PRACTICAL FILE

ADVANCED DISTRIBUTED SYSTEMS
(ITMDE01)

Master of Technology
in
Mobile Communication and Network Technology

By

INEESH RAINA
2023PMN4207

Submitted To : Mr. Karan Gupta

Department of Information and Technology

List of Experiments

S.No.	Experiments
1.	Write a Program for creating child process using fork() system call. Print the process ID of child and parent process, Implement the program in UNIX/Linux.
2.	Using the "pipe()" system call, implement the following - 1) Perform inter-process communication between a parent and child process. 2) Perform inter-process communication between two child processes.
3.	Implement. Two-Way inter-process communication using "pipe()" system call 1) Child and Parent process 2) Two Child processes This communication must continue until a specific key is pressed or any process send a Stop message.
4.	Implement Two-Way inter-process communication using FIFOs. Consider Two-Way independent process for communication. This Communication must continue until a specific key is pressed or any process send a Stop message.
5.	Implement Two-Way inter-process communication using Message Queues. Consider Two-Way independent process for communication. This Communication must continue until a specific key is pressed or any process send a Stop message is sent by any one of the processes.
6.	Implement the socket() system call for Two-Way inter-process communication 1. Single Client and Single Server. 2. Multiple Client and Single Server. This Communication must continue until a specific key is pressed or any process send a Stop message is sent by any one of the processes.
7.	Implement Two-Way Inter-process communication using Distributed Shared Memory between independent processes. This Communication must continue until a specific key is pressed or a Stop message is sent by any one of the processes.
8.	Implement RPC for Implementing RPC use 'rpcgen' to create client and server stubs . The remote procedure should return the SUM, DIFFERENCE, MULTIPLE and DIVISION of two numbers to the process that has initiated the RPC.

PRACTICAL - 1

Aim: - Write a program for creating Child process using fork() system call. Print the process ID of child and parent process. Implement the program in UNIX/Linux.

Theory- System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child.

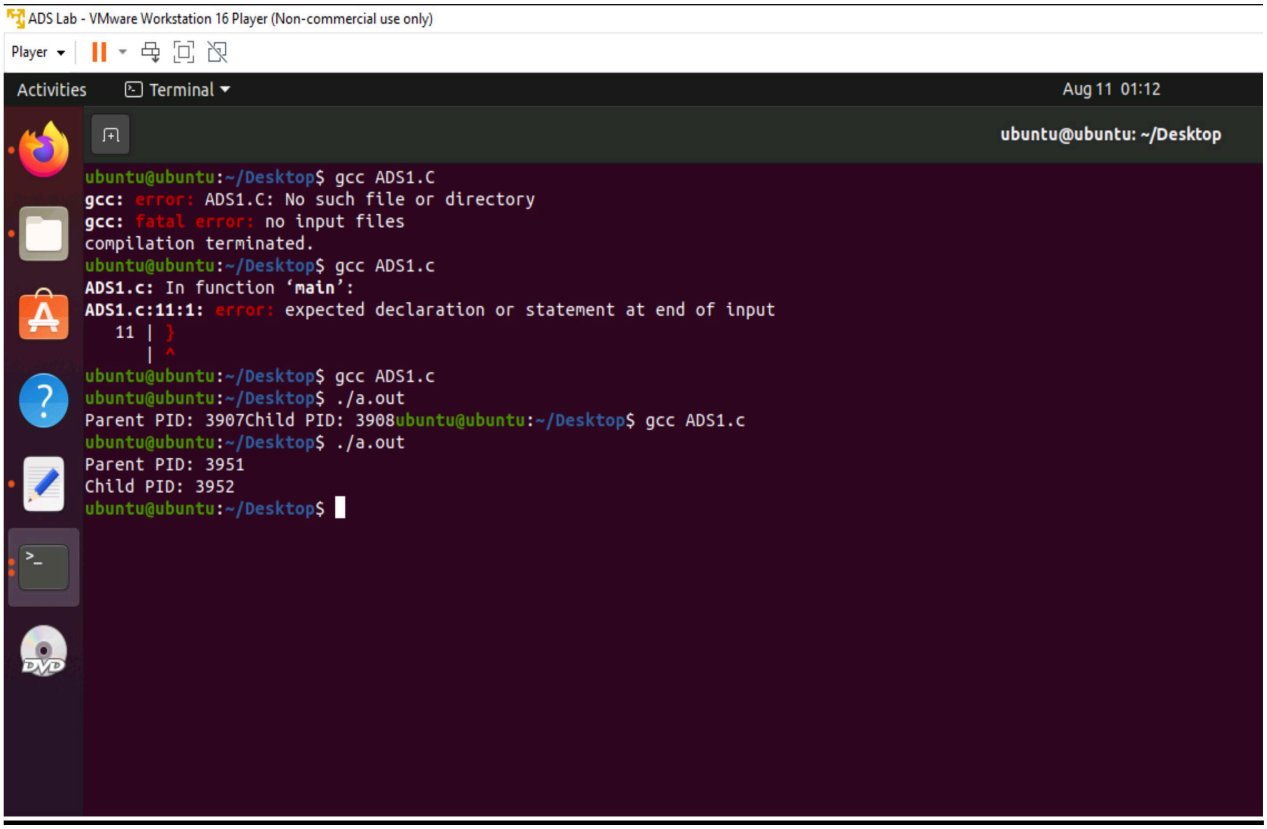
This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero then a new child process is created successfully.
- fork() returns a positive value then, the positive value is the process ID of a child's process to the parent.

