Changes directory to dirname on the remote machine.
9	lcd dirname Changes directory to dirname on the local machine.
10	quit Helps logout from the current login.

Example:

\$ftp amrood.com

Connected to amrood.com.

220 amrood.com FTP server (Ver 4.9 Thu Sep 2 20:35:07 CDT 2009)

Name (amrood.com:amrood): amrood

331 Password required for amrood.

Password:

230 User amrood logged in.

ftp> dir

200 PORT command successful.

150 Opening data connection for /bin/ls.

total 1464

drwxr-sr-x 3 amrood group 1024 Mar 11 20:04 Mail

```
drwxr-sr-x  2 amrood  group      1536 Mar  3 18:07 Misc
drwxr-sr-x  5 amrood  group      512 Dec  7 10:59 OldStuff
drwxr-sr-x  2 amrood  group     1024 Mar 11 15:24 bin
drwxr-sr-x  5 amrood  group     3072 Mar 13 16:10 mpl
-rw-r--r--  1 amrood  group    209671 Mar 15 10:57 myfile.out
drwxr-sr-x  3 amrood  group      512 Jan  5 13:32 public
drwxr-sr-x  3 amrood  group      512 Feb 10 10:17 pvm3
```

226 Transfer complete.

```
ftp> cd mpl
```

250 CWD command successful.

```
ftp> dir
```

200 PORT command successful.

150 Opening data connection for /bin/ls.

total 7320

```
-rw-r--r--  1 amrood  group      1630 Aug  8 1994  dboard.f
-rw-r-----  1 amrood  group      4340 Jul 17 1994  vttest.c
-rwxr-xr-x   1 amrood  group     525574 Feb 15 11:52 wave_shift
-rw-r--r--  1 amrood  group      1648 Aug  5 1994  wide.list
-rwxr-xr-x   1 amrood  group      4019 Feb 14 16:26 fix.c
```

226 Transfer complete.

```
ftp> get wave_shift
```

200 PORT command successful.

150 Opening data connection for wave_shift (525574 bytes).

226 Transfer complete.

528454 bytes received in 1.296 seconds (398.1 Kbytes/s)

```
ftp> quit
```

221 Goodbye.

\$

The telnet Utility

There are times when we are required to connect to a remote Unix machine and work on that machine remotely. Telnet is a utility that allows a computer user at one site to make a connection, login and then conduct work on a computer at another site.

Once you login using Telnet, you can perform all the activities on your remotely connected machine. The following is an example of Telnet session –

C:>telnet amrood.com

Trying...

Connected to amrood.com.

Escape character is '^]'.

login: amrood

amrood's Password:

```

*                                                    *
*                                                    *
*  WELCOME TO AMROOD.COM                               *
*                                                    *
*                                                    *
```

Last unsuccessful login: Fri Mar 3 12:01:09 IST 2009

Last login: Wed Mar 8 18:33:27 IST 2009 on pts/10

{ do your work }

\$ logout

Connection closed.

C: >

The finger Utility

The `finger` command displays information about users on a given host. The host can be either local or remote.

Finger may be disabled on other systems for security reasons.

Following is the simple syntax to use the finger command –

Check all the logged-in users on the local machine –

\$ finger

Login	Name	Tty	Idle	Login Time	Office
amrood		pts/0		Jun 25 08:03	(62.61.164.115)

Get information about a specific user available on the local machine –

\$ finger amrood

Login: amrood Name: (null)

Directory: /home/amrood Shell: /bin/bash

On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115

No mail.

No Plan.

Sending Email

You use the Unix mail command to send and receive mail. Here is the syntax to send an email –

syntax-

\$mail [-s subject] [-c cc-addr] [-b bcc-addr] to-addr

Here are important options related to mail command `-s`

Example

Sr.No.	Option & Description
1	-s Specifies subject on the command line.
2	-c Sends carbon copies to the list of users. List should be a commaseparated list of names.
3	-b Sends blind carbon copies to list. List should be a commaseparated list of names.

\$mail -s "Test Message" admin@yahoo.com

LAB CYLCE -I

part-B

B. Practice the following low-level C Programs that use System calls

Program-i)

AIM: To copy contents of a source file to target file

Description:

System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system.

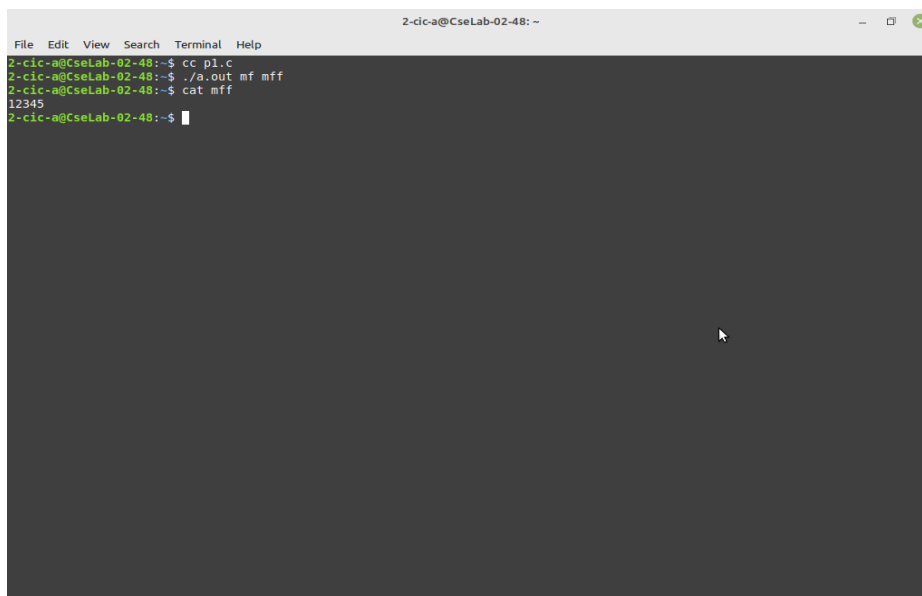
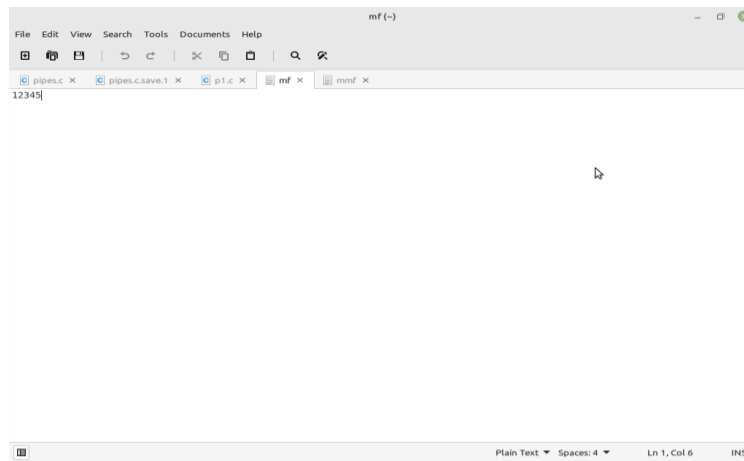
System calls in Unix are used for file system control, process control, interprocess communication etc. Access to the Unix kernel is only available through these system calls. Generally, system calls are similar to function calls, the only difference is that they remove the control from the user process.

program:

```
#include<stdio.h>
# include<fcntl.h> main(int argc,char *argv[])
{
int fd1,fd2,count; char buf[512]; if(argc!=3)
{
printf("give sufficient filenames"); exit(1);
}
else
{
fd1=open(argv[1],O_RDONLY); if(fd1==-1)
{
printf("source file does not exist"); exit(1);
}
fd2=open(argv[2],O_WRONLY); if(fd2==-1) fd2=creat(argv[2],0666);
while((count=read(fd1,buf,512))>0)
{
write(fd2,buf,count);
}
close(fd);
close(fd1);
}
```

```
int c=unlink(argv[1]); if(c==0)
printf("unlinked successfully"); else
printf("link error");
}
```

output:



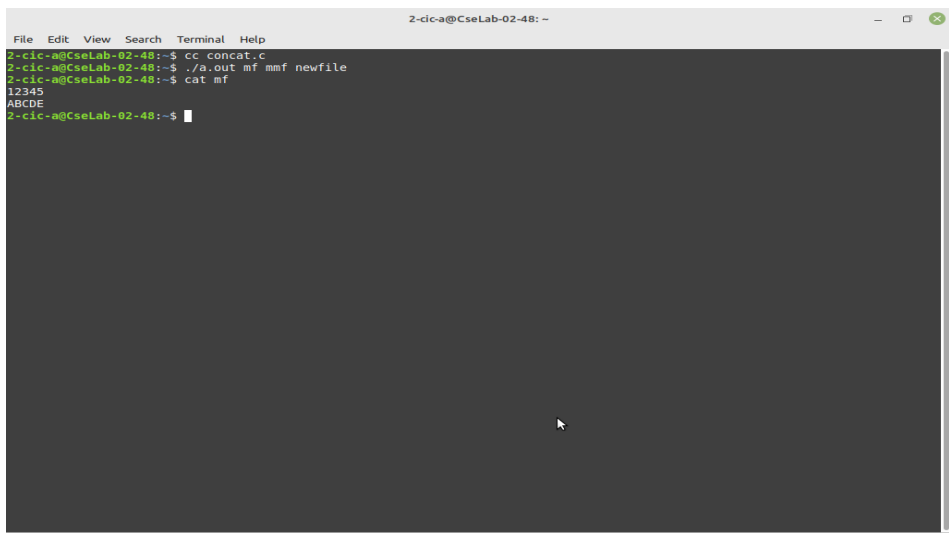
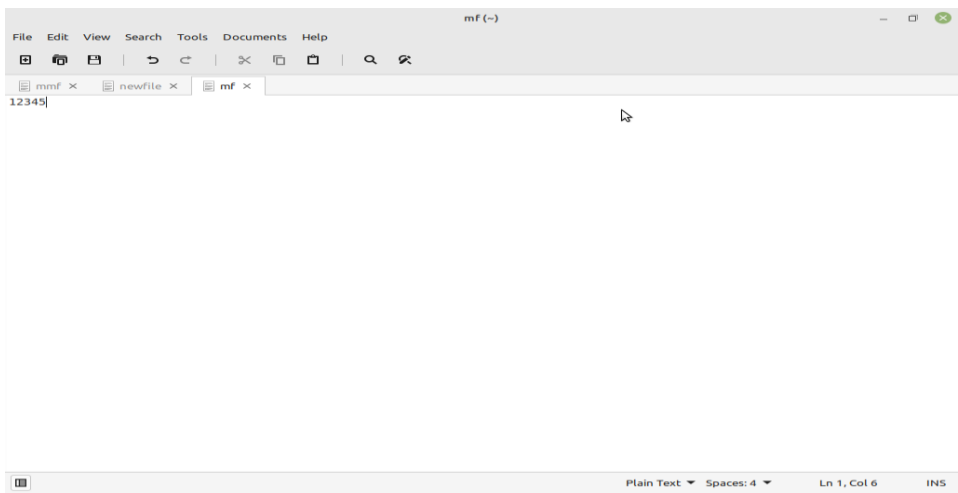
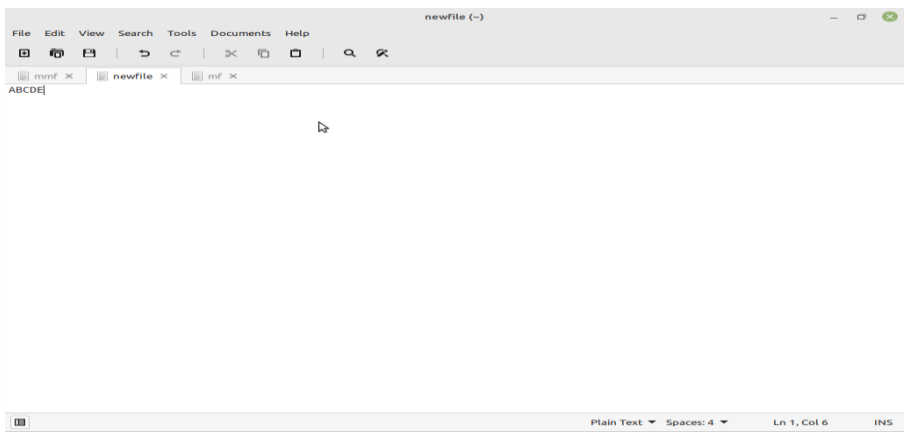
program-ii)

