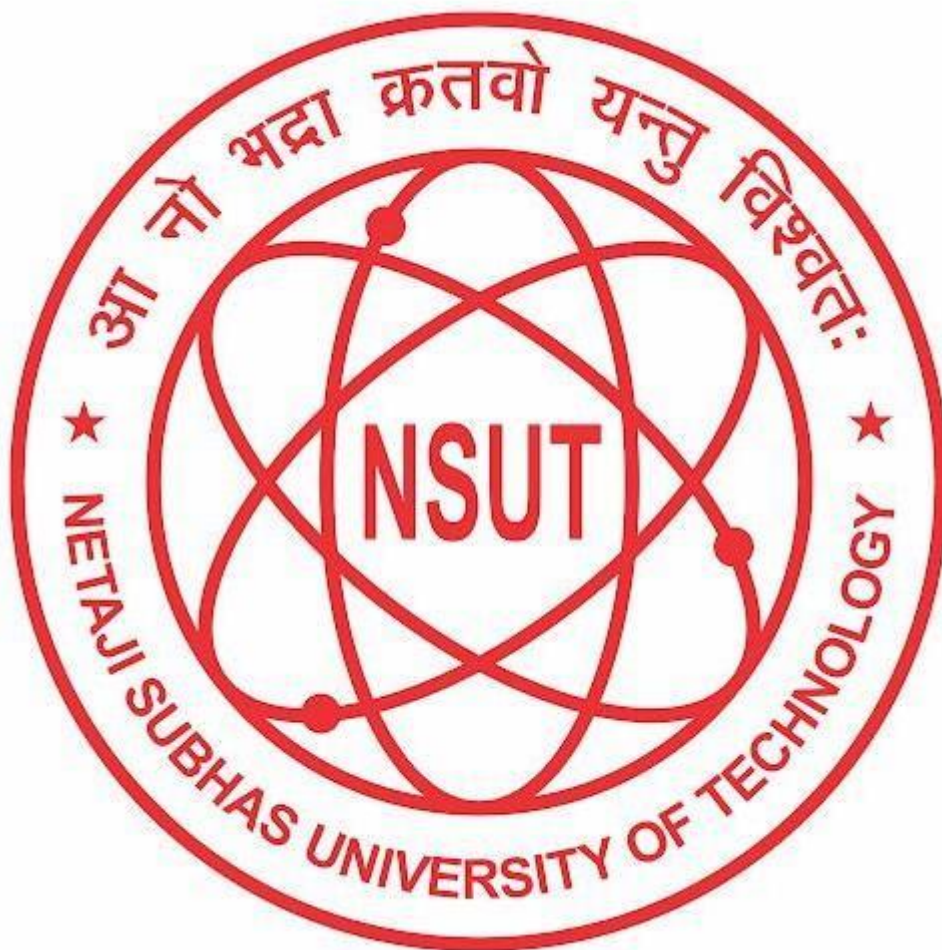


PRACTICAL FILE
Master of Technology
In
ADVANCED DISTRIBUTED SYSTEMS
(ITMDE01)
Mobile Communication and Network Technology
By
Rishabh
2023PMN4213



Submitted To: Mr. Karan Gupta
Department of Information and Technology

PRACTICAL FILE
Master of Technology
In
ADVANCED DISTRIBUTED SYSTEMS
(ITMDE01)
Mobile Communication and Network Technology
By
Aayush
2023PMN4218



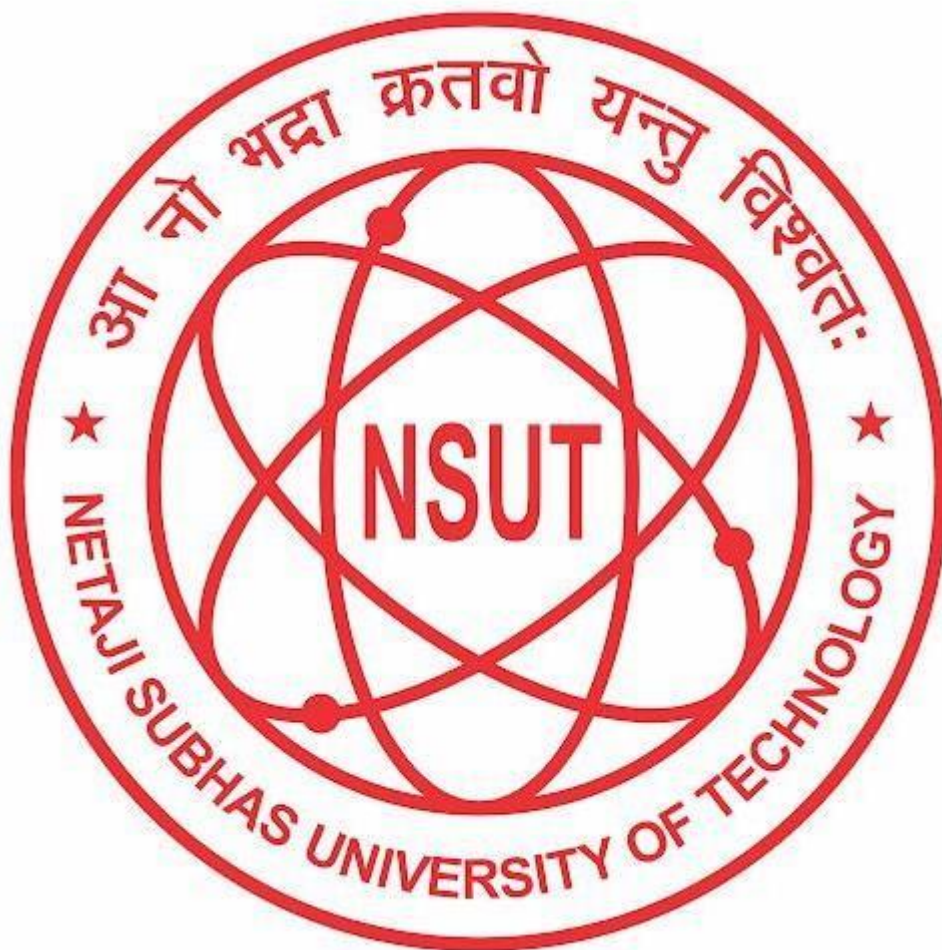
Submitted To: Mr. Karan Gupta
Department of Information and Technology

PRACTICAL FILE
Master of Technology
In
ADVANCED DISTRIBUTED SYSTEMS
(ITMDE01)
Mobile Communication and Network Technology
By
Moaaz Ahmed
2023PMN4227



Submitted To: Mr. Karan Gupta
Department of Information and Technology

PRACTICAL FILE
Master of Technology
In
ADVANCED DISTRIBUTED SYSTEMS
(ITMDE01)
Mobile Communication and Network Technology
By
Srijan Mishra
2023PMN4228



Submitted To: Mr. Karan Gupta
Department of Information and Technology

INDEX

SL NO	TITLE
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 :- <ol style="list-style-type: none">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 between: <ol style="list-style-type: none">1. Child and parent process2. Two child processes <p>This communication must continue until a specific key is pressed or any process sends a STOP message</p>
4	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.
5	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.
6	Implement a Socket() System call for Two-Way Inter-Process Communication between: <ol style="list-style-type: none">1. Single Client and Single Server.2. Multiple Clients and Single Server. <p>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.</p>
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 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.