Code: C Language

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
 void main(){
        int pid=fork();
        if (pid==0){
                printf("Child PID: %d\n",getpid());
        }
        else if(pid>0){
                printf("Parent PID: %d\n",getpid());
        }
}
```

Output:


ADS Lab - VMware Workstation 16 Player (Non-commercial use only)
Player
Activities
Terminal
Aug 11 01:12
ubuntu@ubuntu: ~/Desktop
ubuntu@ubuntu:~/Desktop$ gcc ADS1.C
gcc: error: ADS1.C: No such file or directory
gcc: fatal error: no input files
compilation terminated.
ubuntu@ubuntu:~/Desktop$ gcc ADS1.c
ADS1.c: In function 'main':
ADS1.c:11:1: error: expected declaration or statement at end of input
11 | }
| ^
ubuntu@ubuntu:~/Desktop$ gcc ADS1.c
ubuntu@ubuntu:~/Desktop$ ./a.out
Parent PID: 3907Child PID: 3908ubuntu@ubuntu:~/Desktop$ gcc ADS1.c
ubuntu@ubuntu:~/Desktop$ ./a.out
Parent PID: 3951
Child PID: 3952
ubuntu@ubuntu:~/Desktop$

PRACTICAL - 2

Aim: Using the "pipe()" system call, implement the following :-

(1) Perform inter-process communication between a Parent and Child process.

(2) Perform inter-process communication between TWO Child processes.

THEORY: Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).

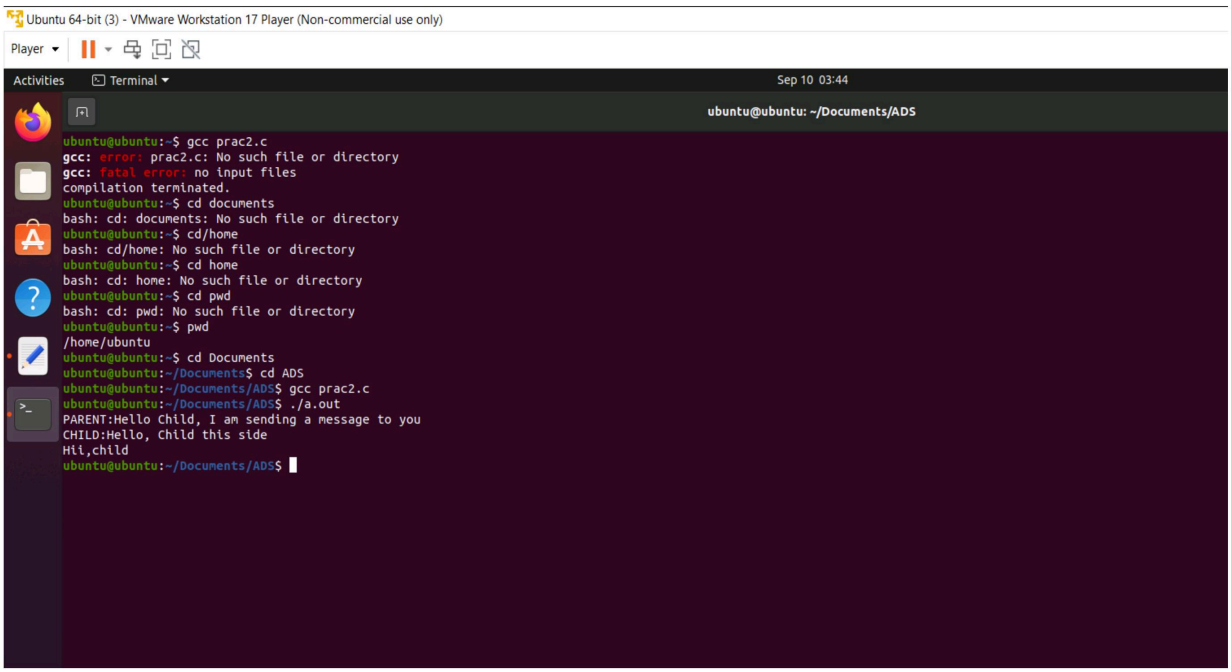
This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

CODE 1:

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

int fd[2],n;

pid_t p;

char b[20];


pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]

p=fork();

if(p>0)
```

```
{
printf("PARENT:Hello Child, I am sending a message to you\n");

write(fd[1],"Hii,child\n",10); //fd[1] is the write end of the pipe

}
else
{
printf("CHILD:Hello, Child this side\n");

n=read(fd[0],b,100); //fd[0] is the read end of the pipe

write(1,b,n);
}
}
```

OUTPUT:

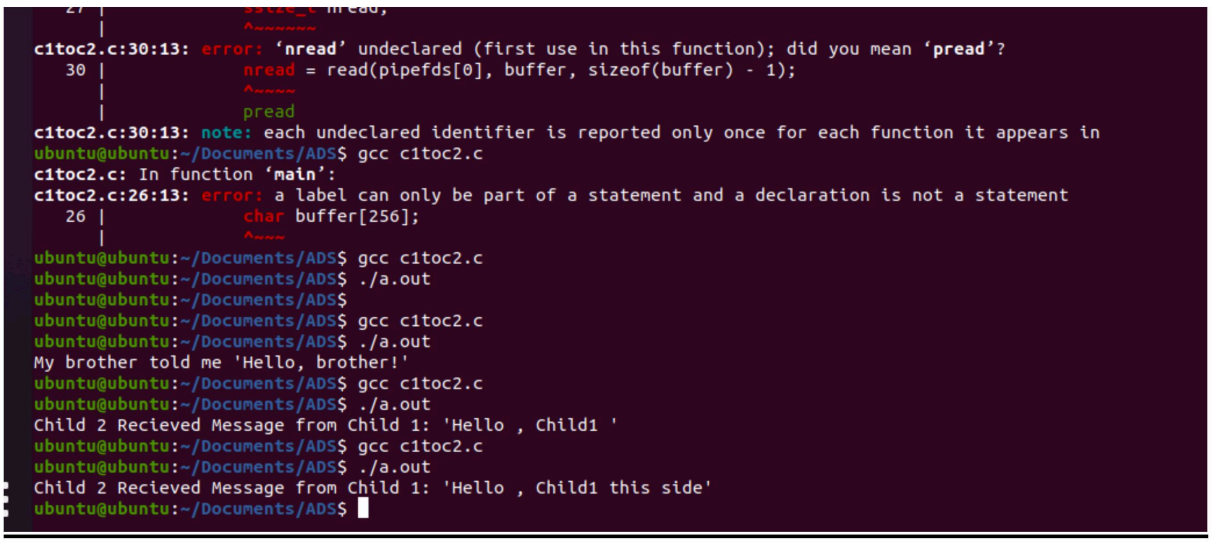


CODE 2:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main() {
    int pipefds[2],n;
    pid_t child1, child2;
    char buffer[256];
    if (pipe(pipefds)==-1){
     printf("Pipe cannot be created");}

    switch (child1 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 1 */
            close(pipefds[0]);

            write(pipefds[1], "Hello , Child1 this side", 30);



    }

    switch (child2 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 2 */
            //char buffer[256];
```

```
        close(pipefds[1]);
        n = read(pipefds[0], buffer, sizeof(buffer) - 1);
        if (n < 0) {
            // handle error
        } else {
            buffer[n] = '\0';
            printf("Child 2 Recieved Message from Child 1: '%s'\n", buffer);
        }
        close(pipefds[0]);