AIM: To concatenate given files in to a new file

```
#include<stdio.h>
#include<fcntl.h>
Int main(int argc,char *argv[])
{
int fdi,fdt,count; char buf[512];
if(argc<=3)
{
printf("give sufficient filenames"); exit(1);
}
fdt=open(argv[1],O_CREAT|O_WRITE,0666);
for ( i=2;i<=argc-1;i++)
{
fdi=open(argv[i],O_RDONLY);
if(fdi==-1)
{
printf("source file does not exist"); exit(1);
}
While((n=ead(fdi,buf,512))>0)
{
write(fdt,buf,n);
}
close(fdi);
}
close(fdt);
}
return 0;

}
```

output:



program-iii)

To move a file from one location to another location in file system

pogram:

```
#include<stdio.h>
#include<fcntl.h>
int main(int argc,char *argv[])
{
    int fd; char buf[512];
    if(argc!=3)
    {
        printf("give a file name, location to move"); exit(1);
    }
    fd=open(argv[1],O_RDONLY);
    fdt=creat(argv[2]/argv[1],O_CREAT|O_WRITE, 0666);
    while((n=read(fd,buf,512))>0)
    {
        write(fdt,buf,n);
    }
    close(fd);
    remove(argv[1]);
    close(fdt);
}
return 0;
```

Excepected output:

np/File1
Hello

Np2/File1
Hello

program-iv)

AIM: To print the statistics of a given file

```
#include <sys/stat.h>          /* declare the 'stat' structure */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define  FILENAME "text"      /* PUT YOUR FILE NAME HERE */

char * format_time(time_t cal_time);

void file_stat(char * filename);

/*****
*****/

int main()
{
    file_stat(FILENAME);
}

/*****
*****/

void file_stat(char * filename)
{
    struct stat stat_p;        /* 'stat_p' is a pointer to a structure* of type 'stat'. */
                               /* Get stats for file and place them in* the
                               structure.*/

    if ( -1 ==  stat (filename, &stat_p))
    {
        printf(" Error occoured attempting to stat %s\n", filename);
        exit(0);
    } /* Print a few structure members. */

    printf("Stats for %s \n", filename);
    printf("Modify time is %s", format_time(stat_p.st_mtime));
```

```

    /* Access time does not get updated if the filesystem is NFS mounted!
    */

    printf("Access time is %s", format_time(stat_p.st_atime));

    printf("File size is    %d bytes\n", stat_p.st_size);
}

/*****
*****/

char * format_time(time_t cal_time)
{
    struct tm *time_struct;

    static char string[30]; /* Put the calendar time into a structure* if type 'tm'.
                           */

    time_struct=localtime(&cal_time);

                           /* Build a formatted date from the * structure.*

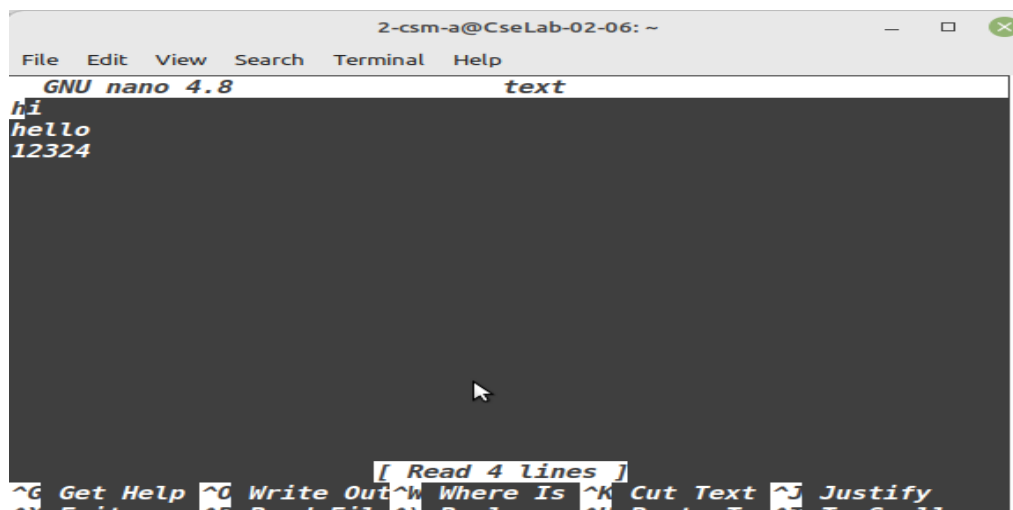
    strftime(string, sizeof string, "%h %e %H:%M\n", time_struct);

                           /* Return the date/time    */

    return(string);
}

```

output:



```

2-csm-a@CseLab-02-06: ~
File Edit View Search Terminal Help
GNU nano 4.8 text
hi
hello
12324
[ Read 4 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^F Read File ^R Replace ^I Paste Text ^_ To Spell

```



```
2-csm-a@CseLab-02-06: ~  
File Edit View Search Terminal Help  
one_location_another.c 模板  
Pictures  
2-csm-a@CseLab-02-06:~$ pico text  
2-csm-a@CseLab-02-06:~$ pico stat.c  
2-csm-a@CseLab-02-06:~$ cc stat.c  
stat.c: In function 'file_stat':  
stat.c:57:27: warning: format '%d' expects argument of type 'int', but argument 2 has type '__off_t' {aka 'long int'} [-Wformat=]  
57 |     printf("File size is %d bytes\n", stat_p.st_size);  
    |                               ^~  
    |                               |  
    |                               int  
    |                               |  
    |                               __off_t {aka  
a long int}  
    |                               |  
    |                               %ld  
2-csm-a@CseLab-02-06:~$ ./a.out  
Stats for text  
Modify time is May 27 20:01  
Access time is May 27 20:01  
File size is 17 bytes  
2-csm-a@CseLab-02-06:~$
```

program-v)

AIM: To enlist the number of errors identified by your system

Program:

```
#include<stdio.h>

#include<error.h>

#include<errno.h>

#include<string.h>

int main{

int i;

printf("list o all errors are \n");

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

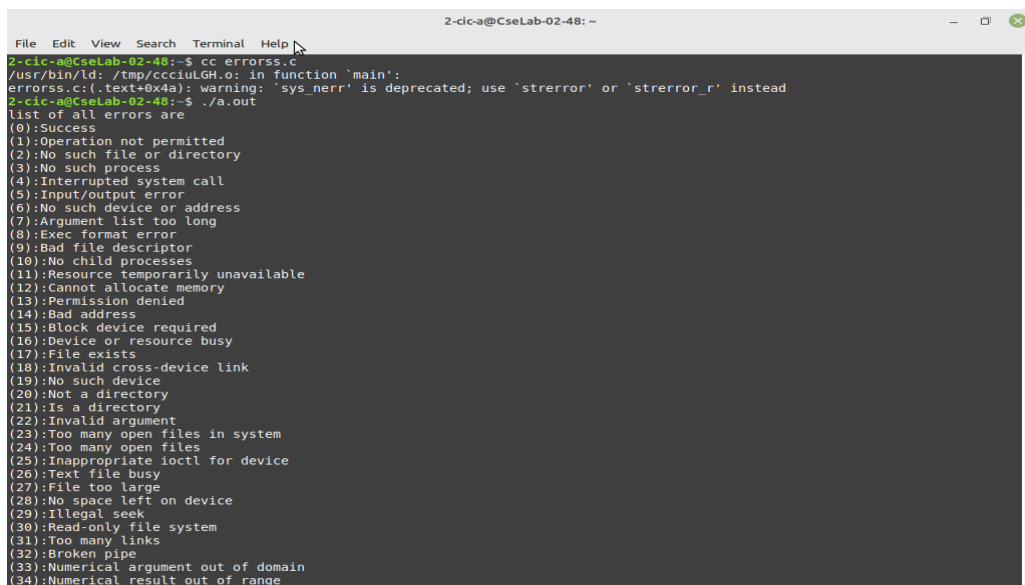
{

printf("(%d):%s\n",i,strerror(i))

}

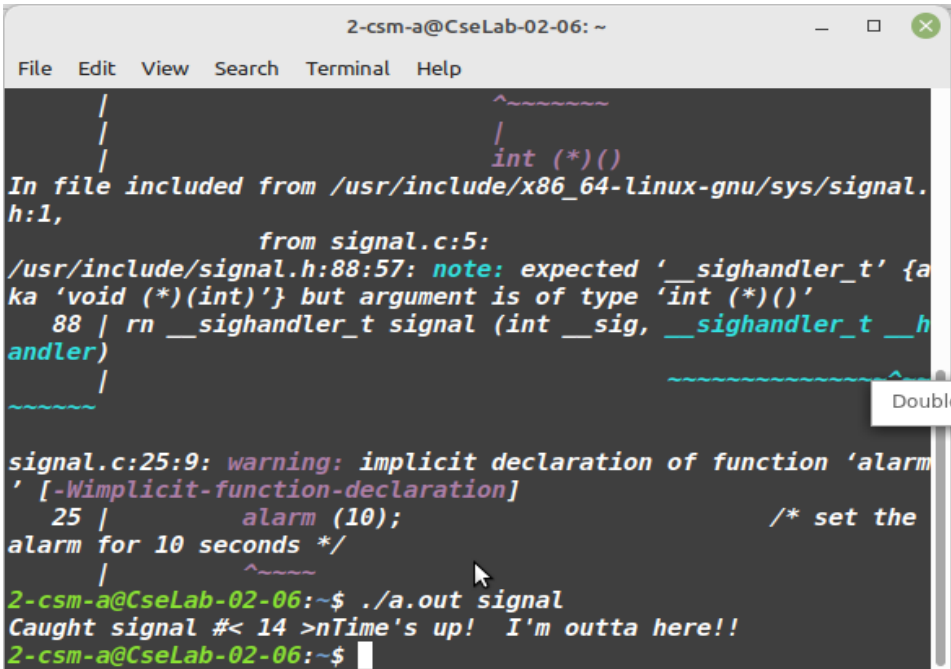
}
```

output:



```
2-cic-a@CseLab-02-48: ~
File Edit View Search Terminal Help
2-cic-a@CseLab-02-48:~$ cc errorss.c
/usr/bin/ld: /tmp/ccciULGH.o: in function `main':
errorss.c:(.text+0x4a): warning: `sys_nerr' is deprecated; use `strerror' or `strerror_r' instead
2-cic-a@CseLab-02-48:~$ ./a.out
list of all errors are
(0):Success
(1):Operation not permitted
(2):No such file or directory
(3):No such process
(4):Interrupted system call
(5):Input/output error
(6):No such device or address
(7):Argument list too long
(8):Exec format error
(9):Bad file descriptor
(10):No child processes
(11):Resource temporarily unavailable
(12):Cannot allocate memory
(13):Permission denied
(14):Bad address
(15):Block device required
(16):Device or resource busy
(17):File exists
(18):Invalid cross-device link
(19):No such device
(20):Not a directory
(21):Is a directory
(22):Invalid argument
(23):Too many open files in system
(24):Too many open files
(25):Inappropriate ioctl for device
(26):Text file busy
(27):File too large
(28):No space left on device
(29):Illegal seek
(30):Read-only file system
(31):Too many links
(32):Broken pipe
(33):Numerical argument out of domain
(34):Numerical result out of range
```


output:



```
2-csm-a@CseLab-02-06: ~  
File Edit View Search Terminal Help  
/~~~~~  
/      |  
/      | int (*)()  
In file included from /usr/include/x86_64-linux-gnu/sys/signal.  
h:1,  
      from signal.c:5:  
/usr/include/signal.h:88:57: note: expected '__sig_handler_t' {a  
ka 'void (*)(int)'} but argument is of type 'int (*)()'   
88 | rn __sig_handler_t signal (int __sig, __sig_handler_t __h  
andler)   
~~~~~  
~~~~~  
signal.c:25:9: warning: implicit declaration of function 'alarm  
' [-Wimplicit-function-declaration]  
25 |      alarm (10);          /* set the  
alarm for 10 seconds */  
    |      ~~~~~  
2-csm-a@CseLab-02-06:~$ ./a.out signal  
Caught signal #< 14 >nTime's up!  I'm outta here!!  
2-csm-a@CseLab-02-06:~$
```