Experiment– 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 is a program in which the computer requests a service from the kernel.

Fork function

- It is used for creating a new process that is child process.
- It runs concurrently with the process that makes the fork() call (parent process).
- Child and parent both start execution after next statement of the fork call that means they both hold same program counter(pc) value and child process use the resource used by parent process.

Parameters and Return Value

It takes no parameter and returns an integer value.

Negative Value: Creation of child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive: Returned to parent, the value contains process ID of newly created Child.

Code:

```
#include <iostream>
#include <unistd.h>
using namespace std;

int main()

{
int r_value = fork();
if (r_value < 0)
{
perror("fork");
exit(EXIT_FAILURE);
}
else if (r_value > 0)
{
// for Positive value
cout << "parent process ID: " << getpid()
<< endl;
}
else
{
// For Zero value
cout << "child process ID: " << getpid()
<< endl;
}
return 0;
}
```

Output:

```
parent process ID: 16709
child process ID: 16713
[1] + Done          "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-hc05dpvo.asq" 1>"/tmp/Microsoft-MIEngine-Out-2x4havbu.ir0"
moaaz@DESKTOP-EPOK034:~$
```

Explanation: The program includes the necessary headers: `<iostream>` for input and output, and

`<unistd.h>` for the `fork()` and related system calls.

Inside the `main()` function, a variable `r_value` is declared that will store the result of the `fork()` call, indicating whether the process is the parent or child.

Possible scenarios:

- If `fork()` returns -1, an error occurred during the process creation, and the program displays an error message using `perror()`.
- If `fork()` returns a positive value (greater than 0), the process is in the parent context. The parent process outputs a message indicating that it's the parent and displays its process ID using `getpid()`.
- If `fork()` returns 0, the process is in the child context. The child process outputs a message indicating that it's the child and displays its process ID using `getpid()`.

The program uses an `else if` block to handle the parent context and an `else` block to handle the child context. Both the parent and child processes then exit the `if-else` blocks and return 0 to terminate the program.

Experiment- 2

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

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

The objective of the provided C code is to illustrate inter-process communication (IPC) using pipes in a parent-child process scenario.

The program creates a pipe, forks a child process, and demonstrates the communication of data (a message) from the parent process to the child process through the pipe.

Here are the key steps and objectives of the code:

1. Pipe Creation: The code creates a pipe using the `pipe()` function.

This pipe establishes a unidirectional communication channel between the parent and child processes.

2. Forking: The program forks into two processes - the parent process and the child process. The purpose of forking is to create separate execution contexts for each process.

3. Parent Process:

- The parent process closes the read end of the pipe since it will be writing to the pipe.

- It prepares a message, in this case, "Hello from the parent process!", and writes it to the pipe using the `write()` function.
- After writing, it closes the write end of the pipe to signal that it has finished writing.

4. Child Process:

- The child process closes the write end of the pipe since it will be reading from the pipe.

- It reads data from the pipe using the `read()` function and stores it in a buffer.

- The child process then prints the received message.

- After reading, it closes the read end of the pipe to signal that it has finished reading.

5. Output and Demonstration: The program includes print statements to show the progress of each process. This helps to visualize the flow of execution and the interaction between the parent and child processes.

Overall, the code serves as a basic example of how pipes can be used for inter-process communication, allowing one process to send data to another process in a synchronized manner.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main() { int fd[2];
if (pipe(fd) == -1) {
perror("Pipe creation failed");
return 1;
}
pid_t pid = fork();
if (pid > 0) {
close(fd[0]);
```

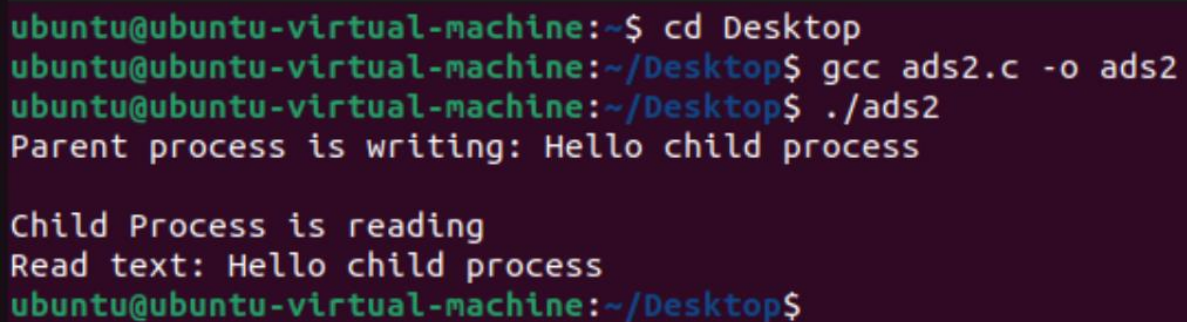


```

char *text = "Hello child process\n";
printf("Parent process is writing: %s", text);
write(fd[1], text, strlen(text) + 1);
close(fd[1]);
} else if (pid == 0) {
close(fd[1]);
char readBuffer[100];
printf("\nChild Process is reading\n");
read(fd[0], readBuffer, sizeof(readBuffer));
printf("Read text: %s", readBuffer);
close(fd[0]);
} else {
perror("Fork failed");
return 1;
}
return 0;}

```

Output



```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc ads2.c -o ads2
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./ads2
Parent process is writing: Hello child process

Child Process is reading
Read text: Hello child process
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

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

Code2

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main() {
int pipe_fd[2];
if (pipe(pipe_fd) == -1) {
perror("pipe");
exit(EXIT_FAILURE);
}
pid_t child1_pid = fork();
if (child1_pid == -1)
{
perror("fork");
exit(EXIT_FAILURE);
}

```