    }


}
```

OUTPUT



Practical -3

AIM: Implement **TWO-WAY** *Inter-process communication* using "pipe()" system call between:
1. Child and parent process
2. Two child processes
This communication must continue until a specific key is pressed or any process sends a STOP message.

Theory: Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication –

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.

2) Create a child process.

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

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

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

6) Perform the communication as required

CODE 1:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

#define BUFFER_SIZE 100

int main() {
    int parent_to_child_pipe[2];
    int child_to_parent_pipe[2];
    char message[BUFFER_SIZE];
    char exit_key;

    if (pipe(parent_to_child_pipe) == -1 || pipe(child_to_parent_pipe) == -1) {
        perror("Pipe creation failed");
        return 1;
    }

    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process

        close(parent_to_child_pipe[1]); // Close parent's write end
        close(child_to_parent_pipe[0]); // Close parent's read end

        while (1) {
            // Receive and display the message from the parent
            ssize_t bytes_read = read(parent_to_child_pipe[0], message, sizeof(message));
            if (bytes_read <= 0) {
                break;
            }
            message[bytes_read] = '\0';

            printf("Child received: %s", message);

            // Send response to the parent
            printf("Child: Enter a message (or 'STOP', 'Q', or 'q' to quit): ");
            fgets(message, sizeof(message), stdin);
            write(child_to_parent_pipe[1], message, strlen(message));

            // Check for the exit key
            if (strcmp(message, "STOP\n") == 0 || strcmp(message, "Q\n") == 0 ||
strcmp(message, "q\n") == 0) {
                break;
            }
        }
```

```
        close(parent_to_child_pipe[0]); // Close parent's read end
        close(child_to_parent_pipe[1]); // Close parent's write end
    } else {
        // Parent process
        close(parent_to_child_pipe[0]); // Close child's read end
        close(child_to_parent_pipe[1]); // Close child's write end

        while (1) {
            printf("Parent: Enter a message (or 'STOP', 'Q', or 'q' to quit): ");
            fgets(message, sizeof(message), stdin);
            write(parent_to_child_pipe[1], message, strlen(message));

            // Check for the exit key
            if (strcmp(message, "STOP\n") == 0 || strcmp(message, "Q\n") == 0 ||
strcmp(message, "q\n") == 0) {
                break;
            }

            // Receive and display the response from the child
            ssize_t bytes_read = read(child_to_parent_pipe[0], message, sizeof(message));
            if (bytes_read <= 0) {
                break;
            }
            message[bytes_read] = '\0';

            printf("Parent received: %s", message);
        }

        close(parent_to_child_pipe[1]); // Close child's write end
        close(child_to_parent_pipe[0]); // Close child's read end

        // Wait for the child process to finish
        wait(NULL);
    }

    return 0;
}
```

Output

```
ineesh@ineesh-To-be-filled-by-O-E-M:~/Desktop$ g++ 5.cpp -o f
ineesh@ineesh-To-be-filled-by-O-E-M:~/Desktop$ ./f
Parent: Enter a message (or 'STOP', 'Q', or 'q' to quit): hello
Child received: hello
Child: Enter a message (or 'STOP', 'Q', or 'q' to quit): bye
Parent received: bye
Parent: Enter a message (or 'STOP', 'Q', or 'q' to quit): good morning
Child received: good morning
Child: Enter a message (or 'STOP', 'Q', or 'q' to quit): 123
Parent received: 123
Parent: Enter a message (or 'STOP', 'Q', or 'q' to quit): q
Child received: q
Child: Enter a message (or 'STOP', 'Q', or 'q' to quit): q
ineesh@ineesh-To-be-filled-by-O-E-M:~/Desktop$
```

CODE 2:

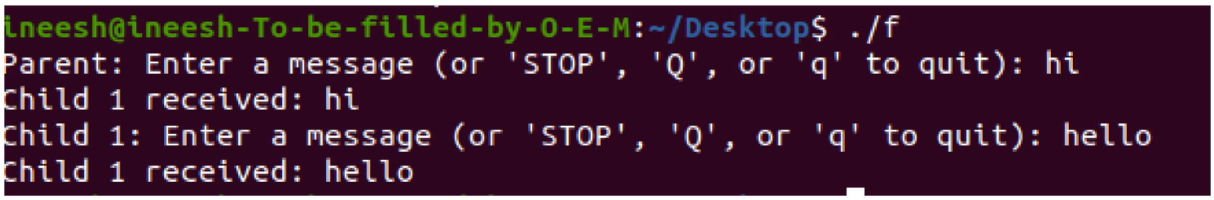
```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>


#define BUFFER_SIZE 100

int main() {
    int parent_to_child_pipe[2];
    int child1_to_parent_pipe[2];
    int child2_to_parent_pipe[2];
    char message[BUFFER_SIZE];
    char exit_key;

    if (pipe(parent_to_child_pipe) == -1 || pipe(child1_to_parent_pipe) == -1 ||
pipe(child2_to_parent_pipe) == -1) {
        perror("Pipe creation failed");
        return 1;
    }

    pid_t child_pid1 = fork();

    if (child_pid1 == -1) {
        perror("Fork failed for child process 1");
        return 1;
    } else if (child_pid1 == 0) {
        // Child process 1
```

```
        close(parent_to_child_pipe[1]); // Close parent's write end
        close(child1_to_parent_pipe[0]); // Close parent's read end
        close(child2_to_parent_pipe[0]); // Close other child's read end
        close(child2_to_parent_pipe[1]); // Close other child's write end

        while (1) {
            // Receive and display the message from the parent
            ssize_t bytes_read = read(parent_to_child_pipe[0], message, sizeof(message));
            if (bytes_read <= 0) {
                break;
            }
            message[bytes_read] = '\0';

            printf("Child 1 received: %s", message);

            // Send response to the parent
            printf("Child 1: Enter a message (or 'STOP', 'Q', or 'q' to quit): ");
            fgets(message, sizeof(message), stdin);
            write(child1_to_parent_pipe[1], message, strlen(message));

            // Check for the exit key
            if (strcmp(message, "STOP\n") == 0 || strcmp(message, "Q\n") == 0 ||
strcmp(message, "q\n") == 0) {
                break;
            }
        }

        close(parent_to_child_pipe[0]); // Close parent's read end
        close(child1_to_parent_pipe[1]); // Close parent's write end
    } else {
        // Parent process
        pid_t child_pid2 = fork();

        if (child_pid2 == -1) {
            perror("Fork failed for child process 2");
            return 1;
        } else if (child_pid2 == 0) {
            // Child process 2

            close(parent_to_child_pipe[1]); // Close parent's write end
            close(child1_to_parent_pipe[0]); // Close other child's read end
            close(child2_to_parent_pipe[0]); // Close parent's read end
            close(child2_to_parent_pipe[1]); // Close parent's write end

            while (1) {
                // Receive and display the message from the parent
                ssize_t bytes_read = read(parent_to_child_pipe[0], message, sizeof(message));
                if (bytes_read <= 0) {
                    break;
                }
                message[bytes_read] = '\0';
```

```
            printf("Child 2 received: %s", message);

            // Send response to the parent
            printf("Child 2: Enter a message (or 'STOP', 'Q', or 'q' to quit): ");
            fgets(message, sizeof(message), stdin);
            write(child2_to_parent_pipe[1], message, strlen(message));

            // Check for the exit key
            if (strcmp(message, "STOP\n") == 0 || strcmp(message, "Q\n") == 0 ||
strcmp(message, "q\n") == 0) {
                break;
            }
        }

        close(parent_to_child_pipe[0]); // Close parent's read end
        close(child2_to_parent_pipe[1]); // Close parent's write end
    } else {
        // Parent process

        close(parent_to_child_pipe[0]); // Close child 1's read end
        close(child1_to_parent_pipe[1]); // Close child 1's write end
        close(child2_to_parent_pipe[1]); // Close child 2's write end

        while (1) {
            printf("Parent: Enter a message (or 'STOP', 'Q', or 'q' to quit): ");
            fgets(message, sizeof(message), stdin);
            write(parent_to_child_pipe[1], message, strlen(message));

            // Check for the exit key
            if (strcmp(message, "STOP\n") == 0 || strcmp(message, "Q\n") == 0 ||
strcmp(message, "q\n") == 0) {
                break;
            }

            // Receive and display the response from child 1
            ssize_t bytes_read1 = read(child1_to_parent_pipe[0], message,
sizeof(message));
            if (bytes_read1 <= 0) {
                break;
            }
            message[bytes_read1] = '\0';

            printf("Child 1 received: %s", message);

            // Receive and display the response from child 2
            ssize_t bytes_read2 = read(child2_to_parent_pipe[0], message,
sizeof(message));
            if (bytes_read2 <= 0) {
                break;
            }
            message[bytes_read2] = '\0';
```

```
            printf("Child 2 received: %s", message);
        }

        close(parent_to_child_pipe[1]); // Close child 1's write end
        close(child1_to_parent_pipe[0]); // Close child 1's read end
        close(child2_to_parent_pipe[0]); // Close child 2's read end

        // Wait for child processes to finish
        wait(NULL);
        wait(NULL);
    }
  }

  return 0;
}
```