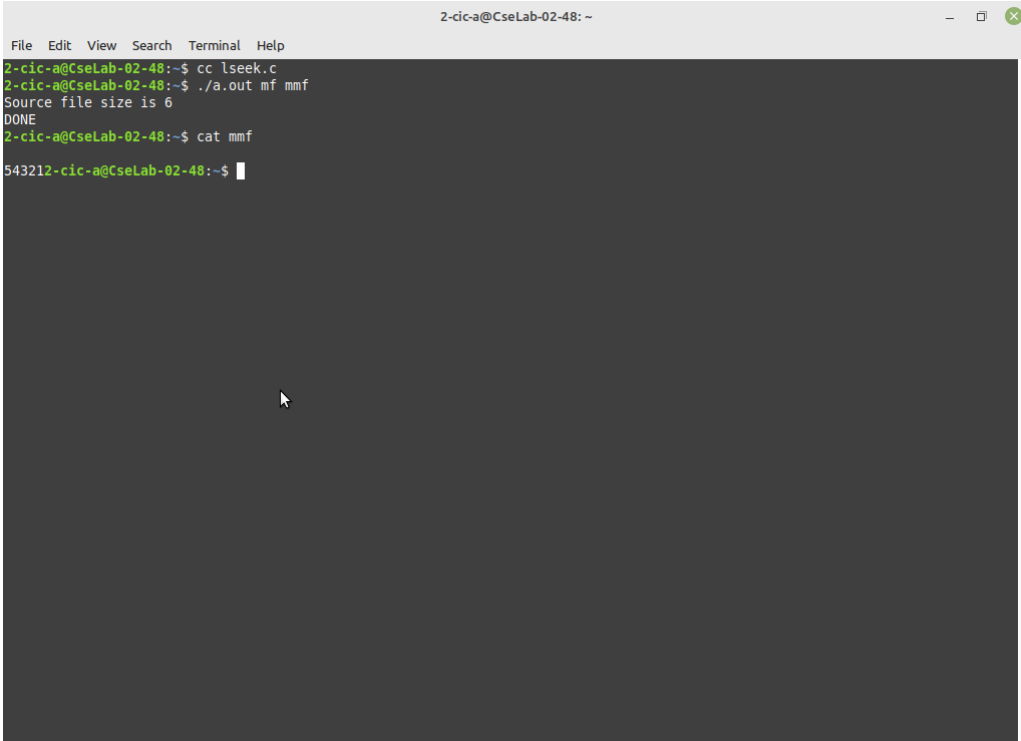
Double

program-vii)

AIM: To create a new file that contains the source file contents in reverse order

```
#include<stdio.h>
# include<fcntl.h>
int main(int argc,char *argv[])
{
int fd1,fd2,count; char buf[512]; if(argc!=3)
{
printf("give sufficient filenames"); exit(1);
}
else
{
fd1=open(argv[1],O_RDONLY);
if(fd1==-1)
{
printf("source file does not exist");
exit(1);
}
fd2=open(argv[2],O_WRONLY);
if(fd2==-1) fd2=creat(argv[2],0666);
fd=lseek(fd1,-1,2);
while((count=read(fd1,&ch, 1))>0)
{
write(fd2,&ch, 1);
fd=lseek(fd1,-1,2);
}
close(fd);
close(fd1);
}
return 0;
}
```

output:



```
2-cic-a@CseLab-02-48: ~  
File Edit View Search Terminal Help  
2-cic-a@CseLab-02-48:~$ cc lseek.c  
2-cic-a@CseLab-02-48:~$ ./a.out mf mmf  
Source file size is 6  
DONE  
2-cic-a@CseLab-02-48:~$ cat mmf  
543212-cic-a@CseLab-02-48:~$
```

LAB CYLCE -II

A. Practice the following local IPC programs.

program-i) implement a half duplex pipe

Algorithm

Step 1 – Create a pipe.

Step 2 – Create a child process.

Step 3 – Parent process writes to the pipe.

Step 4 – Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 – Repeat step 3 and step 4 once again.

PIPE creating program:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <errno.h>
```

```
#include <unistd.h>
```

```
int main()
```

```
{
```

```
    int pfd[2];
```

```
    char buf[30];
```

```
    if (pipe(pfd) == -1) {
```

```
        perror("pipe error");
```

```
        exit(1);
```

```
    }
```

```
    printf("writing to file descriptor %d\n", pfd[1]);
```

```
    write(pfd[1], "Hello", 6);
```

```
    printf("reading from file descriptor %d\n", pfd[0]);
```

```
    read(pfds[0], buf, 6);

    printf("read \"%s\"\n", buf);

    return 0;

}
```

<u>Compile :</u>
gcc pipe.c -o pipe
./pipe

program:Uni-directional Pipe

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <errno.h>
```

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

```
#include <unistd.h>
```

```
int main(void)
```

```
{
```

```
    int pfds[2];
```

```
    char buf[30];
```

```
    if (pipe(pfds) == -1) {
```

```
        perror("pipe error");
```

```
        exit(1);
```

```
    }
```

```
    if (!fork()) {
```

```
        printf(" CHILD: writing to the pipe\n");
```

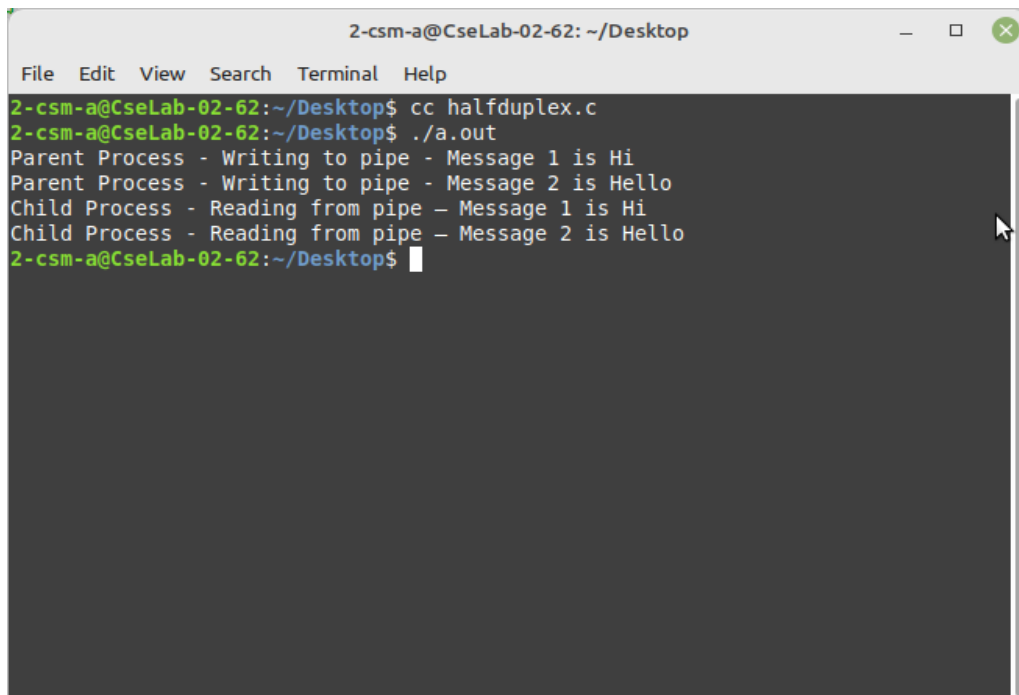
```
        write(pfds[1], "Hello", 6);
```

```
        printf(" CHILD: exiting\n");
```



```
        exit(0);  
    } else {  
        printf("PARENT: reading from pipe\n");  
        read(pfds[0], buf, 6);  
        printf("PARENT: read \"%s\"\n", buf);  
        wait(NULL);  
    }  
    return 0;  
}
```

output:



```
2-csm-a@CseLab-02-62: ~/Desktop  
File Edit View Search Terminal Help  
2-csm-a@CseLab-02-62:~/Desktop$ cc halfduplex.c  
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out  
Parent Process - Writing to pipe - Message 1 is Hi  
Parent Process - Writing to pipe - Message 2 is Hello  
Child Process - Reading from pipe - Message 1 is Hi  
Child Process - Reading from pipe - Message 2 is Hello  
2-csm-a@CseLab-02-62:~/Desktop$
```

program-ii) implement a full duplex pipe

steps to achieve two-way communication :

Step 1 – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

Step 2 – Create a child process.

Step 3 – Close unwanted ends as only one end is needed for each communication.

Step 4 – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 – Perform the communication as required.

Algorithm

Step 1 – Create pipe1 for the parent process to write and the child process to read.

Step 2 – Create pipe2 for the child process to write and the parent process to read.

Step 3 – Close the unwanted ends of the pipe from the parent and child side.

Step 4 – Parent process to write a message and child process to read and display on the screen.

Step 5 – Child process to write a message and parent process to read and display on the screen.

Bi-directional Pipe

```
#include <error.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#define MAX 128

int main()

{

    int x[2],y[2],i,j,k,n,m;

    char msg[MAX];

    n=13;

    i=pipe(x);

    j=pipe(y);

    if((i==-1)||(j==-1))

    {

        perror("pipes creation failed");

        return -1;

    }

    k=fork();

    if(k==-1)

    {

        perror("child not created");

        return -1;

    }

    if(k>0)
```

```
{  
  
    close(x[0]);  
    close(y[1]);  
    write(x[1], "hello! child\n", 13);  
    sleep(1);  
    read(y[0], msg, n);  
    write(1, msg, n);  
}  
  
else  
{  
  
    close(x[1]);  
    close(y[0]);  
    m=read(x[0], msg, n);  
    write(1, msg, m);  
    write(y[1], "hai parent\n", 13);  
}  
    return 0;  
}
```

Expected OTUPUT:

In Parent: Writing to pipe 1 – Message is Hi

In Child: Reading from pipe 1 – Message is Hi

In Child: Writing to pipe 2 – Message is Hello

In Parent: Reading from pipe 2 – Message is Hello

program-iii)

AIM: Implement a uni directional FIFO

Algorithm:

Step 1 – Create two processes, one is fifoserver and another one is fifoclient.

Step 2 – Server process performs the following –

- Creates a named pipe (using system call mknod()) with name “MYFIFO”, if not created.
- Opens the named pipe for read only purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others. Waits infinitely for message from the Client.
- If the message received from the client is not “end”, prints the message. If the message is “end”, closes the fifo and ends the process.

Step 3 – Client process performs the following –

- Opens the named pipe for write only purposes.
- Accepts the string from the user.
- Checks, if the user enters “end” or other than “end”. Either way, it sends a message to the server. However, if the string is “end”, this closes the FIFO and also ends the process.
- Repeats infinitely until the user enters string “end”.

FIFO server file.

```
/* Filename: fifoserver.c */
```

```
#include <stdio.h>
```

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

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

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <string.h>
```

```
#define FIFO_FILE "MYFIFO"

int main()
{
    int fd;

    char readbuf[80];

    char end[10];

    int to_end;

    int read_bytes;

    /* Create the FIFO if it does not exist */
    mknod(FIFO_FILE, S_IFIFO|0640, 0);

    strcpy(end, "end");

    while(1)
    {
        fd = open(FIFO_FILE, O_RDONLY);

        read_bytes = read(fd, readbuf, sizeof(readbuf));

        readbuf[read_bytes] = '\0';

        printf("Received string: \"%s\" and length is %d\n", readbuf,
(int)strlen(readbuf));

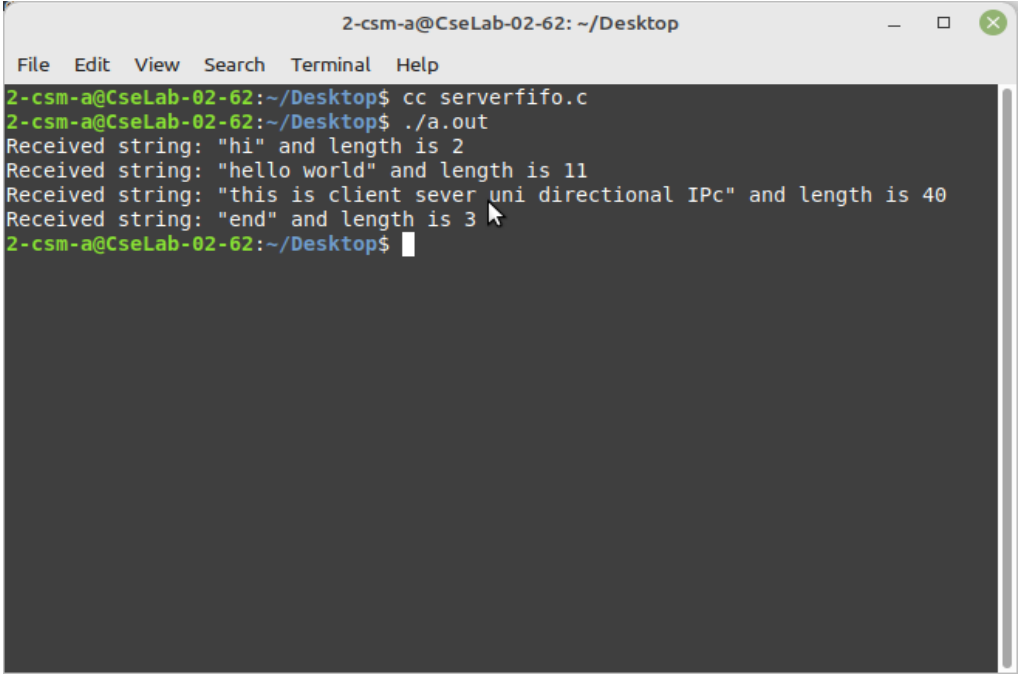
        to_end = strcmp(readbuf, end);

        if (to_end == 0) {
            close(fd);

            break;
        }
    }

    return 0;
}
```

OUTPUT:

A terminal window titled "2-csm-a@CseLab-02-62: ~/Desktop" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
2-csm-a@CseLab-02-62:~/Desktop$ cc serverfifo.c
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out
Received string: "hi" and length is 2
Received string: "hello world" and length is 11
Received string: "this is client sever uni directional IPc" and length is 40
Received string: "end" and length is 3
2-csm-a@CseLab-02-62:~/Desktop$
```