```

if (child1_pid == 0)
{
close(pipe_fd[0]); char message[] = "Hello from Child 1";
printf("Child 1 sent: %s\n", message);
write(pipe_fd[1], message, strlen(message) + 1);
close(pipe_fd[1]);
exit(EXIT_SUCCESS);
}
pid_t child2_pid = fork();
if (child2_pid == -1) {
perror("fork");
exit(EXIT_FAILURE);
}
if (child2_pid == 0) {
close(pipe_fd[1]);
char message[50];
read(pipe_fd[0], message, sizeof(message));
printf("Child 2 received: %s\n", message);
close(pipe_fd[0]);
exit(EXIT_SUCCESS);
}
close(pipe_fd[0]);
close(pipe_fd[1]);
wait(NULL);
wait(NULL);
return 0; }

```

Output

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc ads2.c -o ads2
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./ads2
Child 1 sent: Hello from Child 1
Child 2 received: Hello from Child 1
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

Experiment-3

Aim: Implement **TWO-WAY Inter-process communication** using "pipe()" system call between:

1. Child and parent process
2. Two child processes

Theory:

Child and Parent process: 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 –

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. *Code 1 (Communication Between parent and child)*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#define MAX_MESSAGE_LENGTH 100
int main() {
    int parent_to_child[2];
    int child_to_parent[2];
    if (pipe(parent_to_child) == -1 || pipe(child_to_parent) == -1) {
        perror("Pipe creation failed");
        return 1;
    }
    pid_t pid = fork();
    if (pid > 0) {
        // Parent process
        close(parent_to_child[0]);
        close(child_to_parent[1]);
        char parentMessage[MAX_MESSAGE_LENGTH];
        while (1) { printf("Parent sends a message to child: ");
            fgets(parentMessage, sizeof(parentMessage), stdin);
            parentMessage[strcspn(parentMessage, "\n")] = '\0';
            write(parent_to_child[1], parentMessage, strlen(parentMessage) + 1);
            if (strcmp(parentMessage, "STOP") == 0) {
                break;
            }
        }
```

```

}
char childBuffer[MAX_MESSAGE_LENGTH];
read(child_to_parent[0], childBuffer, sizeof(childBuffer));
printf("Parent received message from child: %s\n", childBuffer);
}
close(parent_to_child[1]);
close(child_to_parent[0]);
}
else if (pid == 0) {
close(parent_to_child[1]);
close(child_to_parent[0]);
char childBuffer[MAX_MESSAGE_LENGTH];
while (1) {
read(parent_to_child[0], childBuffer, sizeof(childBuffer));
printf("Child received from parent: %s\n", childBuffer);
if (strcmp(childBuffer, "STOP") == 0) {
break; }
char childMessage[MAX_MESSAGE_LENGTH];
printf("Child sends a message to parent: ");
fgets(childMessage, sizeof(childMessage), stdin);
childMessage[strcspn(childMessage, "\n")] = '\0';
write(child_to_parent[1], childMessage, strlen(childMessage) + 1);
}
close(parent_to_child[0]);
close(child_to_parent[1]);
} else {
perror("Fork failed");
return 1;
}
return 0;
}

```

Output:

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc ads33.c -o ads33
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./ads33
Parent sends a message to child: hello child
Child received from parent: hello child
Child sends a message to parent: hello parent
Parent received message from child: hello parent
Parent sends a message to child: how are you
Child received from parent: how are you
Child sends a message to parent: I am Fine
Parent received message from child: I am Fine
Parent sends a message to child: STOP
Child received from parent: STOP
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

2.Communication between Child 1 and Child2

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main() {
    int child1_to_child2[2]; // Pipe for Child 1 to Child 2 communication
    int child2_to_child1[2]; // Pipe for Child 2 to Child 1 communication
    if (pipe(child1_to_child2) == -1 || pipe(child2_to_child1) == -1) {
        perror("Pipe creation failed");
        return 1;
    }
    pid_t child1_pid = fork();
    if (child1_pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (child1_pid == 0) {
        // Child 1 process
        close(child1_to_child2[0]); // Close Child 1's read end
        close(child2_to_child1[1]); // Close Child 1's write end
        char child1Message[100];
        while (1) {
            printf("Child 1: Send a message to Child 2 : ");
            fgets(child1Message, sizeof(child1Message), stdin);
            write(child1_to_child2[1], child1Message, strlen(child1Message) + 1);
            if (strcmp(child1Message, "STOP\n") == 0) {
                break;
            }
            char child2Buffer[100];
            read(child2_to_child1[0], child2Buffer, sizeof(child2Buffer));
            printf("Child 1 received from Child 2: %s", child2Buffer);
        }
        close(child1_to_child2[1]);
        close(child2_to_child1[0]);
        exit(0);
    } else {
        pid_t child2_pid = fork();
        if (child2_pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (child2_pid == 0) {
            close(child1_to_child2[1]);
            close(child2_to_child1[0]);
            char child2Message[100];
            while (1) {
                read(child1_to_child2[0], child2Message, sizeof(child2Message));
                printf("Child 2 received from Child 1: %s", child2Message);
```

```

if (strcmp(child2Message, "STOP\n") == 0) {
break;
}
char child1Response[100];
printf("Child 2: Send a message to Child 1: ");
fgets(child1Response, sizeof(child1Response), stdin);
write(child2_to_child1[1], child1Response, strlen(child1Response) + 1);
}
close(child1_to_child2[0]);
close(child2_to_child1[1]);
exit(0);
} else {
close(child1_to_child2[0]);
close(child2_to_child1[1]);
wait(NULL);
wait(NULL);
close(child1_to_child2[1]);
close(child2_to_child1[0]);
}
}
return 0;
}

```

Output:

```

ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc ads32.c -o ads32
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./ads32
Child 1: Send a message to Child 2 : hello child 2
Child 2 received from Child 1: hello child 2
Child 2: Send a message to Child 1: hello child 1
Child 1 received from Child 2: hello child 1
Child 1: Send a message to Child 2 : Today morning heavy rainfall happen
Child 2 received from Child 1: Today morning heavy rainfall happen
Child 2: Send a message to Child 1: ohh really are you ok
Child 1 received from Child 2: ohh really are you ok
Child 1: Send a message to Child 2 : yeh it too 15 more minute to rech college
Child 2 received from Child 1: yeh it too 15 more minute to rech college
Child 2: Send a message to Child 1: good
Child 1 received from Child 2: good
Child 1: Send a message to Child 2 : STOP
Child 2 received from Child 1: STOP
ubuntu@ubuntu-virtual-machine:~/Desktop$

```


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:

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
 - A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
 - Usually, a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
 - A FIFO special file is entered into the filesystem by calling `mkfifo()` in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.
- Two-way communication using FIFOs allows for real-time bidirectional data exchange between independent processes. It's commonly used in scenarios where multiple processes need to cooperate and share data while running independently. However, it requires careful synchronization and error handling to ensure reliable communication between the processes.

PROCESS-A:

