**Project Document: Inter-Process Communication (IPC) Framework**

**Contents**

[1. Project Overview](#_gjdgxs)…………………………………………………………………………………..1

[2 Objective](#_m5yye3sds0o2)…………………………………………………………………………………….2

[3.](#_30j0zll) [Scope](#_30j0zll) 3

[Target Environment](#_30bwfta88vdh)……………………………………………………………………………..3

[Primary Audience](#_p98e4l4d272t)……………………………………………………………………………….3

[Programming Language](#_d6lvhfyoe430)……………………………………………………………………….3

4. [Features](#_5bho3bx89d8g)…………………………………………………………………………………………….4

[Message Passing](#_ur0dv8spizab)………………………………………………………………………………4

[Synchronization](#_hsqqxhkqvvoi)………………………………………………………………………………...4

[Shared Memory](#_cv7yztrzebky)………………………………………………………………………………..4

[Process Creation](#_lqx9rvcws4g4)……………………………………………………………………………….4

[Inter-Process Signaling](#_qe4mw1xlkhy)………………………………………………………………………..4

[Pipes for Streamed Communication](#_g8aclhg21z0r)…………………………………………………………4

[Multi-Process Communication](#_dj8h2gcuizu8)………………………………………………………………..4

[Concurrency Management](#_e1z6qyj2a9tt)…………………………………………………………………..4

[Error Handling and Logging](#_s6q9inhj4s7i)…………………………………………………………………4

[Custom Protocol](#_luna3leoxx8m)……………………………………………………………………………….4

[Security Features](#_52rxu8zfdodx)………………………………………………………………………………4

[Timeout and Retransformation](#_l9cgdnerbh2i)……………………………………………………………….4

[Dynamic Configuration](#_hfhsh16pyew9)……………………………………………………………………….4

[Testing and Simulation Toots](#_7el1xzfk3fi0)………………………………………………………………...4

5[. Functional Requirements](#_1fob9te)…………………………………………………………………..5

[Message queue Communication](#_fo53qfqfmmaa)……………………………………………………5

[Semaphore-Based Synchronization](#_9x38zqqrsmsu)………………………………………………..5

[Shared Memory Access](#_a9tqaswl5q1s)………………………………………………………………..5

[Process Creation and Management](#_qt4284n8ieja)……………………………………………………5

[Inter-process Signaling](#_qe4mw1xlkhy)…………………………………………………………………5

[Pipes for Streamed Communication](#_g8aclhg21z0r)……………………………………………………5

[Multi-Process Communication](#_dj8h2gcuizu8)…………………………………………………………..5

[Concurrency Management](#_e1z6qyj2a9tt)………………………………………………………………5

[Error Handling](#_oxsb89f7zurg)…………….………………………………………………………………...5

[Custom Protocol](#_luna3leoxx8m)…………………………………………………………………………….5

[Security Mechanism](#_tf387tl4fxg)…………………………………………………………………………5

6. [Non-functional Requirements](#_d4jltpchf4r4) 6

[Performance](#_5fa1vqry5fob)………………………………………………………………………………...6

[Scalability](#_j603lpm5dsjp)……………………………………………………………………………………6

[Reliability](#_dz8njhuafm4g)…………………………………………………………………………………….6

[Usability](#_copvh2ob7xzf)……………………………………………………………………………………..6

[Security](#_2qkd7tfmskzd)……………………………………………………………………………………..6

[Maintainability](#_8eyqv5rummcq)………………………………………………………………………………6

[Compatibility](#_y7w0jw8g23az)………………………………………………………………………………..6

[Portability](#_97ejzivalrhx)……………………………………………………………………………………6

[Resource Efficiency](#_dg5xflcs4tlt)………………………………………………………………………..6

7[. User Roles](#_2et92p0)………………………………………………………………………………….7

[System Administrator](#_2rdefjk4dq2e)………………………………………………………………………7

[Regular User](#_rh179d6dh0ie)……………………………………………………………………………….7

8[. Input/Output Specifications](#_tyjcwt)…………………………………………………………………8

[Input](#_812um9slb5vn)…………………………………………………………………………………………8

[Output](#_iqqn57q9kxgc)……………………………………………………………………………………….8

9[. Challenges](#_3dy6vkm)………………………………………………………………………………….9

[Resource Management](#_pee822lh2y07)…………………………………………………………………...9

[Concurrency Control](#_9a4bwtdqbr4q)………………………………………………………………………9

[Error Handling](#_oxsb89f7zurg)……………………………………………………………………………..9

10.[System Design](#_e4w3c5js7uxw)…………………………………………………………………………………..10

[Process A](#_aesb424atces)…………………………………………………………………………………10

[Process B](#_jdckyjezd9fy)…………………………………………………………………………………10

11[.](#_1t3h5sf) [Skeleton Code Structure](#_1t3h5sf)…………………………………………………………………11

[ipc.h](#_knxxty2hv1gs)………………………………………………………………………………………..11

[ipc.cpp](#_4lqkjjvkeg7x)……………………………………………………………………………………..11

[ipc\_test.cpp](#_68zpft8hrkd0)……………………………………………………………………………….11

[Makefile](#_rjox2hrbda0c)……………………………………………………………………………………11

11[. Testing and Validation](#_4d34og8)……………………………………………………………………12

[Unit Testing](#_dv82txa0zulv)………………………………………………………………………..11

[Concurrency Testing](#_o81xv47wwpen)……………………………………………………………..11

[Integration Testing](#_6wsfr6x05n9v)………………………………………………………………11

[12. Documentation](#_2s8eyo1)…………………………………………………………………………..12

[Code Documentation](#_xgukvjdx8cs3)………………………………………………………………………..12

[User Guide](#_68ydoxxr1zx)……………………………………………………………………………………12

[13. Milestones](#_17dp8vu)………………………………………………………………………………..13

[Phase1](#_doz6rfwy4u5q)………………………………………………………………………………………..13

[Phase 2](#_9mzp6v7gx631)……………………………………………………………………………………….13

[Phase 3](#_zb00yqz9ledm)……………………………………………………………………………………….13

[Phase 4](#_gd1eltqlu5i0)……………………………………………………………………………………….13

[Phase 5](#_xd2viq98scq4)……………………………………………………………………………………….13

[Phase 6](#_bkdgspf4qcjv)……………………………………………………………………………………….13

[12. References](#_3rdcrjn)……………………………………………………………………………….14

[POSIX IPC Documentation](#_kbup3yzd28qf)…………………………………………………………………14

[Message Queues](#_e5im4vrbiv4d)……………………………………………………………………………14

[Shared Memory](#_cv7yztrzebky)………………………………………………………………………………14

[Semaphores](#_xnppmk90hbu6)………………………………………………………………………………….14

