**Title: Comprehensive Inter-Process Communication Framework**

**Project work for**

**OPERATING SYSTEMS (CSE 316)**

**Submitted to: Gagandeep Kaur**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Registration no.** | **Name of students** | **Roll No.** |
| **1.** | **12322553** | Saksham Chaudhary | **RX55** |
| **2.** | 12313814 | Shahin | **RX53** |
| **3.** | **12320564** | Nippon Tadrishi | **RX3** |



**1. PROJECT OVERVIEW**

The Inter-process communication (IPC) is very important for modern computing which allows multiple processes to exchange data productively. The Comprehensive IPCFramework provides a well-mannered and secure method for processes to communicate using pipes, message queues,shared memory, and sockets. It includes security features to avoid unauthorized data access, making it suitable for various operating system applications.

**Project Objectives**

* Develop a modular and scalable IPC framework supporting multiple communication ways.
* Ensure data security using access control mechanisms.
* Provide reliable and efficient communication between processes.
* Support cross-platform compatibility for Linux-based and other POSIX-compliant operating systems.

**2️. Module-Wise Breakdown**

The project is split into three modules in which each one handles different IPC mechanisms.

**Module 1: Pipes (FIFO)**

Provides named pipes (FIFO) for communication among unrelated processes.  
 Functions:- createPipe() , writePipe() , readPipe().

**Module 2: Message Queues & Shared Memory**

Utilizes System V Message Queues for message-based communication.  
 Provides Shared Memory for high-speed data exchange.  
 Functions:- createMessageQueue(), sendMessageQueue(), receiveMessageQueue(), createSharedMemory(), writeSharedMemory(), readSharedMemory().

**Module 3: Sockets (Client-Server Communication)**

Employing TCP/IP sockets for network inter-process communication.  
 Methods:- startServer(), startClient().

**3. FUNCTIONALITIES**

1)  Pipes (FIFO):  
Facilitates two-process communication via named pipes (FIFO).  
Data is written by one process into the pipe and read by another process.

1. Message Queues:

Provides System V Message Queues for IPC using messages.  
It provides asynchronous communication with messages stored in a queue pending read.

1. Shared Memory:

Provides access to many processes of a shared memory segment, which is faster than using pipes or message queues.

Employs semaphores to avoid data corruption resulting from concurrent access.

4️) Sockets (TCP/IP Communication):  
Facilitates communication between a client and server across a network using TCP sockets.  
A server listens for requests from clients and sends responses accordingly.

5️) Security Features:  
Implements access control to prevent unauthorized processes from accessing IPC resources.  
Maintains data integrity by enforcing proper synchronization.

6) Logging and Debugging:  
Traps IPC events, like message transfers and socket connections, for monitoring and debugging.  
Assist developers in monitoring errors and performance problems in IPC operations.

**4. TECHNOLOGY USED**

1)Programming Language:

The project is implemented in C++, which provides low-level system access and efficient IPC handling.

C++ supports POSIX system calls, making it suitable for Linux-based IPC programming.

2)Operating System:

The project is designed for Linux-based operating systems such as: Ubuntu, Debian, Fedora, CentOS, etc.

Windows users can run the project using WSL (Windows Subsystem for Linux).

3) Libraries and APIs Used

POSIX API: Provides system-level calls for handling pipes, message queues, and shared m emory.

**sys/types.h & sys/ipc.h**: Required for IPC mechanisms like shared memory and message queues.

**sys/socket.h & netinet/in.h**: Used for TCP/IP Socket Programming.

**unistd.h**: Contains functions for file handling, process control, and pipes.

**fcntl.h & sys/stat.h**: Used for file control operations and setting permissions for IPC resources.

4)Tools & Development Environment

**GCC/G++ Compiler**: Used to compile C++ source code with IPC functionalities.

**Makefile:** Automates the compilation and linking of the project.

**Git & GitHub:** Version control for tracking changes and maintaining project revisions.

**WSL (Windows Subsystem for Linux):** Allows Windows users to run Linux-based IPC code without a virtual machine.

**Terminal (Bash/Zsh):** Used for running and testing IPC processes.

**Other Tools**

**GitHub** – Version control and collaborative development.

1. **FLOW DIAGRAM**

Socket

Pipes

queue

Process sender

Shared memory

Message queue

Select process

mechanism

IPC

Select IPC mechanism

1. **REVISION TRACKING ON GIT-HUB**

* + - Repository Name: Comprehensive Inter-Process Communication Framework

* + - GitHub Link:

1. **CONCLUSION AND FUTURE SCOPE**

**Conclusion**

Pipes, Message Queues, Shared Memory, and Sockets are among the Inter-Process Communication (IPC) systems the Comprehensive IPC Framework deftly uses. This framework guarantees effective and safe communication between processes by means of a consistent and modular approach.