```
#include <iostream>
#include <cstring>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

using namespace std;

int main()
{
    int fileDesc;//file descriptor to uniquely identify an open file
```

```

const char *myfifo = "fifo";

// Create the FIFO (named pipe)
mkfifo(myfifo, 0666);

string msgAB;
char receivedMsg[1024];

const string stop_message = "exit";
bool exitFlag = false;
while (!exitFlag)
{
// Open the FIFO
fileDesc = open(myfifo, O_WRONLY);
if (fileDesc == -1)
{
perror("Error opening FIFO for writing");
return 1;
}

cout << "Enter a message (or type 'exit' to quit): ";
getline(cin, msgAB);

if (msgAB == stop_message)
{
exitFlag = true;
}
else
{
write(fileDesc, msgAB.c_str(), msgAB.length() + 1);
close(fileDesc);

fileDesc = open(myfifo, O_RDONLY);
if (fileDesc == -1)
{
perror("Error opening FIFO for reading");
return 1;
}
ssize_t bytesRead = read(fileDesc, receivedMsg, sizeof(receivedMsg));
if (bytesRead == -1)
{
perror("Error reading from FIFO");
close(fileDesc); // Close the file descriptor before exiting
return 1;
}
cout << "ProcessB: " << receivedMsg << endl;
close(fileDesc);
}
}

```

```
return 0;
}
```

PROCESS-B

```
#include <iostream>
#include <cstring>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
```

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

```
using namespace std;
```

```
int main()
{
    int fileDesc;
    const char *myfifo = "fifo";
```

```
// Create the FIFO (named pipe)
mkfifo(myfifo, 0666);
```

```
string msgBA;
char receivedMsg[1024];
```

```
const string stop_message = "exit";
bool exitFlag = false;
```

```
while (!exitFlag)
{
    // Open the FIFO
    fileDesc = open(myfifo, O_WRONLY);
    if (fileDesc == -1)
    {
        perror("Error opening FIFO for writing");
        return 1;
    }
}
```

```
cout << "Enter a message (or type 'exit' to quit): ";
getline(cin, msgBA);
```

```
if (msgBA == stop_message)
{
    exitFlag = true;
}
else
{
```

```
    write(fileDesc, msgBA.c_str(), msgBA.length() + 1);
    close(fileDesc);
```

```

fileDesc = open(myfifo, O_RDONLY);
if (fileDesc == -1)
{
perror("Error opening FIFO for reading");
return 1;
}

ssize_t bytesRead = read(fileDesc, receivedMsg, sizeof(receivedMsg));
if (bytesRead == -1)
{
perror("Error reading from FIFO");
close(fileDesc); // Close the file descriptor before exiting
return 1;
}
cout << "ProcessA: " << receivedMsg << endl;
close(fileDesc);
}
}

return 0;
}

```

Output:

```

moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-4/final$ g++ processA.cpp
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-4/final$ ./a.out
Enter a message (or type 'exit' to quit): hello processA
ProcessB: hi processB
Enter a message (or type 'exit' to quit): how r u going
ProcessB: all good yet
Enter a message (or type 'exit' to quit): hmm,what r u doing processB
ProcessB: nothing much trying to build communication
Enter a message (or type 'exit' to quit): ohh,what type of?
ProcessB: between two processes
Enter a message (or type 'exit' to quit): haha,what we are doing
ProcessB: haha
Enter a message (or type 'exit' to quit): .

```

```

moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-4/final$ g++ processB.cpp
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-4/final$ ./a.out
Enter a message (or type 'exit' to quit): hey
ProcessA: hello processA
Enter a message (or type 'exit' to quit): hi processB
ProcessA: how r u going
Enter a message (or type 'exit' to quit): all good yet
ProcessA: hmm,what r u doing processB
Enter a message (or type 'exit' to quit): nothing much trying to build communication
ProcessA: ohh,what type of?
Enter a message (or type 'exit' to quit): between two processes
ProcessA: haha,what we are doing
Enter a message (or type 'exit' to quit): haha
.

```

Experiment-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:

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- A new queue is created, or an existing queue opened by msgget().
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 of which are specified to msgsnd() when the message is added to a queue.
- Messages are fetched from a queue by msgrcv().
- All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key to gain access to the queue in the first place.
- ftok(): is used to generate a unique key.
- msgget(): either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- msgsnd(): Data is placed on to a message queue by calling msgsnd().
- msgrcv(): messages are retrieved from a queue.
- msgctl(): It performs various operations on a queue. Generally, it is use to destroy message queue.

PROCESS-A:

```
#include <iostream>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string>
#include <cstring>
#include <cstdlib>
#include <csignal>
using namespace std;
```

```
const key_t msgqueueKey = 64; // Define a key for the message queue.
```

```
int mqueue = -1; // Initialize the message queue descriptor.
```

```
struct Message
{
```

```

long mtype; // Message type (used for filtering messages).
char mtext[1024]; // Message content.
};

int main()

{
// Attempt to create or access an existing message queue with the
specified key.
mqueue = msgget(msgqueueKey, 0666 | IPC_CREAT);
if (mqueue == -1)
{
cerr << "Message queue creation failed." << endl;
exit(1); // Exit the program with an error code.
}

while (true)
{
try
{
Message msg;

// Receive a message from the message queue with type 1 (any
type).
msgrcv(mqueue, &msg, sizeof(msg.mtext), 1, 0);
string receivedMessage = msg.mtext;

if (receivedMessage == "STOP")
{
cout << "Aborting connection." << endl;
msgctl(mqueue, IPC_RMID, nullptr); // Remove the message
queue.

exit(0); // Exit the program
successfully.
}

cout << "Message received from processB: " << receivedMessage <<
endl;

string response;
cout << "Enter a message: ";
getline(cin, response);

msg.mtype = 1;
strncpy(msg.mtext, response.c_str(), sizeof(msg.mtext));
msgsnd(mqueue, &msg, sizeof(msg.mtext), 0); // Send a response
message to processB.
}
catch (int err)

```



```

{
cerr << "Some error occurred: " << err << endl;
msgctl(mqueue, IPC_RMID, nullptr); // Remove the message queue on
error.
exit(1); // Exit the program with an
error code.
}
}

return 0;

}

```

PROCESS-B:

```

#include <iostream>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string>
#include <cstring>
#include <cstdlib>
#include <csignal>
using namespace std;

const key_t msgqueueKey = 64; // Define the same key for the message queue as
processA.

