**Inter-Process Communication (IPC) Framework**

Contents

[**1.Project overview:** 2](#_Toc176507987)

[**2.Scope:** 3](#_Toc176507988)

[**3. Functional Requirements:** 3](#_Toc176507989)

[**4. Non-functional Requirements:** 3](#_Toc176507990)

[**5.User Roles:** 4](#_Toc176507991)

[**6. Input/Output Specifications:** 4](#_Toc176507992)

[**6.1 Input** 4](#_Toc176507993)

[**6.2 Output** 4](#_Toc176507994)

[**7. Challenges:** 5](#_Toc176507995)

[**8.Code:** 5](#_Toc176507996)

[**9. Testing and Validation:** 11](#_Toc176507997)

[**10. Documentation:** 15](#_Toc176507998)

[**10.1 User Guide** 15](#_Toc176507999)

[**10.2 Implementation** 16](#_Toc176508000)

[**11. Milestones:** 16](#_Toc176508001)

[**12.References:** 17](#_Toc176508002)

# 

# 

# **1.Project overview:**

Project Title: Inter-Process Communication (IPC) Framework

Description: Develop a robust Inter-Process Communication (IPC) framework that allows

multiple processes to communicate with each other efficiently. The framework will support

various IPC mechanisms such as message queues, shared memory, and semaphores,

providing a unified API for process synchronization and data exchange.

**Objective:**

Implement a flexible and scalable IPC framework that abstracts underlying

complexities, offering a clean and intuitive interface for developers to utilize in their

applications.

# **2.Scope:**

Target Environment: Linux-based operating systems.

Primary Audience: System programmers, developers working on multi-process applications,

and students learning about process communication.

**Technologies:**

Programming Language: C++

Libraries: POSIX IPC mechanisms (message queues, shared memory, semaphores)

# **3. Functional Requirements:**

* **Message Queue Management:** The framework should provide functions to create, send, and receive messages through message queues.
* **Shared Memory Management:** The framework should allow processes to allocate, attach, and detach shared memory segments.
* **Semaphore Operations:** The framework should include semaphore initialization, wait (lock), and signal (unlock) operations for synchronization.
* **Unified API:** The framework should offer a unified, easy-to-use API for developers, abstracting the complexities of underlying IPC mechanisms.
* **Error Handling:** The framework should include robust error handling to deal with system-level errors like resource allocation failures or communication interruptions.

# **4. Non-functional Requirements:**

* **Performance:** The IPC framework should have minimal overhead, ensuring efficient inter-process communication and synchronization.
* **Scalability:** The framework should support scaling to multiple processes without significant performance degradation.
* **Portability:** The framework should be portable across UNIX-like systems (e.g., Linux, BSD), adhering to POSIX standards.
* **Reliability:** The framework should be reliable, ensuring data integrity and correct process synchronization even under heavy load.
* **Usability:** The API should be intuitive, well-documented, and easy to integrate into existing projects.

# **5.User Roles:**

* **Developers**: Primary users who will integrate the IPC framework into their applications, utilizing its API for communication and synchronization.
* **Test Engineers**: Users responsible for testing the framework’s functionality, performance, and reliability, ensuring it meets the specified requirements.
* **System Administrators**: Users who might be involved in deploying applications that use the IPC framework, ensuring the system environment is set up correctly.

# **6. Input/Output Specifications:**

## **6.1 Input**

* Key values for IPC resource identification (e.g., ftok generated keys).
* Message data for queues (structured messages with a type and text).
* Data to be shared in memory segments.
* Semaphore operations indicating process synchronization requirements.

## **6.2 Output**

* Successful communication between processes, verified through message reception and shared memory content.
* Proper synchronization of processes using semaphores.
* Error messages and logs for debugging any issues with IPC mechanisms.

# **7. Challenges:**

* **Resource Management:** Efficiently managing IPC resources like message queues, shared memory, and semaphores to prevent leaks and ensure proper cleanup.
* **Concurrency:** Handling concurrent access to shared resources, ensuring data consistency and process synchronization.
* **Error Handling:** Providing robust error handling mechanisms that can gracefully recover from IPC failures.
* **Testing:** Thoroughly testing the framework to ensure it performs reliably under various scenarios, including stress testing with multiple processes.
* **Portability:** Ensuring the framework works across different UNIX-like systems without requiring significant changes.