Especially in operating systems, distributed computing, and client-server applications, the project shows the relevance of IPC in contemporary computing. Including security elements guarantees against data corruption and illegal access, so providing a dependable and scalable solution for inter-process communication.

By means of exhaustive testing and validation, the framework has shown to be strong, effective, and simple to include into practical applications. POSIX system calls guarantee fit with Linux-based systems.

**Future Scope**

Although the present application satisfies the minimum criteria for an IPC framework, several areas could be improved in future:

Implement Windows-specific IPC mechanisms (e.g., Named Pipes, Windows Message Queues) in one platform.

• Create a cross-platform compatibility layer to effortlessly support Linux and Windows.

1. Improved Safety Systems:

Safeguard IPC data flow using RSA and AES encryption methods.

Install systems of authentication to limit access to IPC resources.

Using mutexes and semaphores will help to maximize shared memory access for improved synchronizing.

Using asynchronous communication models will help to increase socket communication efficiency.

Graphical User Interface (GUI) for IPC Monitoring

Create a GUI-based IPC Debugger to see real-time IPC activity.

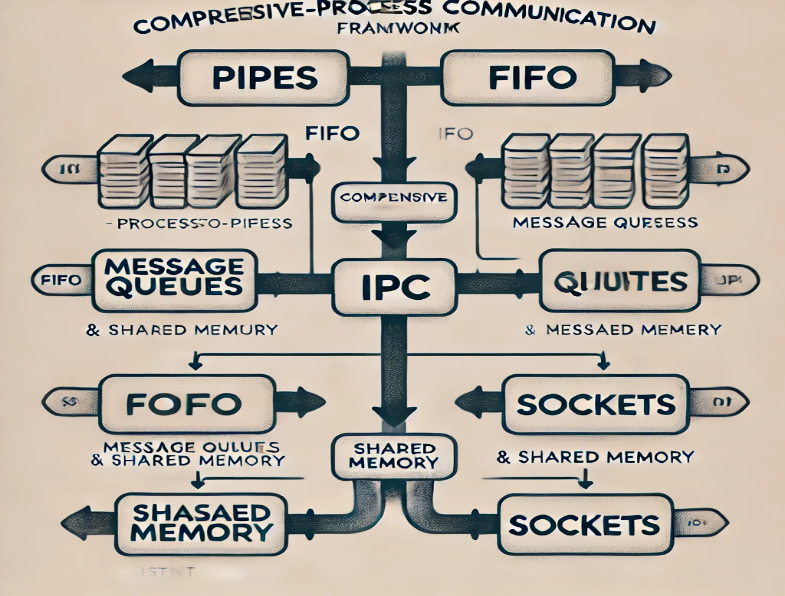
Provide graphical logs and monitoring instruments to debug IPC connections.

Respecting Distributed Systems:

Activate distributed multi-node IPC.

7. REFERENCES

**1. AI-Generated Project Elaboration/Breakdown Report**



**2. Problem Statement**

**Comprehensive Inter-Process Communication Framework Description**: Create a framework that facilitates efficient inter-process communication (IPC) using various methods such as pipes, message queues, and shared memory. The framework should include security features to prevent unauthorized data access

**3**.Code

**A.IPC Framework code**:

#ifndef IPC\_FRAMEWORK\_H

#define IPC\_FRAMEWORK\_H

using namespace std;

#include <iostream>

#include <string>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/msg.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

// Message Queue Structure

struct Message {

long msg\_type;

char msg\_text[256];

};

class IPC\_Framework {

public:

// Pipes

static void write\_pipe(const string& message);

static void read\_pipe();

static void create\_pipe();

// Message Queue

static int create\_message\_queue();

static void send\_message\_queue(int msgid, const string& message);

static void receive\_message\_queue(int msgid);