int mqueue = -1; // Initialize the message queue descriptor.

struct Message
{
long mtype; // Message type (used for filtering messages).
char mtext[1024]; // Message content.
};

int main()
{
while (mqueue == -1)
{
try
{
// Try to access an existing message queue with the specified
key.
mqueue = msgget(msgqueueKey, 0666);
if (mqueue == -1)
{
cout << "Queue does not exist, wait..." << endl;
sleep(3); // Sleep for 3 seconds and then retry.
}
}
}
}

```

```

catch (int err)
{
cerr << "An error occurred: " << err << endl;
exit(1); // Exit the program with an error code.
}
}

while (true)
{
try
{

string message;
cout << "Enter a message: ";
getline(cin, message);
Message msg;
msg.mtype = 1;
strncpy(msg.mtext, message.c_str(), sizeof(msg.mtext) - 1);
msgsnd(mqueue, &msg, sizeof(msg.mtext), 0); // Send a message to
processA.

sleep(2); // Sleep for 2 seconds.
// Receive a response message from processA with type 1 (any
type).
msgrcv(mqueue, &msg, sizeof(msg.mtext), 1, 0);
string receivedMessage = msg.mtext;
if (receivedMessage == "STOP")
{
cout << "Aborting the connection" << endl;
msgctl(mqueue, IPC_RMID, nullptr); // Remove the message
queue.
exit(0); // Exit the program
successfully.
}
cout << "Message received by processA: " << receivedMessage <<
endl;
}
catch (int err)
{
cerr << "Some error occurred: " << err << endl;
msgctl(mqueue, IPC_RMID, nullptr); // Remove the message queue on
error.
exit(1); // Exit the program with an
error code.
}
}
return 0;
}

```

Output:

```

moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS$ cd assignment-5
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$ g++ processA.cpp
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$ ./a.out
Message received from processB: hello processA...
Enter a message: hi processB, Any help required from my side
Message received from processB: actually yes
Enter a message: yah! sure plz tell me...
Message received from processB: I need the current value of variable x and y.
Enter a message: x is 13 and y is 7.
Message received from processB: okay,thanks for the info
Enter a message: no problem, can i stop the connection now IF Nothing More Is required From my side
Message received from processB: sure,we can close connection
Enter a message: hmm ok
Aborting connection.
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$

```

```

moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS$ cd assignment-5
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$ g++ processB.cpp
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$ ./a.out
Enter a message: hello processA...
Message received by processA: hi processB, Any help required from my side
Enter a message: actually yes
Message received by processA: yah! sure plz tell me...
Enter a message: I need the current value of variable x and y.
Message received by processA: x is 13 and y is 7.
Enter a message: okay,thanks for the info
Message received by processA: no problem, can i stop the connection now IF Nothing More Is required From my side
Enter a message: sure,we can close connection
Message received by processA: hmm ok
Enter a message: STOP
Aborting the connection
moaaz@DESKTOP-EPOK034:~/NSUT assignment/ADS/assignment-5$

```

EXPERIMENT-6

AIM: Implement a Socket() System call for Two-Way Inter-Process Communication between:

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

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: A socket is one endpoint of a two way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place. Like 'Pipe' is used to create pipes and sockets is created using 'socket' system call. The socket provides bidirectional FIFO Communication facility over the network. A socket connecting to the network is created at each end of the communication. Each socket has a specific address. This address is composed of an IP address and a port number. Socket are generally employed in client server applications. The server creates a socket, attaches it to a network port addresses then waits for the client to contact it. The client creates a socket and then attempts to connect to the server socket. When the connection is established, transfer of data takes place.

CODE

1. Program for Single Client and Single Server a. Client1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> #include <string.h>
#include <arpa/inet.h>
#define PORT 12345
int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1) {
        printf("Error creating socket\n");
        return EXIT_FAILURE;
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    if (connect(client_socket, (struct sockaddr*)&server_addr, sizeof(server_addr))
        == -1)
    {
        printf("Error connecting to server\n");
        return EXIT_FAILURE;
    }
    printf("Connected to server\n");
    char buffer[1024];
    while (1) {
        printf("Client to server ");
        fgets(buffer, sizeof(buffer), stdin);
        send(client_socket, buffer, strlen(buffer), 0);
        if (buffer[0] == 'q' || strcmp(buffer, "STOP", 4) == 0) { printf("Communication stopped.\n");
            break;
        }
    }
}
```

```

}
recv(client_socket, buffer, sizeof(buffer), 0);
printf("Server: %s\n", buffer);
if (strncmp(buffer, "STOP", 4) == 0) {
printf("Communication stopped.\n");
break;
}
}
close(client_socket);
return 0;
}

```

b. Server1.c

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#define PORT 12345
int main() {
int server_socket, client_socket;
struct sockaddr_in server_addr, client_addr;
socklen_t addr_size;
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1) { printf("Error creating socket\n");
return EXIT_FAILURE;
}
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = INADDR_ANY;
if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) ==
-1) {
printf("Error binding socket\n");
return EXIT_FAILURE;
}
if (listen(server_socket, 10) == -1) {
printf("Error listening\n");
return EXIT_FAILURE;
}
printf("Server listening on port %d...\n", PORT);
addr_size = sizeof(client_addr);
client_socket = accept(server_socket, (struct sockaddr*)&client_addr,
&addr_size);
if (client_socket == -1) {
printf("Error accepting connection\n");
return EXIT_FAILURE;
}
printf("Connection accepted from %s:%d\n", inet_ntoa(client_addr.sin_addr),
ntohs(client_addr.sin_port));
char buffer[1024];
while (1) { recv(client_socket, buffer, sizeof(buffer), 0);

```

```

printf("Client: %s\n", buffer);
if (buffer[0] == 'q' || strcmp(buffer, "STOP", 4) == 0) {
printf("Communication stopped.\n");
break;
}
printf("Server to client: ");
fgets(buffer, sizeof(buffer), stdin);
send(client_socket, buffer, strlen(buffer), 0);
if (strcmp(buffer, "STOP", 4) == 0) {
printf("Communication stopped.\n");
break;
}
}
close(client_socket);
close(server_socket);
return 0;
}

```

Output

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc client1.c
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./client1
Connected to the server
Client to server: hello friend
Server: hello
friend

Client to server: how are you
Server: I am fine
u

Client to server: ok
Server: STOP
fine
u

Communication stopped.
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc server1.c
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./server
bash: ./server: No such file or directory
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./server1
Server is listening on port 54321...
Connected to a client from 127.0.0.1:48802
Received from client: hello friend

Server to client: hello
Received from client: how are you

Server to client: I am fine
Received from client: ok
m fine

Server to client: STOP
Communication stopped.
ubuntu@ubuntu-virtual-machine:~/Desktop$