Output



Practical -4

AIM: Implement TWO-WAY Inter-process communication using FIFOs.
Consider TWO independent processes for communication.
This communication must continue until a specific key is pressed or any process sends a STOP message.

Theory: A FIFO is a named pipe which stande for First-In-First-Out meaning the data that is written into the pipe first will be read out first always.

A FIFO has to be open at both ends simultaneously. The fifos get listed in the directory tree and any process can access it using its name by providing the approproiate path.

fifo are created using the function mkfifo() which takes as arguments

1. The name of the fifo that has to be created
2. The permissions for the file.

Once the file is created, it needs to be opened using the system call open() and the data can be read and written from the file using read() and write system calls.

One of the examples you can think of using a named pipe is communication between a server and a client. If ther are two fifos one of the server and the other of the client, then the client can send request to the server on the server fifo which the server will read and respond back with the reply on the client's fifo.

Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

#define MESSAGE_SIZE 256

struct msg_buffer {
    long msg_type;
    char msg_text[MESSAGE_SIZE];
};

int main() {
    struct msg_buffer message;
    int msgid;

    // Create a unique message queue key
    key_t key = ftok("/tmp", 'A');

    // Create a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);

    if (msgid == -1) {
        perror("Message queue creation failed");
        return 1;
    }

    signal(SIGINT, SIG_IGN); // Ignore Ctrl+C to keep the communication running

    while (1) {
        printf("Process 1: Enter a message (or 'STOP' to quit): ");
        fgets(message.msg_text, sizeof(message.msg_text), stdin);

        // Send the message to Process 2
        message.msg_type = 1;
        msgsnd(msgid, &message, sizeof(message), 0);

        if (strcmp(message.msg_text, "STOP\n") == 0) {
            break;
        }

        // Receive and display the response from Process 2
        msgrcv(msgid, &message, sizeof(message), 2, 0);
        printf("Process 1 received: %s", message.msg_text);
    }
```

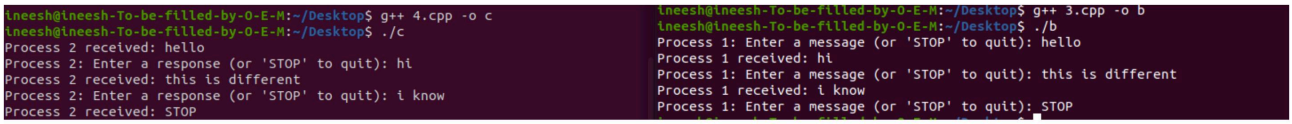
```
    // Remove the message queue when done
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

2nd Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

#define MESSAGE_SIZE 256

struct msg_buffer {
    long msg_type;
    char msg_text[MESSAGE_SIZE];
};

int main() {
    struct msg_buffer message;
    int msgid;

    // Create the same message queue key used by Process 1
    key_t key = ftok("/tmp", 'A');

    // Access the message queue
    msgid = msgget(key, 0666);

    if (msgid == -1) {
        perror("Message queue access failed");
        return 1;
    }

    signal(SIGINT, SIG_IGN); // Ignore Ctrl+C to keep the communication running

    while (1) {
        // Receive and display the message from Process 1
        msgrcv(msgid, &message, sizeof(message), 1, 0);
        printf("Process 2 received: %s", message.msg_text);

        if (strcmp(message.msg_text, "STOP\n") == 0) {
            break;
        }

        printf("Process 2: Enter a response (or 'STOP' to quit): ");
        fgets(message.msg_text, sizeof(message.msg_text), stdin);
```