FIFO client

Program:

```
/* Filename: fifoclient.c */

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FIFO_FILE "MYFIFO"

int main() {

    int fd;

    int end_process;
```

```

int stringlen;

char readbuf[80];

char end_str[5];

printf("FIFO_CLIENT: Send messages, infinitely, to end enter \"end\"\\n");

fd = open(FIFO_FILE, O_CREAT|O_WRONLY);

strcpy(end_str, "end");

while (1) {

    printf("Enter string: ");

    fgets(readbuf, sizeof(readbuf), stdin);

    stringlen = strlen(readbuf);

    readbuf[stringlen - 1] = '\\0';

    end_process = strcmp(readbuf, end_str);

    //printf("end_process is %d\\n", end_process);

    if (end_process != 0) {

        write(fd, readbuf, strlen(readbuf));

        printf("Sent string: \"%s\" and string length is %d\\n", readbuf,
(int)strlen(readbuf));

    } else {

        write(fd, readbuf, strlen(readbuf));

        printf("Sent string: \"%s\" and string length is %d\\n", readbuf,
(int)strlen(readbuf));

        close(fd);

        break;

    }

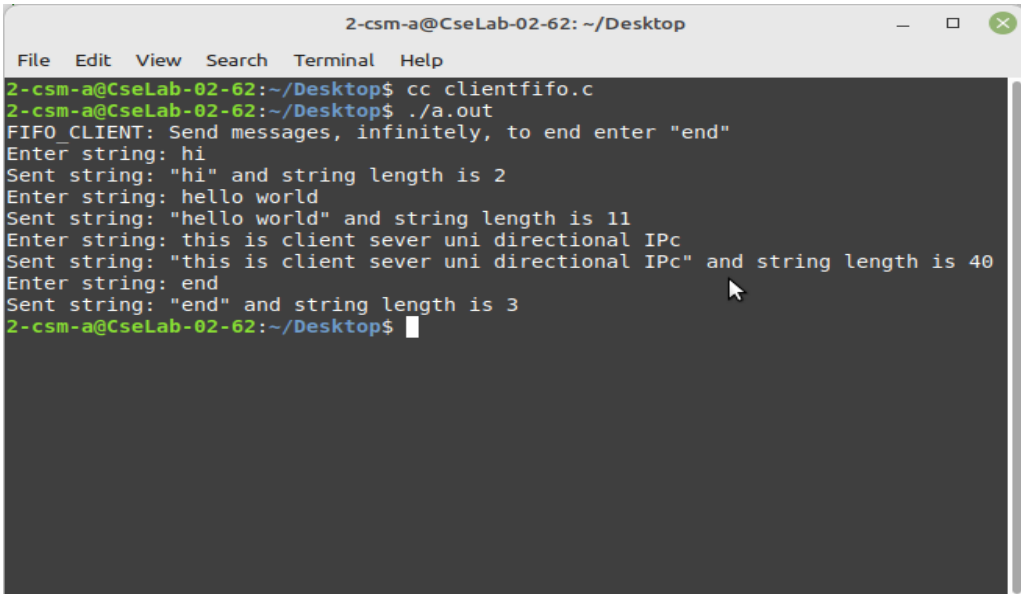
}

return 0;

}

```


Output:



```
2-csm-a@CseLab-02-62: ~/Desktop
File Edit View Search Terminal Help
2-csm-a@CseLab-02-62:~/Desktop$ cc clientfifo.c
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out
FIFO_CLIENT: Send messages, infinitely, to end enter "end"
Enter string: hi
Sent string: "hi" and string length is 2
Enter string: hello world
Sent string: "hello world" and string length is 11
Enter string: this is client sever uni directional IPC
Sent string: "this is client sever uni directional IPC" and string length is 40
Enter string: end
Sent string: "end" and string length is 3
2-csm-a@CseLab-02-62:~/Desktop$
```

program-iv)

AIM:Implement a Bi -directional FIFO

Step 1 – Create two processes, one is fifoserver_twoway and another one is fifoclient_twoway.

Step 2 – Server process performs the following –

Creates a named pipe (using library function mkfifo()) with name “fifo_twoway” in /tmp directory, if not created.

- Opens the named pipe for read and write purposes.
- Here, created FIFO with permissions of read and write for Owner. Read for Group and no permissions for Others.
- Waits infinitely for a message from the client.
- If the message received from the client is not “end”, prints the message and reverses the string. The reversed string is sent back to the client. If the message is “end”, closes the fifo and ends the process.

Step 3 – Client process performs the following –

- Opens the named pipe for read and write purposes.
- Accepts string from the user.
- Checks, if the user enters “end” or other than “end”. Either way, it sends a message to the server. However, if the string is “end”, this closes the FIFO and also ends the process.
- If the message is sent as not “end”, it waits for the message (reversed string) from the client and prints the reversed string.
- Repeats infinitely until the user enters the string “end”.

Program:

```
/* Filename: fifoclient_twoway.c */

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"

int main() {
    int fd;
    int end_process;
    int stringlen;
    int read_bytes;
    char readbuf[80];
    char end_str[5];

    printf("FIFO_CLIENT: Send messages, infinitely, to end enter\n");

    fd = open(FIFO_FILE, O_CREAT|O_RDWR);
    strcpy(end_str, "end");

    while (1) {
        printf("Enter string: ");
        fgets(readbuf, sizeof(readbuf), stdin);
        stringlen = strlen(readbuf);
        readbuf[stringlen - 1] = '\0';
        end_process = strcmp(readbuf, end_str);

        //printf("end_process is %d\n", end_process);
        if (end_process != 0) {
            write(fd, readbuf, strlen(readbuf));
        }
    }
}
```

```

        printf("FIFOCLIENT: Sent string: \"%s\" and string length
        is %d\n", readbuf, (int)strlen(readbuf));

        read_bytes = read(fd, readbuf, sizeof(readbuf));

        readbuf[read_bytes] = '\0';

        printf("FIFOCLIENT: Received string: \"%s\" and length
        is %d\n", readbuf, (int)strlen(readbuf));

    }

else {

    write(fd, readbuf, strlen(readbuf));

    printf("FIFOCLIENT: Sent string: \"%s\" and string length
    is %d\n", readbuf, (int)strlen(readbuf));

    close(fd);

    break;

}

}

return 0;

}

```

OUTPUT:

```

2-csm-a@CseLab-02-62: ~/Desktop
File Edit View Search Terminal Help
2-csm-a@CseLab-02-62:~/Desktop$ ls
4260      codeblocks.desktop  halfduplex.png  singlepipe.o
a.out     fullduplex.c        MYFIFO          uniclient.png
bidirclient.c  fullduplex.o        org.thonny.Thonny.desktop  vidirclient.c
bidirserver.c  google-chrome.desktop  serverfifo.c
clientfifo.c   halfduplex.c         serveruni.png
clientuni.png  halfduplex.o         singlepipe.c
2-csm-a@CseLab-02-62:~/Desktop$ cc bidirclient.c
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out
FIFO CLIENT: Send messages, infinitely, to end enter "end"
Enter string: hello
FIFOCLIENT: Sent string: "hello" and string length is 5
FIFOCLIENT: Received string: "olleh" and length is 5
Enter string: hi
FIFOCLIENT: Sent string: "hi" and string length is 2
FIFOCLIENT: Received string: "ih" and length is 2
Enter string: end
FIFOCLIENT: Sent string: "end" and string length is 3
2-csm-a@CseLab-02-62:~/Desktop$

```

```

/* Filename: fifoserver_twoway.c */

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#define FIFO_FILE "/tmp/fifo_twoway"

void reverse_string(char *);

int main() {

    int fd;

    char readbuf[80];

    char end[10];

    int to_end;

    int read_bytes;

    /* Create the FIFO if it does not exist */

    mkfifo(FIFO_FILE, S_IFIFO|0640);

    strcpy(end, "end");

    fd = open(FIFO_FILE, O_RDWR);

    while(1) {

        read_bytes = read(fd, readbuf, sizeof(readbuf));

        readbuf[read_bytes] = '\0';

        printf("FIFOSERVER: Received string: \"%s\" and length\n\n", readbuf, (int)strlen(readbuf));

        to_end = strcmp(readbuf, end);

        if (to_end == 0) {

            close(fd);

            break;

        }

    }

```

```

reverse_string(readbuf);

printf("FIFOSEVER: Sending Reversed String: \"%s\" and
      length is %d\n", readbuf, (int) strlen(readbuf));

write(fd, readbuf, strlen(readbuf));

/*

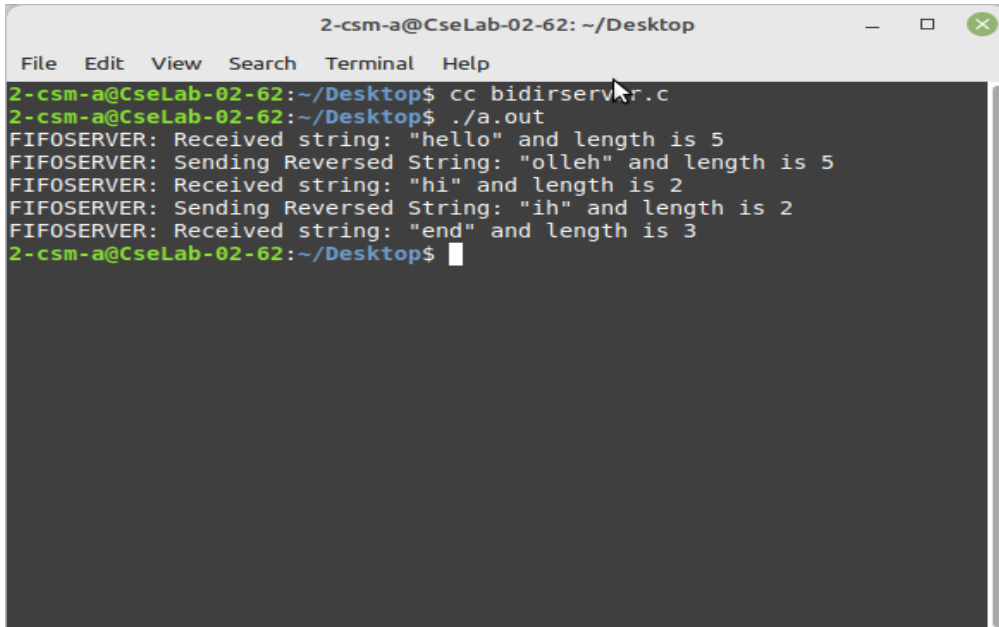
sleep - This is to make sure other process reads this, otherwise
this process would retrieve the message */

sleep(2);
}
return 0;
}

void reverse_string(char *str) {
    int last, limit, first;
    char temp;
    last = strlen(str) - 1;
    limit = last/2;
    first = 0;
    while (first < last) {
        temp = str[first];
        str[first] = str[last];
        str[last] = temp;
        first++;
        last--;
    }
    return;
}

```

OUTPUT:



```
2-csm-a@CseLab-02-62: ~/Desktop
File Edit View Search Terminal Help
2-csm-a@CseLab-02-62:~/Desktop$ cc bidirserver.c
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out
FIFOSEVER: Received string: "hello" and length is 5
FIFOSEVER: Sending Reversed String: "olleh" and length is 5
FIFOSEVER: Received string: "hi" and length is 2
FIFOSEVER: Sending Reversed String: "ih" and length is 2
FIFOSEVER: Received string: "end" and length is 3
2-csm-a@CseLab-02-62:~/Desktop$
```

program-v)

AIM: Implement a message queue

To perform communication using message queues, following are the steps –

Step 1 – Create a message queue or connect to an already existing message queue (msgget())

Step 2 – Write into message queue (msgsnd())

Step 3 – Read from the message queue (msgrcv())

Step 4 – Perform control operations on the message queue (msgctl())

Algorithm:

Step 1 – Create two processes, one is for sending into message queue (msgq_send.c) and another is for retrieving from the message queue (msgq_recv.c)

Step 2 – Creating the key, using ftok() function. For this, initially file msgq.txt is created to get a unique key.