```


2. Program for Multiple Clients and Single Server *a. Server2.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>
#define PORT 8080
#define MAX_CLIENTS 5
struct ClientData {
    int socket;
    struct sockaddr_in address;
    int stop_flag;
};
void *handle_client(void *arg) {
    struct ClientData *client_data = (struct ClientData *)arg;
    char buffer[1024];
    while (1) {
        recv(client_data->socket, buffer, sizeof(buffer), 0); if (strcmp(buffer, "STOP\n") == 0) {
            printf("Client %s:%d requested to stop communication.\n",
                inet_ntoa(client_data->address.sin_addr),
                ntohs(client_data->address.sin_port));
            client_data->stop_flag = 1;
            break;
        }
        printf("Client %s:%d: %s", inet_ntoa(client_data->address.sin_addr),
            ntohs(client_data->address.sin_port), buffer);
        printf("Enter message for client: ");
        fgets(buffer, sizeof(buffer), stdin);
        if (client_data->stop_flag) {
            printf("Server: Stopping communication with %s:%d\n",
                inet_ntoa(client_data->address.sin_addr),
                ntohs(client_data->address.sin_port));
            break;
        }
        send(client_data->socket, buffer, strlen(buffer), 0); }
    close(client_data->socket);
    printf("Client disconnected: %s:%d\n", inet_ntoa(client_data->address.sin_addr),
        ntohs(client_data->address.sin_port));
    free(client_data);
    pthread_exit(NULL);
}
int main() {
    int server_socket, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(client_addr);
```

```

server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1) {
    perror("Error creating socket");
    exit(EXIT_FAILURE);
}
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT); if (bind(server_socket, (struct sockaddr*)&server_addr,
sizeof(server_addr)) == -1) {
    perror("Error binding socket");
    exit(EXIT_FAILURE);
}
if (listen(server_socket, MAX_CLIENTS) == -1) {
    perror("Error listening for connections");
    exit(EXIT_FAILURE);
}
printf("Server listening on port %d...\n", PORT);
while (1) {
    new_socket = accept(server_socket, (struct
sockaddr*)&client_addr, &addr_size);
    if (new_socket == -1) {
        perror("Error accepting connection");
        exit(EXIT_FAILURE);
    }
    struct ClientData *client_data = (struct ClientData
*)malloc(sizeof(struct ClientData));
    client_data->socket = new_socket; client_data->address = client_addr;
    client_data->stop_flag = 0;
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, handle_client, (void
*)client_data) != 0) {
        perror("Error creating thread");
        exit(EXIT_FAILURE);
    }
    pthread_detach(thread_id);
}
close(server_socket);
return 0;
}

```

b. Client11.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h> #define PORT 8080
#define SERVER_IP "127.0.0.1"
int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[1024];

```

```

client_socket = socket(AF_INET, SOCK_STREAM, 0);
if (client_socket == -1) {
    perror("Error creating socket");
    exit(EXIT_FAILURE);
}
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr)
    <= 0) {
    perror("Invalid address/ Address not supported");
    exit(EXIT_FAILURE);
} if (connect(client_socket, (struct sockaddr*)&server_addr,
    sizeof(server_addr)) == -1) {
    perror("Error connecting to server");
    exit(EXIT_FAILURE);
}
printf("Connected to server\n");
while (1) {
    printf("Enter message (type 'STOP' to quit): ");
    fgets(buffer, sizeof(buffer), stdin);
    send(client_socket, buffer, strlen(buffer), 0);
    if (strcmp(buffer, "STOP\n") == 0) {
        printf("Client exiting...\n");
        break;
    }
    memset(buffer, 0, sizeof(buffer)); // Clear the buffer
    before receiving recv(client_socket, buffer, sizeof(buffer), 0);
    printf("Server: %s", buffer);
}
close(client_socket);
return 0;
}

```

c. Client2.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#define PORT 8080
#define SERVER_IP "127.0.0.1"
int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[1024]; client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1) {
        perror("Error creating socket");
        exit(EXIT_FAILURE);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

```

```

if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
perror("Invalid address/ Address not supported");
exit(EXIT_FAILURE);
}
if (connect(client_socket, (struct sockaddr*)&server_addr,
sizeof(server_addr)) == -1) {
perror("Error connecting to server");
exit(EXIT_FAILURE);
}
printf("Connected to server\n");
while (1) {
printf("Enter message (type 'exit' to quit): "); fgets(buffer, sizeof(buffer), stdin);
send(client_socket, buffer, strlen(buffer), 0);
if (strcmp(buffer, "STOP\n") == 0) {
break;
}
recv(client_socket, buffer, sizeof(buffer), 0);
printf("Server: %s", buffer);
}
close(client_socket);
return 0;
}

```

Output:

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc client11.c -o client
ubuntu@ubuntu-virtual-machine:~/Desktop$ chmod +x client
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./client
Connected to server
Enter message (type 'STOP' to quit): heloo
Server: hi
Enter message (type 'STOP' to quit): i am student name shrijan
Server: hello
Enter message (type 'STOP' to quit): STOP
Client exiting...
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

```
ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc server2.c -o ser
ubuntu@ubuntu-virtual-machine:~/Desktop$ chmod +x ser
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./ser
Server listening on port 8080...
Client 127.0.0.1:38820: heloo
Enter message for client: hi
Client 127.0.0.1:38820: i am student name shrijan
Enter message for client: Client 127.0.0.1:38804: hello
hello
Enter message for client: heelo
Client 127.0.0.1:38804: iam student rishabh
Enter message for client: ok
Client 127.0.0.1:38804: STOP
tudent rishabh
Enter message for client: Client 127.0.0.1:38820: STOP
```