```
        // Send the response back to Process 1
        message.msg_type = 2;
        msgsnd(msgid, &message, sizeof(message), 0);
    }

    return 0;
}
```

Output



Practical-5

AIM: Implement TWO-WAY Inter-process communication using Message Queues. Consider TWO independent processes for communication. This communication must continue till a specific key is pressed or a STOP message is sent by any one of the processes.

Theory : The message queues can be thought of as a linked list of messages that is used by the communicating processes. The message queues are stored within the kernel. Since there can be different message queues, each is identified by its own id.There can be different types of messages in the message queue, and unlike pipe, the message queue does not strictly follow the FIFO mechanism.

Steps to Perform IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. Below are the following steps to perform communication using message queues.

1. A new queue is created or an existing queue opened by msgget().

2. New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all specified to msgsnd() when the message is added to a queue.

3. Messages are fetched from a queue by msgrcv(). We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field. All processes can exchange information through access to a common system message queue. The sending process places a message onto a queue that another process can read. Each message is given an identification or type so that processes can select the

appropriate message. The process must share a common key to gain access to the queue in the first place.

4. Perform control operations on the message queue msgctl().

Code -

Sender_Code

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

```
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long msg_type;
    char msg[100];
} message;

int main() {
    key_t my_key;
    int msg_id;
 my_key = ftok("progfile", 65); // Used to create a unique key
    msg_id = msgget(my_key, 0666 | IPC_CREAT); // Create or access the message queue
    message.msg_type = 1;

    while (1) {
        // message to be sent to the receiver
        printf("\nSender: Enter a message to send (or 'exit' to quit): ");
        fgets(message.msg, sizeof(message.msg), stdin);

        // Send the message to the receiver
        msgsnd(msg_id, &message, sizeof(message.msg), 0);
```

```
        // Check if the message is "exit" to quit
        if (strcmp(message.msg, "exit\n") == 0) {
            break;
        }
    }

    return 0;
}
```

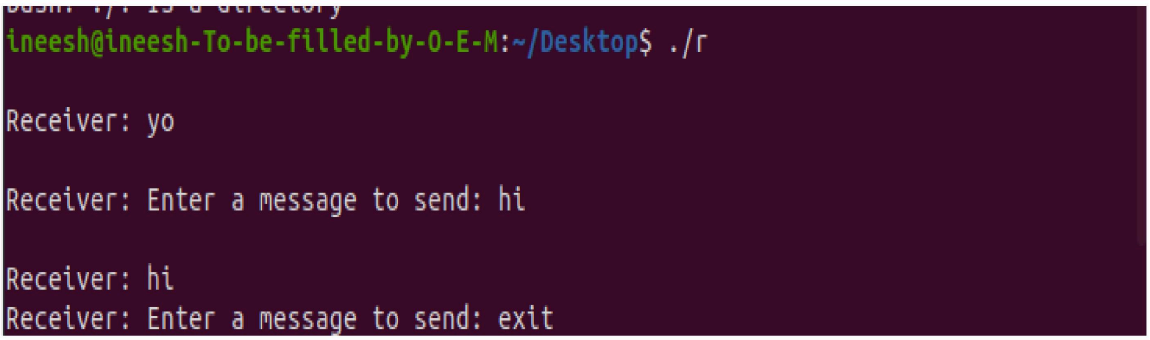
Receiver_Code

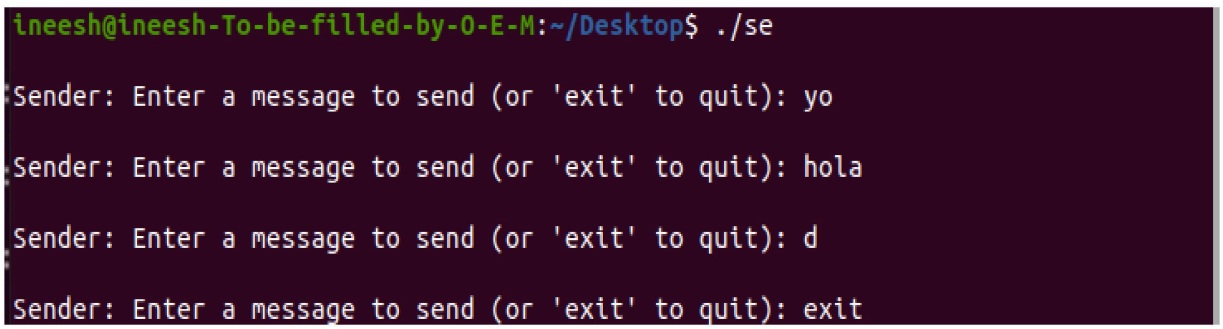
```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long msg_type;
    char msg[100];
} message;

int main() {
    key_t my_key;
    int msg_id;

    my_key = ftok("progfile", 65); // Used to create a unique key
```

```
msg_id = msgget(my_key, 0666 | IPC_CREAT); // Create or access the message queue
message.msg_type = 1;

while (1) {
   // Receive a message from the sender
   msgrcv(msg_id, &message, sizeof(message), 1, 0);
   printf("\nReceiver: %s", message.msg);

   // Check if the received message from the sender is "exit" to quit
   if (strcmp(message.msg, "exit") == 0) {
      break;
   }

   // Send the message back to the sender
   printf("\nReceiver: Enter a message to send: ");
   scanf("%s", message.msg);
   msgsnd(msg_id, &message, sizeof(message), 0);

   // Check if the sent message is "exit" to quit
   if (strcmp(message.msg, "exit") == 0) {
      break;
   }
}
```

```
    return 0;
}
```

Output:





PRACTICAL - 6

Aim : Implement the socket() system call for Two-Way inter-process communication

1) Single Client and Single Server.
2) Multiple Client and Single Server.

This Communication must continue until a specific key is pressed or any process send a Stop message is sent by any one of the processes.

Theory : A socket is a communication endpoint that enables bidirectional data flow between processes either on the same machine or across a network. Sockets facilitate the transfer of data between a client and a server, allowing two-way communication.

IPC sockets (aka Unix domain sockets) enable channel-based communication for processes on the same physical device (host), whereas network sockets enable this kind of IPC for processes that can run on different hosts, thereby bringing networking into play. Network sockets need support from an underlying protocol such as TCP (Transmission Control Protocol) or the lower-level UDP (User Datagram Protocol).

By contrast, IPC sockets rely upon the local system kernel to support communication; in particular, IPC sockets communicate using a local file as a socket address. Despite these implementation differences, the IPC socket and network socket APIs are the same in the essentials. The forthcoming example covers network sockets, but the sample server and client programs can run on the same machine because the server uses network address localhost.

Single Client Single Server : The server and client are running on the same machine. The server listens for incoming connections, and upon a client's connection request, it accepts the connection. Then, both the server and client can send and receive messages through their respective sockets.

Multiple Client and Single Server : Implementing multiple clients and a single server using sockets for Two-Way IPC involves efficiently managing multiple connections. Strategies such as threading, multiprocessing, or asynchronous programming are employed to handle concurrent connections, allowing bidirectional communication between multiple clients and the server concurrently.

Code : Client

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4444

int main(){
        int clientSocket, ret;
        struct sockaddr_in serverAddr;
        char buffer[1024];

        clientSocket = socket(AF_INET, SOCK_STREAM, 0);
        if(clientSocket < 0){
                printf("[-]Error in connection.\n");
                exit(1);
        }
        printf("[+]Client Socket is created.\n");

        memset(&serverAddr, '\0', sizeof(serverAddr));
```