Step 3 – The sending process performs the following.

- Reads the string input from the user
- Removes the new line, if it exists
- Sends into message queue
- Repeats the process until the end of input (CTRL + D)
- Once the end of input is received, sends the message “end” to signify the end of the process

Step 4 – In the receiving process, performs the following.

- Reads the message from the queue
- Displays the output
- If the received message is “end”, finishes the process and exits

Program:

```
/* Filename: msgq_send.c */  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
#include <errno.h>  
  
#include <string.h>  
  
#include <sys/types.h>  
  
#include <sys/ipc.h>  
  
#include <sys/msg.h>  
  
struct my_msgbuf {  
    long mtype;  
    char mtext[200];  
};  
  
int main(void)  
{  
    struct my_msgbuf buf;  
    int msqid;
```



```

key_t key;

if ((key = ftok("sender.c", 'B')) == -1) {
    perror("ftok");
    exit(1);
}

if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
    perror("msgget");
    exit(1);
}

printf("Enter lines of text, ^D to quit:\n");

buf.mtype = 1; /* we don't really care in this case */

while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
    int len = strlen(buf.mtext);

    /* remove newline at end, if it exists */
    if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

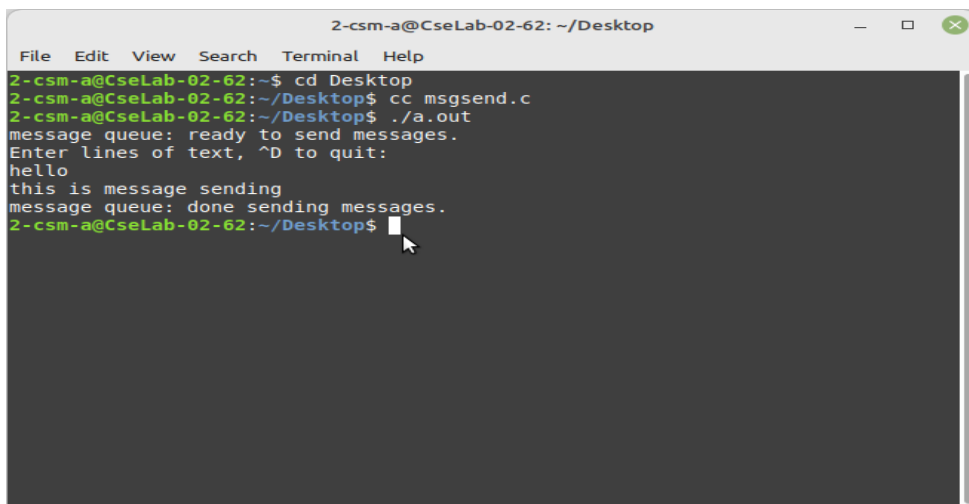
    if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
        perror("msgsnd");
}

if (msgctl(msqid, IPC_RMID, NULL) == -1) {
    perror("msgctl");
    exit(1);
}

return 0;
}

```

OUTPUT:



```
2-csm-a@CseLab-02-62: ~/Desktop
File Edit View Search Terminal Help
2-csm-a@CseLab-02-62:~$ cd Desktop
2-csm-a@CseLab-02-62:~/Desktop$ cc msgsend.c
2-csm-a@CseLab-02-62:~/Desktop$ ./a.out
message queue: ready to send messages.
Enter lines of text, ^D to quit:
hello
this is message sending
message queue: done sending messages.
2-csm-a@CseLab-02-62:~/Desktop$
```

/* Filename: msgq_recv.c */

Receiver :

```
#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct my_msgbuf {

    long mtype;

    char mtext[200];

};

int main(void)

{

    struct my_msgbuf buf;
```

```

int msqid;

key_t key;

if ((key = ftok("sender.c", 'B')) == -1) { /* same key as 2_send.c */

    perror("ftok");

    exit(1);

}


if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */

    perror("msgget");

    exit(1);

}

printf("ready to receive messages.\n");

for(;;) { /* never quits! */

    if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {

        perror("msgrcv");

        exit(1);

    }

    printf("received: \"%s\"\n", buf.mtext);

}

return 0;

}

```

<u>Compile :</u>

gcc sender.c -o ms

gcc reciever.c -o mr

./ms (run in first terminal)

```
# ./mr (run in second terminal)
```

Expcted-OUTPUT:

message queue: ready to receive messages.

rcvd: "this is line 1"

rcvd: "this is line 2"

rcvd: "end"

message queue: done receiving messages.

program-vi)

AIM : Implement shared memory

Description:

Shared memory is a memory shared between two or more processes. However, why do we need to share memory or some other means of communication?

To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques. As we are already aware, communication can be between related or unrelated processes.

Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.

Program: SHARED MEMORY FOR WRITER PROCESS

```
#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define BUF_SIZE 1024

#define SHM_KEY 0x1234

struct shmseg {
```

```

    int cnt;

    int complete;

    char buf[BUF_SIZE];
};

int fill_buffer(char * bufptr, int size);

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

    int shmid, numtimes;

    struct shmseg *shmp;

    char *bufptr;

    int spaceavailable;

    shmid = shmget(SHM_KEY, sizeof(struct shmseg),
0644|IPC_CREAT);

    if (shmid == -1) {

        perror("Shared memory");

        return 1;

    }

    // Attach to the segment to get a pointer to it.

    shmp = shmat(shmid, NULL, 0);

    if (shmp == (void *) -1) {

        perror("Shared memory attach");

        return 1;

    }

    /* Transfer blocks of data from buffer to shared memory */

    bufptr = shmp->buf;

    spaceavailable = BUF_SIZE;

    for (numtimes = 0; numtimes < 5; numtimes++) {

        shmp->cnt = fill_buffer(bufptr, spaceavailable);

        shmp->complete = 0;

        printf("Writing Process: Shared Memory Write: Wrote %d
bytes\n", shmp->cnt);

```

```

        bufptr = shmp->buf;
        spaceavailable = BUF_SIZE;
        sleep(3);
    }
    printf("Writing Process: Wrote %d times\n", numtimes);
    shmp->complete = 1;
    if (shmdt(shmp) == -1) {
        perror("shmdt");
        return 1;
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl");
        return 1;
    }
    printf("Writing Process: Complete\n");
    return 0;
}

int fill_buffer(char * bufptr, int size) {
    static char ch = 'A';
    int filled_count;
    //printf("size is %d\n", size);
    memset(bufptr, ch, size - 1);
    bufptr[size-1] = '\0';
    if (ch > 122)
        ch = 65;
    if ( (ch >= 65) && (ch <= 122) ) {
        if ( (ch >= 91) && (ch <= 96) ) {
            ch = 65;
        }
    }
}

```

```

    }

    filled_count = strlen(bufptr);

    //printf("buffer count is: %d\n", filled_count);

    //printf("buffer filled is:%s\n", bufptr);

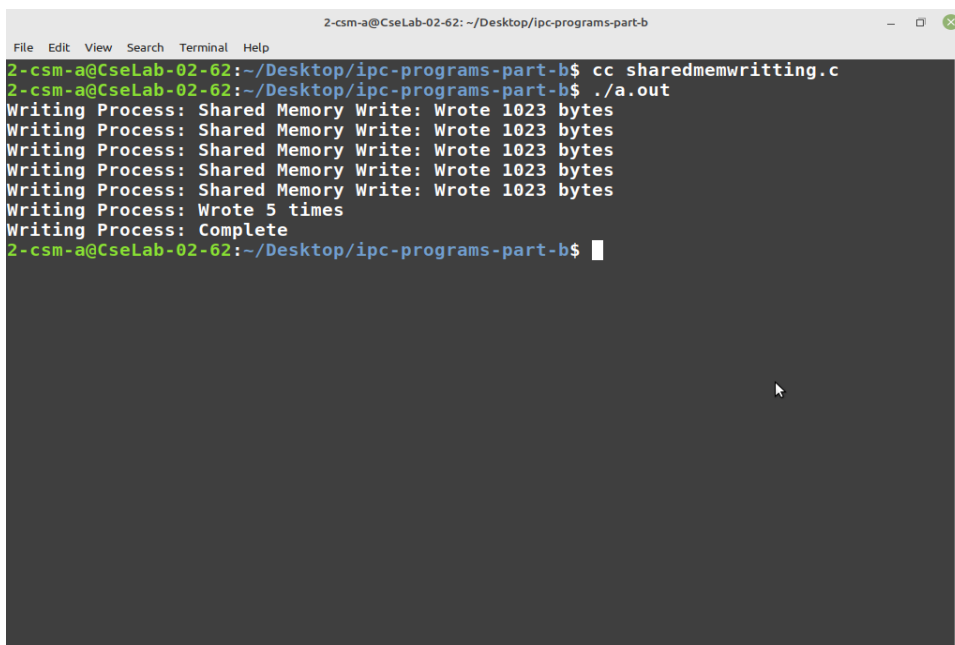
    ch++;

    return filled_count;

}

```

OUTPUT:



```

2-csm-a@CseLab-02-62: ~/Desktop/ipc-programs-part-b
File Edit View Search Terminal Help
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$ cc sharedmemwritting.c
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$ ./a.out
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Shared Memory Write: Wrote 1023 bytes
Writing Process: Wrote 5 times
Writing Process: Complete
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$

```

SHARED MEMORY FOR READ PROCESS:

program:

```

#include <iostream>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

using namespace std;

```



```

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}

```

Expected output:

segment contains :

```

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA

```

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA

```

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA

```

shared Memory: Read 1023 byte

program-vii)

AIM: Demonstrate semaphores

Description:

To protect the critical/common region shared among multiple processes.

multiple processes are using the same region of code and if all want to access parallelly then the outcome is overlapped. Say, for example, multiple users are using one printer only (common/critical section), say 3 users, given 3 jobs at same time, if all the jobs start parallelly, then one user output is overlapped with another. So, we need to protect that using semaphores i.e., locking the critical section when one process is running and unlocking when it is done. This would be repeated for each user/process so that one job is not overlapped with another job.

Basically semaphores are classified into two types –

Binary Semaphores – Only two states 0 & 1, i.e., locked/unlocked or available/unavailable, Mutex implementation.

Counting Semaphores – Semaphores which allow arbitrary resource count are called counting semaphores.

check the shared memory writing program.

```
#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define SHM_KEY 0x12345

struct shmseg {
```

```
    int cntr;

    int write_complete;

    int read_complete;

};

void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int
total_count);

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

    int shmid;

    struct shmseg *shmp;

    char *bufptr;

    int total_count;

    int sleep_time;

    pid_t pid;

    if (argc != 2)

        total_count = 10000;

    else {

        total_count = atoi(argv[1]);

        if (total_count < 10000)

            total_count = 10000;

    }

    printf("Total Count is %d\n", total_count);

    shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT);

    if (shmid == -1) {

        perror("Shared memory");

        return 1;

    }
```

```

// Attach to the segment to get a pointer to it.

shmp = shmat(shmid, NULL, 0);

if (shmp == (void *) -1) {

    perror("Shared memory attach");

    return 1;

}

shmp->cntr = 0;

pid = fork();

/* Parent Process - Writing Once */

if (pid > 0) {

    shared_memory_cntr_increment(pid, shmp, total_count);

} else if (pid == 0) {

    shared_memory_cntr_increment(pid, shmp, total_count);

    return 0;

} else {

    perror("Fork Failure\n");

    return 1;

}

while (shmp->read_complete != 1)

sleep(1);

if (shmdt(shmp) == -1) {

    perror("shmdt");

    return 1;

}

if (shmctl(shmid, IPC_RMID, 0) == -1) {

    perror("shmctl");