# **8.Code:**

# **ipc\_framework.h**

#ifndef IPC\_FRAMEWORK\_H

#define IPC\_FRAMEWORK\_H

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <string>

struct msg\_buffer {

long msg\_type;

char msg\_text[100];

};

int init\_message\_queue(key\_t key);

int send\_message(int msgid, struct msg\_buffer \*message);

int receive\_message(int msgid, struct msg\_buffer \*message, long msg\_type);

int init\_shared\_memory(key\_t key, size\_t size);

void\* attach\_shared\_memory(int shmid);

int init\_semaphore(key\_t key, int num\_sems);

int semaphore\_wait(int semid, int sem\_num);

int semaphore\_signal(int semid, int sem\_num);

#endif // IPC\_FRAMEWORK\_H

**ipc\_framework.cpp**

#include "ipc\_framework.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

// Message Queue Functions

int init\_message\_queue(key\_t key) {

int msgid = msgget(key, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("msgget");

exit(EXIT\_FAILURE);

}

return msgid;

}

int send\_message(int msgid, struct msg\_buffer \*message) {

if (msgsnd(msgid, message, sizeof(message->msg\_text), 0) == -1) {

perror("msgsnd");

return -1;

}

return 0;

}

int receive\_message(int msgid, struct msg\_buffer \*message, long msg\_type) {

if (msgrcv(msgid, message, sizeof(message->msg\_text), msg\_type, 0) == -1) {

perror("msgrcv");

return -1;

}

return 0;

}

// Shared Memory Functions

int init\_shared\_memory(key\_t key, size\_t size) {

int shmid = shmget(key, size, 0666 | IPC\_CREAT);

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

return shmid;

}

void \*attach\_shared\_memory(int shmid) {

void \*shmaddr = shmat(shmid, NULL, 0);

if (shmaddr == (void \*)-1) {

perror("shmat");

exit(EXIT\_FAILURE);

}

return shmaddr;

}

// Semaphore Functions

int init\_semaphore(key\_t key, int num\_sems) {

int semid = semget(key, num\_sems, 0666 | IPC\_CREAT);

if (semid == -1) {

perror("semget");

exit(EXIT\_FAILURE);

}

// Initialize semaphore value

union semun {

int val;

} sem\_union;

sem\_union.val = 1; // Set initial value of semaphore to 1

if (semctl(semid, 0, SETVAL, sem\_union) == -1) {

perror("semctl SETVAL");

exit(EXIT\_FAILURE);

}

return semid;

}

int semaphore\_wait(int semid, int sem\_num) {

struct sembuf sops = {static\_cast<short unsigned int>(sem\_num), -1, 0};

if (semop(semid, &sops, 1) == -1) {

perror("semop wait");

return -1;

}

return 0;

}

int semaphore\_signal(int semid, int sem\_num) {

struct sembuf sops = {static\_cast<short unsigned int>(sem\_num), 1, 0};

if (semop(semid, &sops, 1) == -1) {

perror("semop signal");

return -1;

}

return 0;

}

**main.cpp**

#include "ipc\_framework.h"

#include <iostream>

#include <cstring>

int main() {

key\_t key = ftok("progfile", 65);

int msgid = init\_message\_queue(key);

int shmid = init\_shared\_memory(key, 1024);

int semid = init\_semaphore(key, 1);

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello, IPC Framework!");

if (send\_message(msgid, &message) == 0) {

std::cout << "Message sent successfully." << std::endl;

}

if (receive\_message(msgid, &message, 1) == 0) {

std::cout << "Received message: " << message.msg\_text << std::endl;

}

char\* shared\_data = static\_cast<char\*>(attach\_shared\_memory(shmid));

if (semaphore\_wait(semid, 0) == 0) {

std::cout << "Semaphore acquired." << std::endl;

strcpy(shared\_data, "Shared data example");

std::cout << "Shared Memory (after writing): " << shared\_data << std::endl;

if (semaphore\_signal(semid, 0) == 0) {

std::cout << "Semaphore released." << std::endl;

}

} else {

std::cout << "Failed to acquire semaphore." << std::endl;

}

return 0;

}

# **9. Testing and Validation:**

**Unit Testing**

* Test each IPC mechanism individually (message queues, shared memory, semaphores).
* Verify correct creation, communication, and cleanup.

**Integration Testing**

* Test the framework as a whole by simulating scenarios where multiple processes use different IPC mechanisms simultaneously.
* Ensure that processes can communicate and synchronize correctly.

**Concurrency Testing**

* Ensuring the framework handles concurrent processes correctly without data corruption or synchronization issues.
* Test for race conditions, deadlocks, and proper synchronization.

**Error Handling Tests**

* Simulating failure scenarios to verify that the framework handles errors gracefully and recovers correctly.