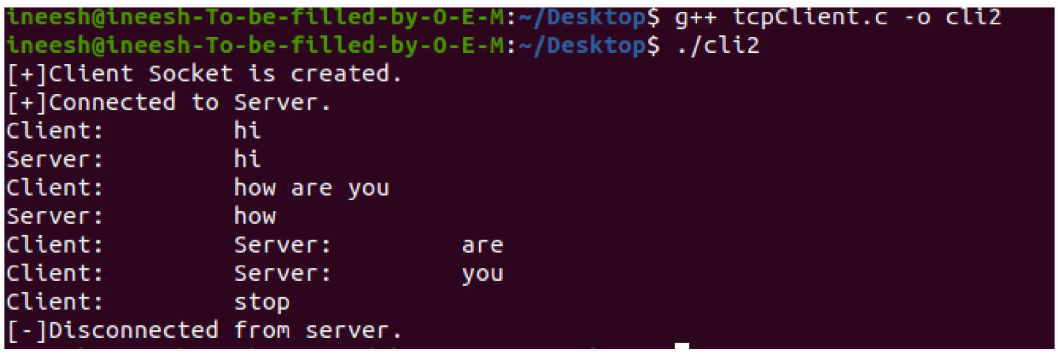
```
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(PORT);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

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){
        printf("Client: \t");
        bzero(buffer, 1024);
        scanf("%s", &buffer[0]);
        send(clientSocket, buffer, strlen(buffer), 0);

        if(strcmp(buffer, ":exit") == 0){
                close(clientSocket);
                printf("[-]Disconnected from server.\n");
                exit(1);
        }
        if(recv(clientSocket, buffer, 1024, 0) < 0){
                printf("[-]Error in receiving data.\n");
```

```
        }else{
                printf("Server: \t%s\n", buffer);
        }
    }
    return 0;
}
```

Output:

Client 1

Client 2



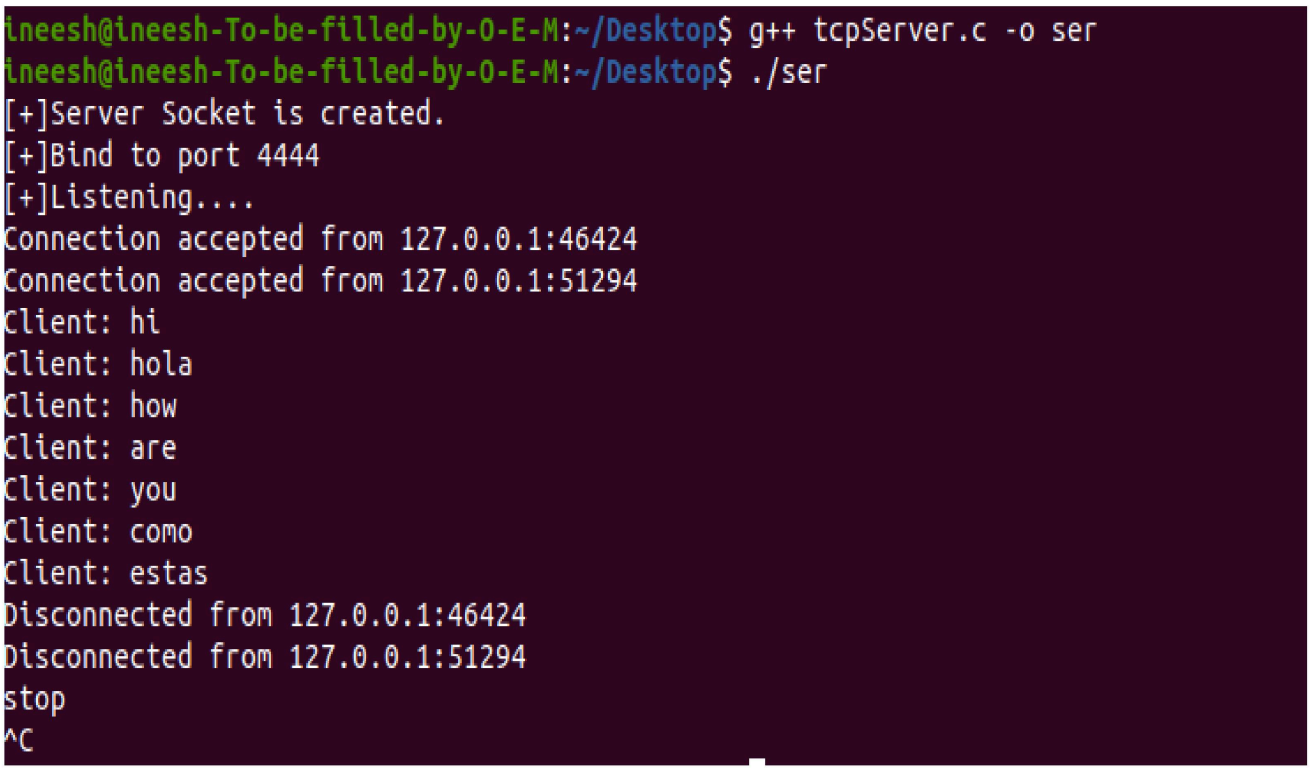
Code : Server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4444
int main(){

        int sockfd, ret;
         struct sockaddr_in serverAddr;
        int newSocket;
        struct sockaddr_in newAddr;
        socklen_t addr_size;
        char buffer[1024];
        pid_t childpid;
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if(sockfd < 0){
                printf("[-]Error in connection.\n");
                exit(1);
        }
```