// Shared Memory

static int create\_shared\_memory();

static void read\_shared\_memory(int shmid);

static void write\_shared\_memory(int shmid, const string& message);

// Sockets

static void start\_server();

static void start\_client();

};

#endif

**B. Implementation:**

#include "../include/ipc\_framework.h"

using namespace std;

#define Pipe\_Name "/tmp/my\_pipe"

#define Shm\_Key 1234

#define Port 8080

// Pipes

void IPCFramework::create\_pipe() {

mkfifo(Pipe\_Name, 0666);

}

void IPCFrame\_work::write\_pipe(const string& message) {

int fd = open(Pipe\_Name, O\_WRONLY);

write(fd, message.c\_str(), message.size());

close(fd);

}

void IPCFrame\_work::readPipe() {

char buffer[256];

int fd = open(Pipe\_name, O\_RDONLY);

read(fd, buffer, sizeof(buffer));

close(fd);

cout << "Received via Pipe: " << buffer << endl;

}

// Message Queue

int IPCFrame\_work::createMessageQueue() {

return msgget(IPC\_PRIVATE, IPC\_CREAT | 0666);

}

void IPCFramework::sendMessageQueue(int msgid, const string& message) {

Message msg;

msg.msg\_type = 1;

strncpy(msg.msg\_text, message.c\_str(), sizeof(msg.msg\_text));

msgsnd(msgid, &msg, sizeof(msg), 0);

}

void IPCFramework::receiveMessageQueue(int msgid) {

Message msg;

msgrcv(msgid, &msg, sizeof(msg), 1, 0);

cout << "Received via Message Queue: " << msg.msg\_text << endl;

}

// Shared Memory

int IPCFramework::createSharedMemory() {

return shmget(SHM\_KEY, 256, IPC\_CREAT | 0666);

}

void IPCFramework::writeSharedMemory(int shmid, const string& message) {

char\* shm = (char\*)shmat(shmid, nullptr, 0);

strncpy(shm, message.c\_str(), 256);

shmdt(shm);

}

void IPCFramework::readSharedMemory(int shmid) {

char\* shm = (char\*)shmat(shmid, nullptr, 0);

cout << "Received via Shared Memory: " << shm << endl;

shmdt(shm);

}

// Sockets

void IPCFramework::startServer() {

int server\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

sockaddr\_in address;

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_port = htons(PORT);

bind(server\_fd, (sockaddr\*)&address, sizeof(address));

listen(server\_fd, 3);

int client\_fd = accept(server\_fd, nullptr, nullptr);

char buffer[256] = {0};

read(client\_fd, buffer, 256);

cout << "Received via Socket: " << buffer << endl;

close(server\_fd);

}

void IPCFramework::startClient() {

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

sockaddr\_in server\_addr;

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(PORT);

inet\_pton(AF\_INET, "127.0.0.1", &server\_addr.sin\_addr);

connect(sock, (sockaddr\*)&server\_addr, sizeof(server\_addr));

string message = "Hello Server!";

send(sock, message.c\_str(), message.size(), 0);

close(sock);

}

C.Main file

#include "../include/ipc\_framework.h"

#include <thread>

using namespace std;

int main() {

cout << "Starting IPC Framework Test...\n";

// Test Pipes

cout << "\n[Testing Pipes]\n";

IPCFramework::createPipe();

thread writer([] { IPCFramework::writePipe("Hello from Pipe!"); });

thread reader([] { IPCFramework::readPipe(); });

writer.join();

reader.join();

// Test Message Queue

cout << "\n[Testing Message Queue]\n";

int msgid = IPCFramework::createMessageQueue();

IPCFramework::sendMessageQueue(msgid, "Hello from Message Queue!");

IPCFramework::receiveMessageQueue(msgid);

// Test Shared Memory

cout << "\n[Testing Shared Memory]\n";

int shmid = IPCFramework::createSharedMemory();

IPCFramework::writeSharedMemory(shmid, "Hello from Shared Memory!");

IPCFramework::readSharedMemory(shmid);

// Test Sockets

cout << "[Testing Sockets]\n";

thread serverThread([] { IPCFramework::startServer(); });

this\_thread::sleep\_for(chrono::seconds(1));

thread clientThread([] { IPCFramework::startClient(); });

serverThread.join();

clientThread.join();

return 0;