```
ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc client2.c -o cl
ubuntu@ubuntu-virtual-machine:~/Desktop$ chmod +x cl
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./cl
Connected to server
Enter message (type 'STOP' to quit): hello
Server: heelo
Enter message (type 'STOP' to quit): iam student rishabh
Server: ok
student rishabh
Enter message (type 'STOP' to quit): STOP
Client exiting...
ubuntu@ubuntu-virtual-machine:~/Desktop$
```

Experiment-7

Aim: 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 of the processes

Theory: To achieve two-way inter-process communication using Distributed Shared Memory (DSM) between independent processes, you can follow these simplified steps:

1. Process Initialization:

- Identify the nodes where your independent processes will run.
- Initialize DSM on each node to create a shared memory region.

2. Shared Memory Mapping:

- Map the shared memory into the address space of each process, allowing them to access it as if it's local.

3. Data Structures and Synchronization:

- Define the data structures and synchronization mechanisms to control access to shared data.
- Use basic synchronization primitives like locks or semaphores.

4. Reading and Writing Data:

- Processes can read from and write to the shared memory. - Agree on a common data structure and protocol for communication.

5. Communication Protocol:

- Implement a simple protocol for processes to communicate via the shared memory.
- Use shared variables to exchange data between processes.

6. Synchronization and Error Handling:

- Ensure proper synchronization to avoid race conditions and data corruption.
- Implement basic error handling for process crashes or failures.

This simplified approach is suitable for educational purposes or simple applications. In practice, more advanced techniques and libraries are often used to handle the complexity and challenges of distributed shared memory systems.**Code:**

Process1:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>
#define SHM_SIZE 1024
```



```

#define KEY 12345
struct shared_data {
char message[SHM_SIZE];
bool stop;
bool ready;
};
int shmid;
int semid;
struct shared_data* shared_memory = NULL; void signal_handler(int signo) {
if (signo == SIGINT || signo == SIGTERM) {
printf("Process 1 received stop signal. Cleaning up...\n");
shared_memory->stop = true;
}
}
int main() {
// Create or access the shared memory segment
shmid = shmget(KEY, sizeof(struct shared_data), 0666 | IPC_CREAT);
if (shmid == -1) {
perror("shmget");
exit(1);
}
// Attach to the shared memory
shared_memory = shmat(shmid, NULL, 0);
if (shared_memory == (struct shared_data*)-1) {
perror("shmat");
exit(1);
}
// Create or access the semaphore
semid = semget(KEY, 1, 0666 | IPC_CREAT);
if (semid == -1) {
perror("semget"); exit(1);
}
// Initialize semaphore
semctl(semid, 0, SETVAL, 1);
// Set up a signal handler for stopping the communication
signal(SIGINT, signal_handler);
signal(SIGTERM, signal_handler);
// Communication loop
while (!shared_memory->stop) {
// Wait for the semaphore
struct sembuf wait_buf = {0, -1, 0};
semop(semid, &wait_buf, 1);
if (!shared_memory->stop) {
if (shared_memory->ready) {
if (strcmp(shared_memory->message, "STOP") == 0) {
printf("Process 1 received stop signal. Cleaning up...\n");
shared_memory->stop = true;
} else {
printf("Message received by Process 1 (or 'STOP' to quit) : %s\n",
shared_memory->message);

```

```

}
printf("Enter a message for Process 2 : ");
fgets(shared_memory->message, SHM_SIZE, stdin); // Remove newline character
size_t len = strlen(shared_memory->message);
if (shared_memory->message[len - 1] == '\n') {
shared_memory->message[len - 1] = '\0';
}
if (strcmp(shared_memory->message, "STOP") == 0) {
shared_memory->stop = true;
}
printf("Process 1 sent: %s\n", shared_memory->message);
shared_memory->ready = false;
}
}
// Release the semaphore
struct sembuf signal_buf = {0, 1, 0};
semop(semid, &signal_buf, 1);
// Wait for Process 2 to signal readiness
while (!shared_memory->ready && !shared_memory->stop) {
usleep(100000); // Sleep for 0.1 seconds
}
} // Clean up
shmdt(shared_memory);
shmctl(shmid, IPC_RMID, NULL);
semctl(semid, 0, IPC_RMID);
return 0;
}
Process2:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>
#define SHM_SIZE 1024
#define KEY 12345
struct shared_data {
char message[SHM_SIZE];
bool stop;
bool ready;
};int shmid;
int semid;
struct shared_data* shared_memory = NULL;
void signal_handler(int signo) {
if (signo == SIGINT || signo == SIGTERM) {
printf("Process 2 received stop signal. Cleaning up...\n");

```

```

shared_memory->stop = true;
}
}
int main() {
// Create or access the shared memory segment
shmid = shmget(KEY, sizeof(struct shared_data), 0666 | IPC_CREAT);
if (shmid == -1) {
perror("shmget");
exit(1);
}
// Attach to the shared memory
shared_memory = shmat(shmid, NULL, 0);
if (shared_memory == (struct shared_data*)-1) {
perror("shmat");
exit(1);
}
// Create or access the semaphore
semid = semget(KEY, 1, 0666 | IPC_CREAT);
if (semid == -1) {
perror("semget");
exit(1); }
// Initialize semaphore
semctl(semid, 0, SETVAL, 1);
// Set up a signal handler for stopping the communication
signal(SIGINT, signal_handler);
signal(SIGTERM, signal_handler);
// Communication loop
while (!shared_memory->stop) {
// Wait for the semaphore
struct sembuf wait_buf = {0, -1, 0};
semop(semid, &wait_buf, 1);
if (!shared_memory->stop) {
if (shared_memory->ready) {
if (strcmp(shared_memory->message, "STOP") == 0) {
printf("Process 2 received stop signal. Cleaning up...\n");
shared_memory->stop = true;
} else {
printf("Message received by Process 2 : %s\n", shared_memory->message);
}
printf("Enter a message for Process 1 (or 'STOP' to quit): ");
fgets(shared_memory->message, SHM_SIZE, stdin);
// Remove newline character
size_t len = strlen(shared_memory->message);
if (shared_memory->message[len - 1] == '\n') {
shared_memory->message[len - 1] = '\0';
} if (strcmp(shared_memory->message, "STOP") == 0) {
printf("Process 2 received stop signal. Cleaning up...\n");
shared_memory->stop = true;
}
printf("Process 2 sent: %s\n", shared_memory->message);
}
}
}

```

```

shared_memory->ready = false;
}
}
// Release the semaphore
struct sembuf signal_buf = {0, 1, 0};
semop(semid, &signal_buf, 1);
// Signal readiness for Process 1
shared_memory->ready = true;
}
// Clean up
shmdt(shared_memory);
shmctl(shmid, IPC_RMID, NULL);
semctl(semid, 0, IPC_RMID);
return 0;
}

```

Output:

PROCESS 1:

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc p1.c -o process1
ubuntu@ubuntu-virtual-machine:~/Desktop$ chmod +x process1
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./process1
Message received by Process 1 (or 'STOP' to quit) : hello
Enter a message for Process 2 : how are you process2
Process 1 sent: how are you process2
Message received by Process 1 (or 'STOP' to quit) : i am good every thing is alright
Enter a message for Process 2 : yup
Process 1 sent: yup
Message received by Process 1 (or 'STOP' to quit) : ok buy
Enter a message for Process 2 : ya
Process 1 sent: ya
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