```
        printf("[+]Server Socket is created.\n");
        memset(&serverAddr, '\0', sizeof(serverAddr));
        serverAddr.sin_family = AF_INET;
        serverAddr.sin_port = htons(PORT);
        serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
        ret = bind(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
        if(ret < 0){
                printf("[-]Error in binding.\n");
                exit(1);
        }
        printf("[+]Bind to port %d\n", 4444);
        if(listen(sockfd, 10) == 0){
                printf("[+]Listening....\n");
        }else{
                printf("[-]Error in binding.\n");
        }
        while(1){
                newSocket = accept(sockfd, (struct sockaddr*)&newAddr, &addr_size);
                if(newSocket < 0){
                        exit(1);
                }
                printf("Connection accepted from %s:%d\n", inet_ntoa(newAddr.sin_addr),
ntohs(newAddr.sin_port));
```

```
                if((childpid = fork()) == 0){
                        close(sockfd);
                        while(1){
                                bzero(buffer, 1024);
                                recv(newSocket, buffer, 1024, 0);
                                if(strcmp(buffer, "stop") == 0){
                                        printf("Disconnected from %s:%d\n",
inet_ntoa(newAddr.sin_addr), ntohs(newAddr.sin_port));
                                        break;
                                }else{
                                        printf("Client: %s\n", buffer);
                                        send(newSocket, buffer, strlen(buffer), 0);
                                        bzero(buffer, sizeof(buffer));
                                }
                        }
                }
        }
        close(newSocket);
        return 0;
}
```

Output :


ineesh@ineesh-To-be-filled-by-O-E-M:~/Desktop$ g++ tcpServer.c -o ser
ineesh@ineesh-To-be-filled-by-O-E-M:~/Desktop$ ./ser
[+]Server Socket is created.
[+]Bind to port 4444
[+]Listening....
Connection accepted from 127.0.0.1:46424
Connection accepted from 127.0.0.1:51294
Client: hi
Client: hola
Client: how
Client: are
Client: you
Client: como
Client: estas
Disconnected from 127.0.0.1:46424
Disconnected from 127.0.0.1:51294
stop
^C

PRACTICAL - 7

Aim: Implement two way interprocess communication using distributed shared memory between independent process.this communication continue until a specific key is pressed or stop msg send by any of the process.

Theory : Implementing two-way interprocess communication using distributed shared memory between independent processes is a complex task, and it typically involves multiple components, such as a shared memory system, synchronization mechanisms, and message passing.

• Initialize Shared Memory: Create a shared memory region that both processes can access. Each process should attach to this shared memory.

• Define Data Structures: Define data structures (e.g., a message queue) within the shared memory to store messages. Make sure to include fields for message content and sender information.

• Synchronization: Implement synchronization mechanisms to ensure that both processes can safely access the shared memory. Techniques like semaphores or mutexes can be used to control access.

• Main Communication Loop: Both processes should enter a main communication loop. In this loop, they continuously check for incoming messages from the shared memory.

• Sending Messages: To send a message, a process writes the message into the shared memory queue and signals the other process that there's a new message to read.

• Receiving Messages: The other process checks the shared memory for incoming messages and processes them accordingly.

• Handling Termination: If either process sends a stop message or a specific key is pressed, both processes should exit their communication loops and clean up the shared memory resources.

process1 takes user input and updates the shared memory accordingly, while process2 reads and processes the shared memory data. The processes run in an infinite loop until a 'stop' message is sent through the shared memory.

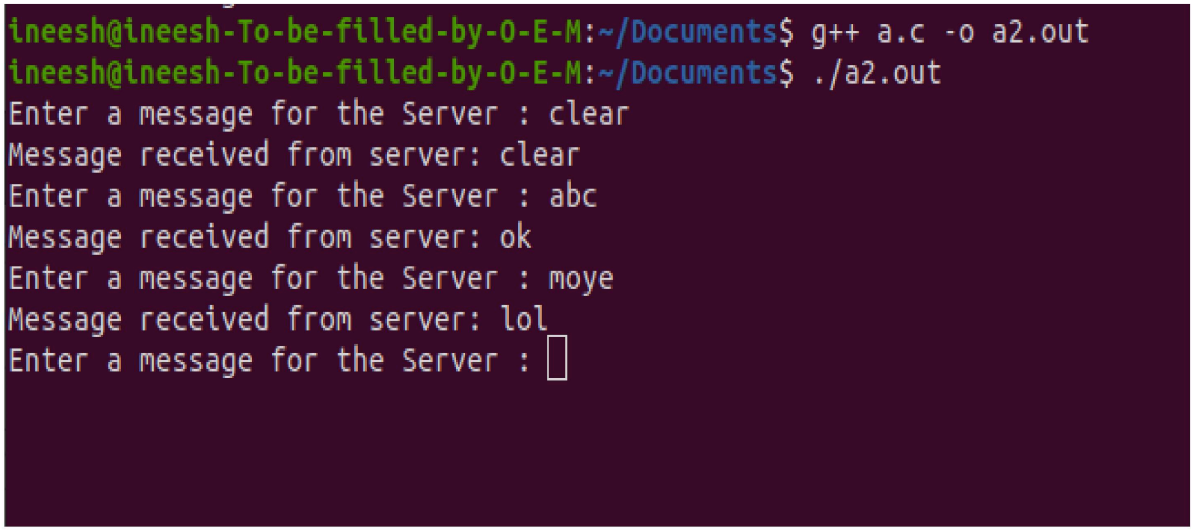
SENDER CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#define SHM_KEY 1024
#define SHM_SIZE 128
int main()
{
   int shmid;
   char
      *shared_memory = NULL;
   shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
   if (shmid < 0)
   {
      printf("Some error occurred !!! \n");
      return 1;
   }
   shared_memory = (char *)shmat(shmid, NULL, 0);
   if ((void *)shared_memory == (void *)-1)
   {
      printf("Some error occurred !!! \n");
```

```
    return 1;
}
while (1)
{
    printf("Enter a message for the Server : ");
    fgets(shared_memory + 1, SHM_SIZE - 1, stdin);
    shared_memory[0] = '1';
    shared_memory[strcspn(shared_memory + 1, "\n") + 1] = '\0';

    if (strcmp(shared_memory + 1, "STOP") == 0)
    {
        printf("Communication Break !!! \n");
        break;
    }
    while (shared_memory[0] == '1')
    {
        sleep(1);
    }
    if (strcmp(shared_memory + 1, "STOP") == 0)
    {
        printf("Communication Break !!! \n");
        break;
    }
```

```
        printf("Message received from server: %s\n", (shared_memory + 1));
    }
    shmdt(shared_memory);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

Output:



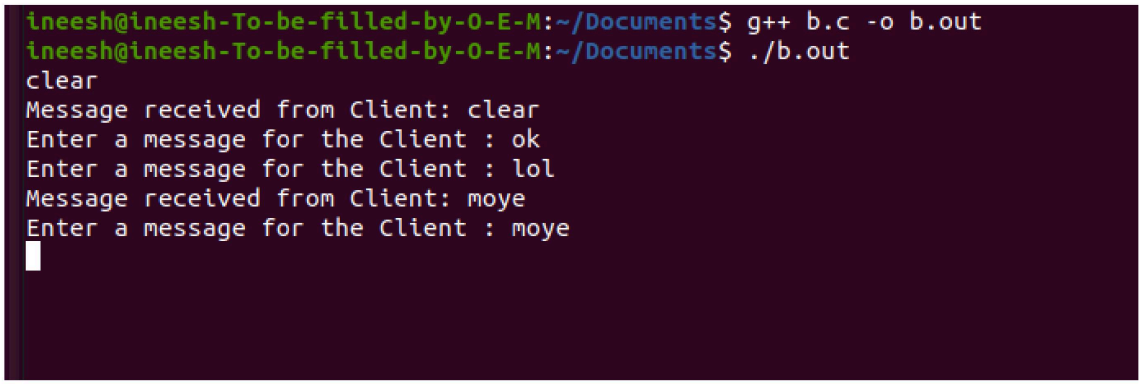
READER CODE:

```
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#define SHM_KEY 1024
#define SHM_SIZE 128
int main()
{
   int shmid;
   char
      *shared_memory = NULL;
   shmid = shmget(SHM_KEY, SHM_SIZE, 0666);
   if (shmid < 0)
   {
      printf("Some error occurred !!! \n");
      return 1;
   }
   shared_memory = (char *)shmat(shmid, NULL, 0);
   if ((void *)shared_memory == (void *)-1)
   {
      printf("Some error occurred !!! \n");
      return 1;
```

```
}
while (1)
{
   while (shared_memory[0] != '1')
   {
      sleep(1);
   }
   printf("Message received from Client: %s\n", (shared_memory + 1));
   printf("Enter a message for the Client : ");
   fgets(shared_memory + 1, SHM_SIZE - 1, stdin);
   shared_memory[0] = '1';
   shared_memory[strcspn(shared_memory + 1, "\n") + 1] = '\0';
   if (strcmp(shared_memory + 1, "STOP") == 0)
   {
      printf("Communication Break !!! \n");
      break;
   }
   else
      shared_memory[0] = '\0';
   while (shared_memory[0] != '1')
   {
      sleep(1);
   }
```

```
        if (strcmp(shared_memory + 1, "STOP") == 0)
        {
            printf("Communication Break !!! \n");
            break;
        }
        printf("Enter a message for the Client : ");
        fgets(shared_memory + 1, SHM_SIZE - 1, stdin);
        shared_memory[0] = '1';
        shared_memory[strcspn(shared_memory + 1, "\n") + 1] = '\0';
        if (strcmp(shared_memory + 1, "STOP") == 0)
        {
            printf("Communication Break !!! \n");
            break;
        }
        // printf("Server sent this message: %s\n", (shared_memory + 1));
        shared_memory[0] = '\0';
    }
    shmdt(shared_memory);
    return 0;
}
```

Output :


ineesh@ineesh-To-be-filled-by-O-E-M:~/Documents$ g++ b.c -o b.out
ineesh@ineesh-To-be-filled-by-O-E-M:~/Documents$ ./b.out
clear
Message received from Client: clear
Enter a message for the Client : ok
Enter a message for the Client : lol
Message received from Client: moye
Enter a message for the Client : moye

PRACTICAL - 8

AIM : Implement RPC. For implementing RPC use "rpcgen" to create client and server stubs. The Remote Procedure should return the SUM, DIFFERENCE, MULTIPLE and DIVISION of two numbers to the process that has initiated the RPC.

THEORY:

1. Remote Procedure Call

A remote procedure call is an inter-process communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call.The sequence of events in a remote procedure call are given as follows –

• The client stub is called by the client.
• The client stub makes a system call to send the message to the server and puts the parameters in the message.
• The message is sent from the client to the server by the client's operating system.
• The message is passed to the server stub by the server operating system.
• The parameters are removed from the message by the server stub.
• Then, the server procedure is called by the server stub.

2. rpcgen

• rpcgen is a tool that generates C code to implement an RPC protocol. The input to rpcgen is a language similar to C known as RPC Language (Remote Procedure Call Language).
• rpcgen is normally used as in the first synopsis where it takes an input file and generates up to four output files.
• If the infile is named proto.x, then rpcgen will generate a header file proto.h, XDR routines in proto_xdr.c, server-side stubs in proto_svc.c, and client-side stubs in proto_clnt.c.

• With the -T option, it will also generate the RPC dispatch table in proto_tbl.i.
• With the -Sc option, it will also generate sample code which would illustrate how to use the remote procedures on the client side. This code would be created in proto_client.c.

• With the -Ss option, it will also generate a sample server code which would illustrate how to write the remote procedures. This code would be created in proto_server.c.

CODE:

1Code:

SERVER

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "add.h"

numbers *
add_1_svc(numbers *argp, struct svc_req *rqstp)
{
        static numbers result1;

        /*
         * insert server code here
         */

         printf("the value given is (%d, %d)", argp -> a, argp -> b);

         result1.a = argp -> a + argp -> b;
```

```
        result1.b = argp -> a * argp -> b;
        result1.c = argp -> a  - argp -> b;
        result1.d = (argp -> b != 0) ? (float)argp -> a  / argp -> b : 0;


        return &result1;
}
```

CLIENT

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "add.h"


void
add_prog_1(char *host, int x, int y)
{
        CLIENT *clnt;
        numbers  *result_1;
        numbers  add_1_arg;
```

```
#ifndef  DEBUG
        clnt = clnt_create (host, ADD_PROG, ADD_VERS, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif   /* DEBUG */
        add_1_arg.a = x;
        add_1_arg.b = y;
        result_1 = add_1(&add_1_arg, clnt);
        if (result_1 == (numbers *) NULL) {
                clnt_perror (clnt, "call failed");
        }
        else{

                printf("\n %d , %d , %d , %d  ",result_1->a, result_1->b, result_1->c, result_1->d);
        }
#ifndef  DEBUG
        clnt_destroy (clnt);
#endif    /* DEBUG */
}


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

        if (argc < 4) {
```

```
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        printf("numbers given are - %d, %d \n", atoi(argv[2]), atoi(argv[3]));
        add_prog_1 (host, atoi(argv[2]), atoi(argv[3]));
exit (0);
}
```

OUTPUT