```

```

        return 1;
    }

    printf("Writing Process: Complete\n");

    return 0;
}

/* Increment the counter of shared memory by total_count in steps of 1 */

void shared_memory_cntr_increment(int pid, struct shmseg *shmp, int
total_count) {
    int cntr;

    int numtimes;

    int sleep_time;

    cntr = shmp->cntr;

    shmp->write_complete = 0;

    if (pid == 0)

        printf("SHM_WRITE: CHILD: Now writing\n");

    else if (pid > 0)

        printf("SHM_WRITE: PARENT: Now writing\n");

    //printf("SHM_CNTR is %d\n", shmp->cntr);

    /* Increment the counter in shared memory by total_count in steps of 1 */

    for (numtimes = 0; numtimes < total_count; numtimes++) {

        cntr += 1;

        shmp->cntr = cntr;

        /* Sleeping for a second for every thousand */

        sleep_time = cntr % 1000;

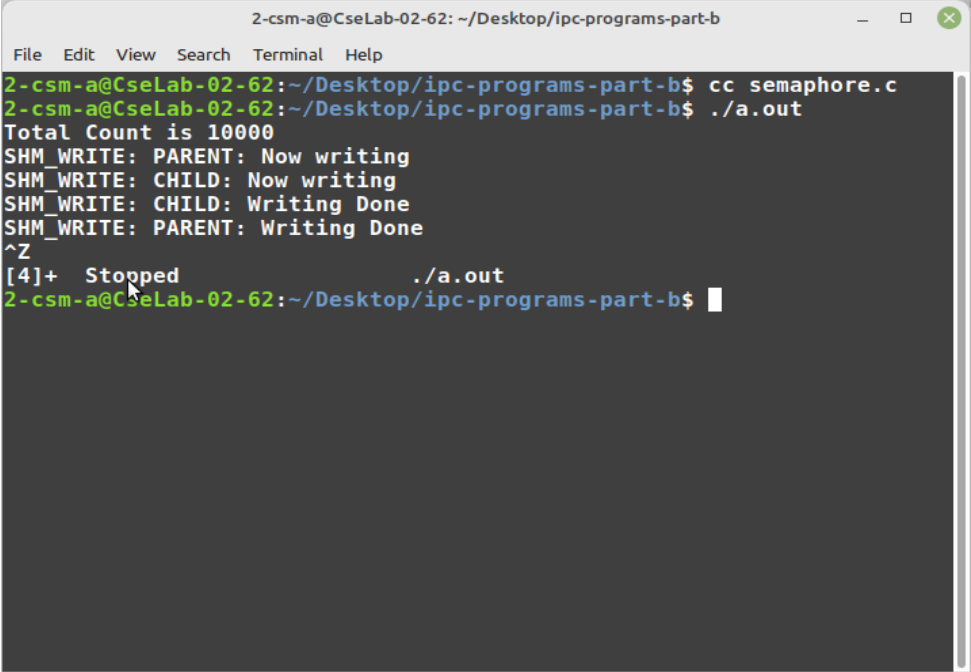
        if (sleep_time == 0)

            sleep(1);

```

```
}  
  
shmp->write_complete = 1;  
  
if (pid == 0)  
  
    printf("SHM_WRITE: CHILD: Writing Done\n");  
  
else if (pid > 0)  
  
    printf("SHM_WRITE: PARENT: Writing Done\n");  
  
return;  
  
}
```

output:

A terminal window titled "2-csm-a@CseLab-02-62: ~/Desktop/ipc-programs-part-b" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the compilation and execution of a C program. The output includes the total count (10000) and status messages for parent and child processes writing to shared memory.

```
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$ cc semaphore.c  
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$ ./a.out  
Total Count is 10000  
SHM_WRITE: PARENT: Now writing  
SHM_WRITE: CHILD: Now writing  
SHM_WRITE: CHILD: Writing Done  
SHM_WRITE: PARENT: Writing Done  
^Z  
[4]+  Stopped                  ./a.out  
2-csm-a@CseLab-02-62:~/Desktop/ipc-programs-part-b$
```

LAB CYLCE -II

B.Practice the following local IPC programs

program-i) develop an Iterative Client Server set up for DayTime service

/* SERVER-program */

Description:

An iterative server processes requests from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

using create(), Create TCP socket. using bind(), Bind the socket to server address. using accept(), At this point, connection is established between client and server, and they are ready to transfer data.

Daytime is a **standard TCP/IP service for determining the date and time on a given machine**. The service is defined for both the TCP and UDP communication protocols. In this example, we will build a client and server for the daytime service in Java using UDP. A daytime server listens for client requests on port 13.

Server:

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <fcntl.h>
```

```
#include <time.h>
```

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

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

```
#include <netinet/in.h>
```

```
#include <time.h>
```

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

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

```
#include <string.h>
```

```

#include <strings.h>

#include <unistd.h>

#define MAXLINE 100

#define LISTENQ 1024

#define SA struct sockaddr

int main(int argc, char *argv[])
{
    int listenfd, connfd;

    struct sockaddr_in servaddr;

    char buff[MAXLINE];

    time_t ticks;

    printf("server program ...\n");

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    printf("socket created ...\n");

    bzero(&servaddr, sizeof(servaddr));

    printf("socket address structure cleared ...\n");

    servaddr.sin_family = AF_INET;

    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    servaddr.sin_port = htons(13); /* daytime server */

    printf("socket address structure filled ...\n");

    bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    printf("socket bound to IP & port13 .. ...\n");

    listen(listenfd, LISTENQ);

    printf("server ready & listening..for clients calls.. ...\n");

    for ( ; ; )
    {

```



```

        connfd = accept(listenfd, (SA *) NULL, NULL);

        printf("client connected.....\n");

        ticks = time(NULL);

        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));

        printf("local time gathered..& writing to client..\n");

        write(connfd, buff, strlen(buff));

        printf("time sent to client ...\n");

        close(connfd);

        printf("client connection terminated ...\n");

    }

}

```

Client :

```

#include <stdio.h>

#include <errno.h>

#include <fcntl.h>

#include <time.h>

#include <sys/time.h>

#include <sys/stat.h>

#include <netinet/in.h>

#include <time.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <string.h>

#include <strings.h>

#include <unistd.h>

#define SA struct sockaddr

```

```

#define MAXLINE 100

int main(int argc, char *argv[])
{
    int sockfd, n;

    char recvline[MAXLINE + 1];

    struct sockaddr_in servaddr;

    if (argc != 2)
    {
        perror("usage: dtc <IPaddress>");
        return -1;
    }

    printf("client program ...\n");

    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket error");
        return -1;
    }

    printf("client socket created...\n");

    bzero(&servaddr, sizeof(servaddr));

    printf("socket address structure cleared for use.....\n");

    servaddr.sin_family = AF_INET;

    servaddr.sin_port = htons(13); /* daytime server */

    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
    {
        printf("inet_pton error for %s", argv[1]);

        perror("inet pton error:");

        return -1;
    }

```

```

}

printf("socket address structure filled with..serv IP, port 13 ....\n");
if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
{
    perror("connect error");
    return -1;
}

printf("connection established with server ..... \n");
printf("reading the time sent from server  ... \n");
printf("& the time is ..... \n");
while ( (n = read(sockfd, recvline, MAXLINE)) > 0)
{
    recvline[n] = 0; /* null terminate */
    if (fputs(recvline, stdout) == EOF)
    {
        perror("fputs error");
        return -1;
    }
}

if (n < 0)
{
    perror("read error");
    return -1;
}

return 0;
}

```

Compile :

```
# gcc dts.c -o dts
```

```
# gcc dtc.c -o dtc
```

```
# ./dts (run in first terminal)
```

```
# ./dtc 127.0.0.1 (run in second terminal)
```

program-ii)

AIM:Develop an Iterative Client Server set up for echo service

Description:

An iterative server processes requests from clients in a serial manner; one connection is served and responded to before the server accepts a new client connection.

An iterative server handles both the connection request and the transaction involved in the call itself. Iterative servers are fairly simple and are suitable for transactions that do not last long. TCP Echo Client.

An echo server is a programme that sends you back the same message you just sent it. For it to work, you will need a client, and a server. The client is what will send the message, and where you will see the response appear. The server receives the message and sends it back to the client.

EchoServer & EchoClient

Header File

Save this as myio.h

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
#include <fcntl.h>
```

```
#include <stdlib.h>
```

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

```
#include <netinet/in.h>
```

```
#include <time.h>
```

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

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

```
#include <string.h>
```

```

#include <unistd.h>

/* Read "n" bytes from a descriptor. */

int readn(int fd, void *vptr, int n)
{
    int nleft;
    int nread;
    char *ptr;

    ptr = vptr;
    nleft = n;

    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0; /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break; /* EOF */
        nleft -= nread;
        ptr += nread;
    }

    return (n - nleft); /* return >= 0 */
}

/* Write "n" bytes to a descriptor. */

int  writen(int fd, const void *vptr, int n)
{
    int nleft;

```

```

int nwritten;

const char *ptr;

ptr = vptr;

nleft = n;
while (nleft > 0)
{
    if ( (nwritten = write(fd, ptr, nleft)) <= 0)
    {
        if (nwritten < 0 && errno == EINTR)
            nwritten=0; /* and call write() again */
        else
            return -1; /* error */
    }
    nleft -= nwritten;
    ptr += nwritten;
}

return (n);
}

/* PAINFULLY SLOW VERSION -- example only */

int
readline(int fd, void *vptr, size_t maxlen)
{
    int n, rc;
    char c, *ptr;

    ptr = vptr;

    for (n = 1; n < maxlen; n++) {

```

again:

```
    if ( (rc = read(fd, &c, 1)) == 1) {  
        *ptr++ = c;  
        if (c == '\n')  
            break; /* newline is stored, like fgets() */  
    } else if (rc == 0) {  
        *ptr = 0;  
        return (n - 1); /* EOF, n - 1 bytes were read */  
    } else {  
        if (errno == EINTR)  
            goto again;  
        return (-1); /* error, errno set by read() */  
    }  
}  
  
    *ptr = 0; /* null terminate like fgets() */  
  
return (n);  
  
}
```

Server :

```
#include <stdio.h>  
  
#include <errno.h>  
  
#include <fcntl.h>  
  
#include <time.h>  
  
#include <sys/time.h>  
  
#include <sys/stat.h>  
  
#include <netinet/in.h>  
  
#include <time.h>
```

```

#include <sys/types.h>

#include <sys/socket.h>

#include <string.h>

#include <strings.h>

#include <unistd.h>

#include <stdlib.h>

#include "myio.h"

#define MAX 100

#define LISTENQ 1024

#define SA struct sockaddr

#define SERV_PORT 9234


void str_echo(int sockfd)
{
    int n;
    char buf[MAX];
    again:
        while((n=read(sockfd,buf,MAX))>0)
            writen(sockfd,buf,n);
        if(n<0 && errno==EINTR)
            goto again;
        else if(n<0)
            perror("str_echo:read error");
}

int main(int argc, char **argv)
{

```



```

int listenfd, connfd,pid,clilen;

struct sockaddr_in servaddr,cliaddr;

char buff[MAX];

time_t ticks;

listenfd = socket(AF_INET, SOCK_STREAM, 0);

printf("socket created\n");

bzero(&servaddr, sizeof(servaddr));

bzero(&cliaddr, sizeof(cliaddr));

printf("socket address structure cleared\n");

servaddr.sin_family = AF_INET;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

servaddr.sin_port = htons(9234); /* echo server */

bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

printf("binding successful\n");

listen(listenfd, LISTENQ);

printf("socket made listening \n");

for ( ; ; )

{

    clilen=sizeof(cliaddr);

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

    printf("client with IP %s and port %d connected\n",inet_ntop(AF_INET,&cliaddr.sin_addr,buff,sizeof(buff)),ntohs(cliaddr.sin_port));

    if((pid=fork())==0)

    {

        close(listenfd);

        str_echo(connfd);

```

```

                                exit(0);
                                }
                                close(connfd);
                                }
                                }

```

Client :

```

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <time.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <stdlib.h>
#include "myio.h"
#define MAX 100
#define SA struct sockaddr
#define SERV_PORT 9234
void str_cli(FILE *fp,int sockfd)
{

```

```

        char sendline[MAX],recvline[MAX];

        while(fgets(sendline,MAX,fp)!=NULL)
        {
                writen(sockfd,sendline,strlen(sendline));

                if(readline(sockfd,recvline,MAX)==0)

                        printf("str_cli:error:server terminated
prematurely");

                fputs(recvline,stdout);

        }
}

int main(int argc, char **argv)
{
        int sockfd, n,i;

        struct sockaddr_in servaddr;

        if (argc != 2)
        {

                perror("usage: ec <IPaddress>");return -1;

        }

        if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {

                perror("socket error");

                return -1;

        }

        bzero(&servaddr, sizeof(servaddr));

        servaddr.sin_family = AF_INET;

        servaddr.sin_port = htons(SERV_PORT); /* echo server */

        if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)

```

```
{  
  
    printf("inet_pton error for %s", argv[1]);  
  
    perror("inet_pton");  
  
    return -1;  
  
}  
  
if ((i=connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)  
  
    {  
  
        perror("connect error");return -1;  
  
    }  
  
str_cli(stdin,sockfd);  
  
exit(0);  
  
}
```

<u>Compile :</u>
gcc echoserver.c -o es
gcc echoclient.c -o ec
./es (run in first terminal)
./ec 127.0.0.1 (run in second terminal)

program-iii)

AIM:Develop a concurrent Client Server set up for DayTime service

Description:

A concurrent server handles multiple clients at the same time.**Fork()** creates a **new child process** that runs in sync with its **Parent process** and returns **0** if child process is created successfully.