PROCESS 2:

```

ubuntu@ubuntu-virtual-machine:~$ cd Desktop
ubuntu@ubuntu-virtual-machine:~/Desktop$ gcc p2.c -o process2
ubuntu@ubuntu-virtual-machine:~/Desktop$ chmod +x process2
ubuntu@ubuntu-virtual-machine:~/Desktop$ ./process2
Message received by Process 2 :
Enter a message for Process 1 (or 'STOP' to quit): hello
Process 2 sent: hello
Message received by Process 2 : how are you process2
Enter a message for Process 1 (or 'STOP' to quit): i am good every thing is alright
Process 2 sent: i am good every thing is alright
Message received by Process 2 : yup
Enter a message for Process 1 (or 'STOP' to quit): ok buy
Process 2 sent: ok buy
Message received by Process 2 : ya
Enter a message for Process 1 (or 'STOP' to quit): STOP
Process 2 received stop signal. Cleaning up...
Process 2 sent: STOP
ubuntu@ubuntu-virtual-machine:~/Desktop$

```

Experiment-8

AIM/OBJECTIVE:

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:

1. Program for calculator.x file

```
struct numbers{
int a;
int b;
};
program CALC_PROG{
version CALC_VERS{
int add(numbers)=1;
int sub(numbers)=2;
int mul(numbers)=3;
float div(numbers)=4;
}=1;
}=0x4562877;
```

2. Program for 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 "calculator.h"
void calc_prog_1(char *host, int x, int y)
{
    CLIENT *clnt;
    int *result_1;
    numbers add_1_arg;
    int *result_2;
    numbers sub_1_arg;
    int *result_3;
    numbers mul_1_arg;
    float *result_4;
    numbers div_1_arg;
    #ifndef DEBUG
    clnt = clnt_create (host, CALC_PROG, CALC_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 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("\nSum = %d", *result_1);
    sub_1_arg.a=x;
    sub_1_arg.b=y;
    result_2 = sub_1(&sub_1_arg, clnt);if (result_2 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("\nDifference = %d", *result_2);
    mul_1_arg.a=x;
    mul_1_arg.b=y;
    result_3 = mul_1(&mul_1_arg, clnt);
    if (result_3 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("\nProduct = %d", *result_3);
    div_1_arg.a=x;
    div_1_arg.b=y;
    result_4 = div_1(&div_1_arg, clnt);
    if (result_4 == (float *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    printf("\nDivision = %f", *result_4);

```

```

#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];
calc_prog_1 (host,atoi(argv[2]),atoi(argv[3]));
exit (0);
}
3. Program for 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 "calculator.h"
int *add_1_svc(numbers *argp, struct svc_req *rqstp)
{
static int result;
printf("add(%d,%d) is called\n",argp->a,argp->b);
result = argp->a + argp->b;
return &result;}
int *sub_1_svc(numbers *argp, struct svc_req *rqstp)
{
static int result;
printf("sub(%d,%d) is called\n",argp->a,argp->b);
result = argp->a - argp->b;
return &result;
}
int *mul_1_svc(numbers *argp, struct svc_req *rqstp)
{
static int result;
printf("mul(%d,%d) is called\n",argp->a,argp->b);
result = argp->a * argp->b;
return &result;
}
float *div_1_svc(numbers *argp, struct svc_req *rqstp)
{
static float result;
printf("div(%d,%d) is called\n",argp->a,argp->b);
if (argp->b != 0)
result = argp->a / argp->b;
else
result = 0;
}

```

return &result;

}

OUTPUT:

```
tanmay13das@DESKTOP-MUQPOIG: ~/ADS_Lab08
tanmay13das@DESKTOP-MUQPOIG:~$ cd ADS_Lab08
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client
[sudo] password for tanmay13das:
usage: ./calculator_client server: host
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client localhost 12 6
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client localhost 12 6
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client localhost 12 6

Sum = 18
Difference = 6
Product = 72
Division = 2.000000tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client localhost 123
usage: ./calculator_client server: host
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_client localhost 123 41

Sum = 164
Difference = 82
Product = 5043
Division = 3.000000tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$

tanmay13das@DESKTOP-MUQPOIG: ~/ADS_Lab08
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ make -f Makefile.add
Command 'make' not found, but can be installed with:

sudo apt install make
sudo apt install make-guile

tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ cd ..
tanmay13das@DESKTOP-MUQPOIG:~$ sudo apt install make
[sudo] password for tanmay13das:

[1]* Stopped          sudo apt install make
tanmay13das@DESKTOP-MUQPOIG:~$ make -f Makefile.add
make: Makefile.add: No such file or directory
make: *** No rule to make target 'Makefile.add'. Stop.
tanmay13das@DESKTOP-MUQPOIG:~$ make -f Makefile.calculator
make: Makefile.calculator: No such file or directory
make: *** No rule to make target 'Makefile.calculator'. Stop.
tanmay13das@DESKTOP-MUQPOIG:~$ cd ADS_Lab08
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ make -f Makefile.calculator
cc -g      -c -o calculator_clnt.o calculator_clnt.c
cc -g      -c -o calculator_client.o calculator_client.c
cc -g      -c -o calculator_xdr.o calculator_xdr.c
cc -g      -o calculator_client  calculator_clnt.o calculator_client.o calculator_xdr.o -lnsl
cc -g      -c -o calculator_svc.o calculator_svc.c
cc -g      -c -o calculator_server.o calculator_server.c
cc -g      -o calculator_server  calculator_svc.o calculator_server.o calculator_xdr.o -lnsl
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_server
[sudo] password for tanmay13das:
^Z
[2]* Stopped          sudo ./calculator_server
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ make -f Makefile.calculator
cc -g      -c -o calculator_clnt.o calculator_clnt.c
cc -g      -c -o calculator_client.o calculator_client.c
cc -g      -c -o calculator_xdr.o calculator_xdr.c
cc -g      -o calculator_client  calculator_clnt.o calculator_client.o calculator_xdr.o -lnsl
cc -g      -c -o calculator_svc.o calculator_svc.c
cc -g      -c -o calculator_server.o calculator_server.c
cc -g      -o calculator_server  calculator_svc.o calculator_server.o calculator_xdr.o -lnsl
tanmay13das@DESKTOP-MUQPOIG:~/ADS_Lab08$ sudo ./calculator_server
add(12,6) is called
sub(12,6) is called
mul(12,6) is called
div(12,6) is called
add(123,41) is called
sub(123,41) is called
mul(123,41) is called
div(123,41) is called
```