**Test code**

**test\_ipc\_framework.cpp**

#include "ipc\_framework.h"

#include <iostream>

#include <cstring>

#include <pthread.h>

void test\_message\_queue() {

key\_t key = ftok("progfile", 65);

int msgid = init\_message\_queue(key);

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Test Message");

if (send\_message(msgid, &message) == 0) {

std::cout << "Unit Test: Message sent successfully." << std::endl;

}

if (receive\_message(msgid, &message, 1) == 0) {

std::cout << "Unit Test: Received message: " << message.msg\_text << std::endl;

}

}

void test\_shared\_memory() {

key\_t key = ftok("progfile", 66);

int shmid = init\_shared\_memory(key, 1024);

char\* shared\_data = static\_cast<char\*>(attach\_shared\_memory(shmid));

strcpy(shared\_data, "Test Shared Memory");

std::cout << "Unit Test: Shared Memory Data: " << shared\_data << std::endl;

}

void test\_semaphore() {

key\_t key = ftok("progfile", 67);

int semid = init\_semaphore(key, 1);

if (semaphore\_wait(semid, 0) == 0) {

std::cout << "Unit Test: Semaphore locked." << std::endl;

}

if (semaphore\_signal(semid, 0) == 0) {

std::cout << "Unit Test: Semaphore unlocked." << std::endl;

}

}

void\* send\_concurrent\_messages(void\* arg) {

int msgid = ((int)arg);

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Concurrent Message");

for (int i = 0; i < 10; ++i) {

send\_message(msgid, &message);

std::cout << "Concurrency Test: Message " << i + 1 << " sent." << std::endl;

}

return nullptr;

}

void concurrency\_test\_message\_queue() {

key\_t key = ftok("progfile", 65);

int msgid = init\_message\_queue(key);

pthread\_t thread1, thread2;

pthread\_create(&thread1, nullptr, send\_concurrent\_messages, &msgid);

pthread\_create(&thread2, nullptr, send\_concurrent\_messages, &msgid);

pthread\_join(thread1, nullptr);

pthread\_join(thread2, nullptr);

struct msg\_buffer message;

for (int i = 0; i < 20; ++i) {

receive\_message(msgid, &message, 1);

std::cout << "Concurrency Test: Received message: " << message.msg\_text << std::endl;

}

}

void integration\_test\_ipc\_framework() {

key\_t key = ftok("progfile", 65);

int msgid = init\_message\_queue(key);

int shmid = init\_shared\_memory(key, 1024);

int semid = init\_semaphore(key, 1);

char\* shared\_data = static\_cast<char\*>(attach\_shared\_memory(shmid));

strcpy(shared\_data, "Integration Test Data");

struct msg\_buffer message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Integration Test Message");

send\_message(msgid, &message);

semaphore\_wait(semid, 0);

receive\_message(msgid, &message, 1);

std::cout << "Integration Test: Received message: " << message.msg\_text << std::endl;

std::cout << "Integration Test: Shared Memory Data: " << shared\_data << std::endl;

semaphore\_signal(semid, 0);

}

int main() {

std::cout << "Running Unit Tests:" << std::endl;

test\_message\_queue();

test\_shared\_memory();

test\_semaphore();

std::cout << "\nRunning Concurrency Test:" << std::endl;

concurrency\_test\_message\_queue();

std::cout << "\nRunning Integration Test:" << std::endl;

integration\_test\_ipc\_framework();

return 0;

}

# **10. Documentation:**

## **10.1 User Guide**

First, we need to create a directory named **“Capstone”.**

Then we need to create file named ipc\_mechanism.cpp

Write the code into that file.

Compile the code using g++ command.

Run the code to get the output.

## **10.2 Implementation**

Go to the root directory.

**command**: cd Capstone

rps@rps-virtual-machine:~/Capstone$

Create the file and enter the code.

**command: vim ipc\_mechanism.cpp**

compile the code using gcc.

**rps@rps-virtual-machine:~/Capstone$** g++ -o ipc\_mechanism ipc\_mechanism.cpp

Run the code to view the output.

**rps@rps-virtual-machine:~/Capstone$** ./ipc\_mechanism

Message sent successfully.

Received message: Hello, IPC Framework!

Semaphore acquired.

Shared Memory (after writing): Shared data example

Semaphore released.

# **11. Milestones:**

Milestones help track the progress of the project. Possible milestones include:

* **Milestone 1**: Initial design and setup of the project structure.
* **Milestone 2**: Implementation of message queue functionality.
* **Milestone 3**: Implementation of shared memory management.
* **Milestone 4**: Implementation of semaphore operations.
* **Milestone 5**: Completion of unit tests and basic functionality verification.
* **Milestone 6**: Concurrency and integration testing.
* **Milestone 7**: Final validation, performance testing, and documentation.

.

# **12.References:**

* [www.youtube.com](http://www.youtube.com)
* [www.github.com](http://www.github.com)
* For documentation we used readme.so
* For coding, we have taken references from online articles.