Whenever a new client will attempt to connect to the TCP server, we will create a new **Child Process** that is going to run in parallel with other clients' execution. In this way, we are going to design a concurrent server without using the **Select() system call**.

A **pid_t (Process id) data type** will be used to hold the Child's process id. Example: **pid_t = fork()**.

This is the simplest technique for creating a concurrent server. Whenever a new client connects to the server, a **fork()** call is executed making a new child process for each new client.

Multi-Threading achieves a concurrent server using a single processed program. Sharing of data/files with connections is usually slower with a **fork()** than with threads.

Select() system call doesn't create multiple processes. Instead, it helps in **multiplexing** all the clients on a single program and doesn't need **non-blocking IO**.

Program to design a concurrent server for handling multiple clients using fork()

Accepting a client makes a new child process that runs concurrently with other clients and the parent process

```
// Accept connection request from client in cliAddr
// socket structure
clientSocket = accept(
    sockfd, (struct sockaddr*)&cliAddr, &addr_size);
// Make a child process by fork() and check if child
```

```

// process is created successfully
if ((childpid = fork()) == 0) {
    // Send a confirmation message to the client for
    // successful connection
    send(clientSocket, "hi client", strlen("hi client"),
        0);
}

```

SERVER

```

// Server side program that sends
// a 'hi client' message
// to every client concurrently
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
// PORT number
#define PORT 4444
int main()
{
    // Server socket id
    int sockfd, ret;
    // Server socket address structures
    struct sockaddr_in serverAddr;
    // Client socket id
    int clientSocket;

```

```

// Client socket address structures
struct sockaddr_in cliAddr;

// Stores byte size of server socket address
socklen_t addr_size;

// Child process id
pid_t childpid;

// Creates a TCP socket id from IPV4 family
sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Error handling if socket id is not valid
if (sockfd < 0) {
    printf("Error in connection.\n");
    exit(1);
}

printf("Server Socket is created.\n");

// Initializing address structure with NULL
memset(&serverAddr, '\0',
    sizeof(serverAddr));

// Assign port number and IP address
// to the socket created
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(PORT);

// 127.0.0.1 is a loopback address
serverAddr.sin_addr.s_addr
    = inet_addr("127.0.0.1");

// Binding the socket id with
// the socket structure
ret = bind(sockfd,
    (struct sockaddr*)&serverAddr,
    sizeof(serverAddr));

```

```
// Error handling
if (ret < 0) {
    printf("Error in binding.\n");
    exit(1);
}

// Listening for connections (upto 10)
if (listen(sockfd, 10) == 0) {
    printf("Listening...\n\n");
}

int cnt = 0;
while (1) {
    // Accept clients and
    // store their information in cliAddr
    clientSocket = accept(
        sockfd, (struct sockaddr*)&cliAddr,
        &addr_size);

    // Error handling
    if (clientSocket < 0) {
        exit(1);
    }

    // Displaying information of
    // connected client
    printf("Connection accepted from %s:%d\n",
        inet_ntoa(cliAddr.sin_addr),
        ntohs(cliAddr.sin_port));

    // Print number of clients
    // connected till now
    printf("Clients connected: %d\n\n",
```



```

        ++cnt);

// Creates a child process
if ((childpid = fork()) == 0) {
    // Closing the server socket id
    close(sockfd);

    // Send a confirmation message
    // to the client
    send(clientSocket, "hi client",
          strlen("hi client"), 0);
}
}

// Close the client socket id
close(clientSocket);

return 0;
}

```

CLIENT:

```

// Client Side program to test
// the TCP server that returns
// a 'hi client' message
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

// PORT number
#define PORT 4444

```

```

int main()
{
    // Socket id
    int clientSocket, ret;

    // Client socket structure
    struct sockaddr_in cliAddr;

    // char array to store incoming message
    char buffer[1024];

    // Creating socket id
    clientSocket = socket(AF_INET,
                          SOCK_STREAM, 0);

    if (clientSocket < 0) {
        printf("Error in connection.\n");
        exit(1);
    }

    printf("Client Socket is created.\n");

    // Initializing socket structure with NULL
    memset(&cliAddr, '\0', sizeof(cliAddr));

    // Initializing buffer array with NULL
    memset(buffer, '\0', sizeof(buffer));

    // Assigning port number and IP address
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);

    // 127.0.0.1 is Loopback IP
    serverAddr.sin_addr.s_addr
        = inet_addr("127.0.0.1");

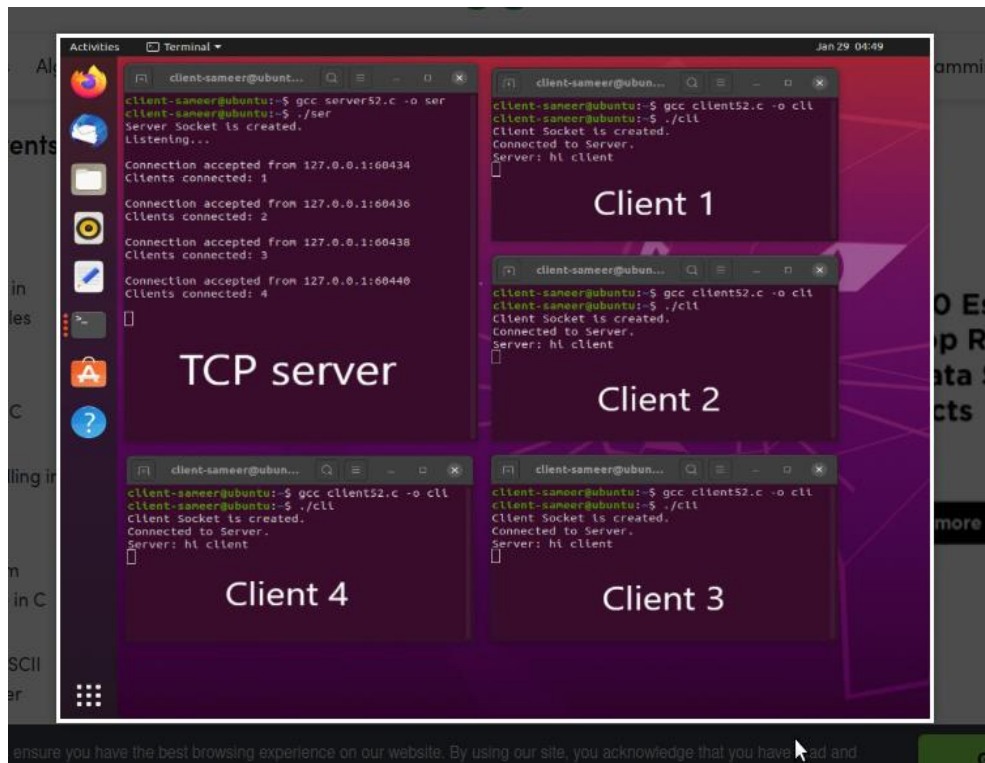
    // connect() to connect to the server
    ret = connect(clientSocket,

```

```
        (struct sockaddr*)&serverAddr,
        sizeof(serverAddr));
if (ret < 0) {
    printf("Error in connection.\n");
    exit(1);
}
printf("Connected to Server.\n");
while (1) {
    // recv() receives the message
    // from server and stores in buffer
    if (recv(clientSocket, buffer, 1024, 0)
        < 0) {
        printf("Error in receiving data.\n");
    }
    // Printing the message on screen
    else {
        printf("Server: %s\n", buffer);
        bzero(buffer, sizeof(buffer));
    }
}

return 0;
}
```

Expeced-ouput:



The screenshot displays a Linux desktop environment with a terminal window open. The terminal shows the execution of a C program that implements a TCP server and four clients. The server, labeled 'TCP server', is running on port 52 and is listening for connections. It has successfully accepted four connections from 127.0.0.1. The four clients, labeled 'Client 1', 'Client 2', 'Client 3', and 'Client 4', are also running on port 52 and have successfully connected to the server. Each client window shows the output of the program, including the creation of the client socket, the connection to the server, and the server's response 'hi client'.

```
client-sameer@ubuntu:~$ gcc server52.c -o ser
client-sameer@ubuntu:~$ ./ser
Server socket is created.
Listening...
Connection accepted from 127.0.0.1:60434
Clients connected: 1
Connection accepted from 127.0.0.1:60436
Clients connected: 2
Connection accepted from 127.0.0.1:60438
Clients connected: 3
Connection accepted from 127.0.0.1:60440
Clients connected: 4
[]

Client 1
client-sameer@ubuntu:~$ gcc client52.c -o cli
client-sameer@ubuntu:~$ ./cli
Client socket is created.
Connected to Server.
Server: hi client
[]

Client 2
client-sameer@ubuntu:~$ gcc client52.c -o cli
client-sameer@ubuntu:~$ ./cli
Client socket is created.
Connected to Server.
Server: hi client
[]

Client 3
client-sameer@ubuntu:~$ gcc client52.c -o cli
client-sameer@ubuntu:~$ ./cli
Client socket is created.
Connected to Server.
Server: hi client
[]

Client 4
client-sameer@ubuntu:~$ gcc client52.c -o cli
client-sameer@ubuntu:~$ ./cli
Client socket is created.
Connected to Server.
Server: hi client
[]
```

program-iv)

AIM:Develop a concurrent Client Server set up for echo service

Description:

Echo is a standard TCP/IP service used primarily for testing reachability, debugging software, and identifying routing problems. The service is defined for both the TCP and UDP communication protocols.

A host may connect to a server that supports the Echo Protocol using the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) on the well-known **port number 7**.

program-v)

AIM:Develop a concurrent Client Server set up for reversing the given string.

Description:

first setup client-server connection. When connection will setup, client will send user input string to server by send system call. At server side, server will wait for string sent by client. Server read string by read system call. After this, server will reverse the string and send back to client.

// C client code to send string to reverse

```
#include <arpa/inet.h>
```

```
#include <netinet/in.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

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

```
#include <unistd.h>
```

```
#define PORT 8090
```

```
// Driver code
```

```
int main()
```

```

{
    struct sockaddr_in address;
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char str[100];
    printf("\nInput the string:");
    scanf("%[^\\n]s", str);
    char buffer[1024] = { 0 };

    // Creating socket file descriptor
    if ((sock = socket(AF_INET,
                      SOCK_STREAM, 0))
        < 0) {
        printf("\n Socket creation error \\n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from
    // text to binary form 127.0.0.1 is local
    // host IP address, this address should be
    // your system local host IP address
    if (inet_pton(AF_INET, "127.0.0.1",

```

```
        &serv_addr.sin_addr)

    <= 0) {

    printf("\nAddress not supported \n");

    return -1;

}

// connect the socket

if (connect(sock, (struct sockaddr*)&serv_addr,

        sizeof(serv_addr))

    < 0) {

    printf("\nConnection Failed \n");

    return -1;

}

int l = strlen(str);

// send string to server side

send(sock, str, sizeof(str), 0);

// read string sent by server

valread = read(sock, str, l);

printf("%s\n", str);

return 0;

}
```

```

// Server C code to reverse a
// string by sent from client

#include <netinet/in.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <unistd.h>

#define PORT 8090

// Driver code

int main()
{
    int server_fd, new_socket, valread;

    struct sockaddr_in address;

    char str[100];

    int addrlen = sizeof(address);

    char buffer[1024] = { 0 };

    char* hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET,
                            SOCK_STREAM, 0)) == 0) {

        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;

    address.sin_addr.s_addr = INADDR_ANY;

    address.sin_port = htons(PORT);

    // Forcefully attaching socket to

```



```

// the port 8090
if (bind(server_fd, (struct sockaddr*)&address,
        sizeof(address)) < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE);
}
// puts the server socket in passive mode
if (listen(server_fd, 3) < 0) {
    perror("listen");
    exit(EXIT_FAILURE);
}
if ((new_socket = accept(server_fd,
        (struct sockaddr*)&address,
        (socklen_t*)&addrlen)) < 0) {
    perror("accept");
    exit(EXIT_FAILURE);
}
// read string send by client
valread = read(new_socket, str,
        sizeof(str));

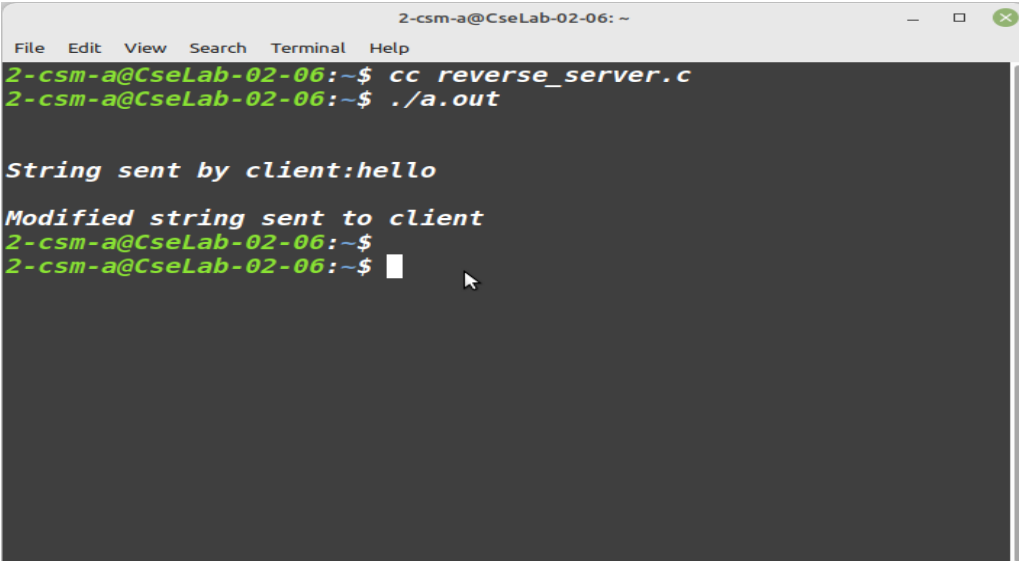
int i, j, temp;
int l = strlen(str);
printf("\nString sent by client:%s\n", str);

// loop to reverse the string
for (i = 0, j = l - 1; i < j; i++, j--) {
    temp = str[i];
    str[i] = str[j];

```

```
        str[j] = temp;
    }
    // send reversed string to client
    // by send system call
    send(new_socket, str, sizeof(str), 0);
    printf("\nModified string sent to client\n");
    return 0;
}
```

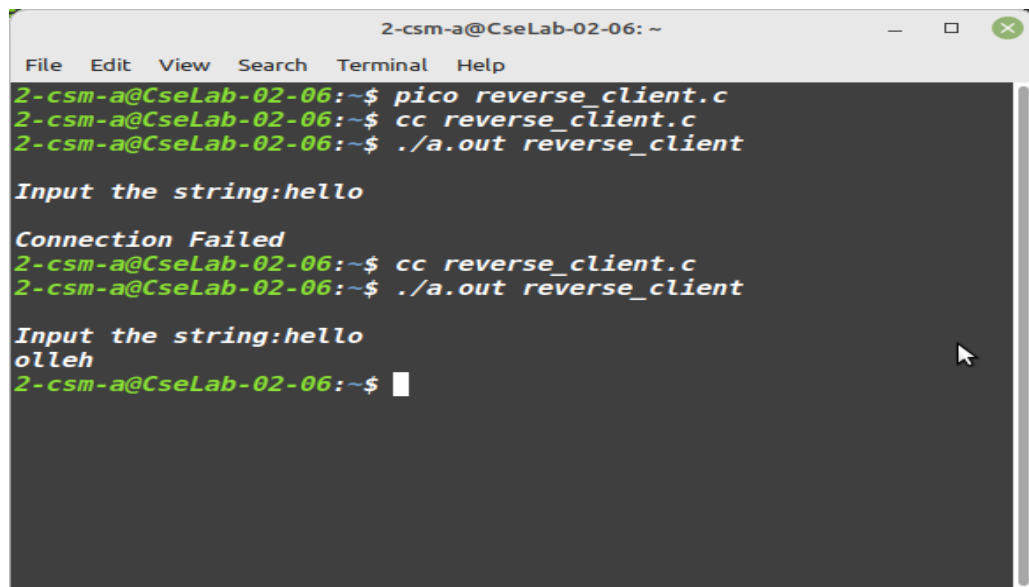
Output:



```
2-csm-a@CseLab-02-06: ~
File Edit View Search Terminal Help
2-csm-a@CseLab-02-06:~$ cc reverse_server.c
2-csm-a@CseLab-02-06:~$ ./a.out

String sent by client:hello

Modified string sent to client
2-csm-a@CseLab-02-06:~$
2-csm-a@CseLab-02-06:~$
```



A terminal window titled "2-csm-a@CseLab-02-06: ~" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following sequence of commands and output:

```
2-csm-a@CseLab-02-06:~$ pico reverse_client.c
2-csm-a@CseLab-02-06:~$ cc reverse_client.c
2-csm-a@CseLab-02-06:~$ ./a.out reverse_client

Input the string:hello

Connection Failed
2-csm-a@CseLab-02-06:~$ cc reverse_client.c
2-csm-a@CseLab-02-06:~$ ./a.out reverse_client

Input the string:hello
olleh
2-csm-a@CseLab-02-06:~$
```

The terminal has a dark background with green text for prompts and standard white text for output. A mouse cursor is visible on the right side of the terminal area.

program-vi)

AIM:Develop a TCP Client Server set up for transfer File service

Description:

1. Server Action: First need to run server application, this server application will open an endpoint with predefined IP address and port number and will remain in listen mode to accept new socket connection request from client. ...
2. Client Action: Now turn is coming to client to request server.

When the sending TCP wants to establish connections, it sends a segment called a SYN to the peer TCP protocol running on the receiving host. The receiving TCP returns a segment called an ACK to acknowledge the successful receipt of the segment. The sending TCP sends another ACK segment, then proceeds to send the data.

SERVER:

```
#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <sys/sendfile.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <netinet/in.h>

#define PATH_MAX 255

int main(int argc, char **argv)

{
```

```

int port = 1234;          /* port number to use */

int sock;                 /* socket descriptor */

int desc;                 /* file descriptor for socket */

int fd;                   /* file descriptor for file to send */

struct sockaddr_in addr;  /* socket parameters for bind */

struct sockaddr_in addr1; /* socket parameters for accept */

int    addrlen;           /* argument to accept */

struct stat stat_buf;     /* argument to fstat */

off_t offset = 0;        /* file offset */

char filename[PATH_MAX]; /* filename to send */

int rc;                   /* holds return code of system calls */

/* check command line arguments, handling an optional port number */
if (argc == 2) {
    port = atoi(argv[1]);

    if (port <= 0) {
        fprintf(stderr, "invalid port: %s\n", argv[1]);
        exit(1);
    }
} else if (argc != 1) {
    fprintf(stderr, "usage: %s [%s]\n", argv[0], port);
    exit(1);
}

/* create Internet domain socket */

sock = socket(AF_INET, SOCK_STREAM, 0);

if (sock == -1) {
    fprintf(stderr, "unable to create socket: %s\n", strerror(errno));
}

```

```

        exit(1);
    }

    /* fill in socket structure */

    memset(&addr, 0, sizeof(addr));

    addr.sin_family = AF_INET;

    addr.sin_addr.s_addr = INADDR_ANY;

    addr.sin_port = htons(port);

    /* bind socket to the port */

    rc = bind(sock, (struct sockaddr *)&addr, sizeof(addr));

    if (rc == -1) {

        fprintf(stderr, "unable to bind to socket: %s\n", strerror(errno));

        exit(1);

    }

    /* listen for clients on the socket */

    rc = listen(sock, 1);

    if (rc == -1) {

        fprintf(stderr, "listen failed: %s\n", strerror(errno));

        exit(1);

    }

    while (1) {

        /* wait for a client to connect */

        desc = accept(sock, (struct sockaddr *)&addr1, &addrlen);

        if (desc == -1) {

            fprintf(stderr, "accept failed: %s\n", strerror(errno));

            exit(1);

        }
    }

```

```

/* get the file name from the client */

rc = recv(desc, filename, sizeof(filename), 0);

if (rc == -1) {

    fprintf(stderr, "recv failed: %s\n", strerror(errno));

    exit(1);

}

/* null terminate and strip any \r and \n from filename */

    filename[rc] = '\0';

if (filename[strlen(filename)-1] == '\n')

    filename[strlen(filename)-1] = '\0';

if (filename[strlen(filename)-1] == '\r')

    filename[strlen(filename)-1] = '\0';

/* exit server if filename is "quit" */

if (strcmp(filename, "quit") == 0) {

    fprintf(stderr, "quit command received, shutting down server\n");

    break;

}

fprintf(stderr, "received request to send file %s\n", filename);

/* open the file to be sent */

fd = open(filename, O_RDONLY);

if (fd == -1) {

    fprintf(stderr, "unable to open '%s': %s\n", filename, strerror(errno));

    exit(1);

}

/* get the size of the file to be sent */

fstat(fd, &stat_buf);

```

```

    /* copy file using sendfile */

    offset = 0;

    rc = sendfile (desc, fd, &offset, stat_buf.st_size);

    if (rc == -1) {

        fprintf(stderr, "error from sendfile: %s\n", strerror(errno));

        exit(1);

    }

    if (rc != stat_buf.st_size) {

        fprintf(stderr, "incomplete transfer from sendfile: %d of %d bytes\n",

            rc,

            (int)stat_buf.st_size);

        exit(1);

    }

    /* close descriptor for file that was sent */

    close(fd);

    /* close socket descriptor */

    close(desc);

}

/* close socket */

close(sock);

return 0;

}

```

Client :

```

/*****

/* Client example requests file data from server */

*****/

```



```

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <sys/sendfile.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <netinet/in.h>

#define SERVER_PORT 1234

main (int argc, char *argv[])
{
    int    rc, sockfd,b;

    char    filename[256];

    char    buffer[32 * 1024];

    struct sockaddr_in    addr;

    struct hostent        *host_ent;

    /******

    /* Initialize the socket address structure    */

    /******

    memset(&addr, 0, sizeof(addr));

    addr.sin_family = AF_INET;

    addr.sin_port    = htons(SERVER_PORT);

    /******

```

```

/* Determine the host name and IP address of the */

/* machine the server is running on */

/*****/

if (argc < 2)
{
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
}
else if (isdigit(*argv[1]))
{
    addr.sin_addr.s_addr = inet_addr(argv[1]);
}
else
{
    host_ent = gethostbyname(argv[1]);

    if (host_ent == NULL)
    {
        printf("Host not found!\n");
        exit(-1);
    }

    memcpy((char *)&addr.sin_addr.s_addr,
           host_ent->h_addr_list[0],
           host_ent->h_length);
}

/*****/

/* Check to see if the user specified a file name */

/* on the command line */

```

```

/*****/

if (argc == 3)
{
    strcpy(filename, argv[2]);
}
else
{
    printf("Enter the name of the file:\n");
    gets(filename);
}

/*****/

/* Create an AF_INET stream socket */

/*****/

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
    perror("socket() failed");
    exit(-1);
}

printf("Socket completed.\n");

/*****/

/* Connect to the server */

/*****/

rc = connect(sockfd,
              (struct sockaddr *)&addr,
              sizeof(struct sockaddr_in));

```

```

if (rc < 0)
{
    perror("connect() failed");

    close(sockfd);

    exit(-1);
}

printf("Connect completed.\n");

/*****

/* Send the request over to the server          */

*****/

rc = send(sockfd, filename, strlen(filename) + 1, 0);

if (rc < 0)
{
    perror("send() failed");

    close(sockfd);

    exit(-1);
}

printf("Request for %s sent\n", filename);

/*****

/* Receive the file from the server              */

*****/

printf("\n%s recieved :\n\n", filename);

do
{
    rc = recv(sockfd, buffer, sizeof(buffer), 0);

    if (rc < 0)

```

```

    {
        perror("recv() failed");
        close(sockfd);
        exit(-1);
    }
    else if (rc == 0)
    {
        printf("End of file\n");
        break;
    }
    printf("%s \n", buffer);
    printf("%d bytes recieved\n",rc);

} while (rc > 0);

/*****/

/* Close the socket */

/*****/

close(sockfd);
}

```

Compile :

```
# gcc fts.c -o fts
```

```
# gcc ftc.c -o ftc
```

```
# ./fts (run in first terminal)
```

```
# ./ftc 127.0.0.1 (run in second terminal)
```