# **1. Project Overview**

**Project Title: Inter-Process Communication (IPC) Framework**

The **IPC (Inter-Process Communication) Framework** project is about creating tools that help different computer programs work together smoothly on a Linux system. These tools allow programs to share information and coordinate their actions, even if they are running on different computers.

**To achieve this, the project uses:**

* **Message Queues**: A way for one program to leave a message for another program to pick up later.
* **Shared Memory**: A common space where programs can read and write information directly.
* **Semaphores**: A system that ensures programs take turns using shared resources, avoiding conflicts.

# **2. Objective:**

**Enable Program Communication:** Allow different programs to talk to each other easily.

**Share Data Efficiently:** Let programs access and share information directly.

**Avoid Conflicts:** Ensure programs don’t interfere with each other when using shared resources.

**Be Flexible:** Create tools that can be used in various larger systems.

# **3. Scope**

### **Target Environment**

* **Linux-based operating systems.**

### **Primary Audience**

* System programmers, developers working on multi-process applications, and students learning about process communication.

### **Programming Language**

* C++

### **Libraries**

* POSIX IPC mechanisms (message queues, shared memory, semaphores)

# 

# **4. Features:**

### Message Passing:

* **Feature**: Enable communication between processes using message queues.
* **Description**: Processes can send and receive messages using a structured message queue system. Each message has a specific type, allowing selective reception of messages by type.

### Synchronization:

* **Feature**: Use of Semaphores.
* **Description**: Semaphores are used to synchronize access to shared resources between multiple processes. This prevents race conditions by controlling the order of execution.

### Shared Memory:

* **Feature**: Memory Sharing Between Processes.
* **Description**: Processes can communicate by sharing a block of memory, allowing them to exchange data quickly. Shared memory is typically faster than message queues or pipes for large data transfers.

### Process Creation:

* **Feature**: Forking New Processes.
* **Description**: The framework supports creating child processes using the fork() system call. The child processes can execute different parts of the application or handle separate tasks.Pipes for Streamed Communication:
* **Feature**: Unidirectional and Bidirectional Pipes.
* **Description**: The framework supports the use of pipes for streaming data between processes. Pipes can be used for one-way or two-way communication, enabling processes to exchange information in a producer-consumer model.

### Multi-Process Communication:

* **Feature**: Communication Across Multiple Processes.
* **Description**: The framework allows for communication not just between two processes but across multiple processes, using a combination of message queues, shared memory, and semaphores.

### Concurrency Management:

* **Feature**: Concurrent Processing.
* **Description**: Support for concurrent execution of processes, allowing multiple tasks to run simultaneously while managing shared resources.

### Testing and Simulation Tools:

* **Feature**: Built-in Testing and Simulation Tools.
* **Description**: Include tools to simulate different IPC scenarios, test the robustness of the system, and evaluate the performance of the communication mechanisms under various conditions.

# 

# **Functional Requirements**

### Message Queue Communication:

* + **Description**: The system should enable processes to communicate using message queues, allowing messages to be sent and received based on message types.
  + **Priority**: High

### Semaphore-Based Synchronization:

* + **Description**: The system must implement semaphores to control access to shared resources and ensure that processes are synchronized correctly.
  + **Priority**: High

### Shared Memory Access:

* + **Description**: Processes should be able to share a memory space for fast data exchange, with support for read and write operations.
  + **Priority**: High

### Process Creation and Management:

* + **Description**: The system should allow for the creation of new processes using fork(), manage their execution, and handle communication between parent and child processes.
  + **Priority**: High

### Inter-Process Signaling:

* + **Description**: The system should support signaling between processes, allowing processes to send and handle signals like SIGINT and SIGTERM.
  + **Priority**: Medium

### Pipe Communication:

* + **Description**: The system should support unidirectional and bidirectional pipes for streaming data between processes.
  + **Priority**: Medium

# **Non-Functional Requirements**

### Performance:

* + **Description**: The system should have low latency and high throughput for message passing, ensuring that communication between processes occurs quickly and efficiently.
  + **Priority**: High

### Scalability:

* + **Description**: The system should be able to handle a large number of processes and high volumes of data without significant performance degradation.
  + **Priority**: Medium

### Reliability:

* + **Description**: The IPC framework should be reliable, ensuring that messages are not lost, and processes are correctly synchronized even under heavy load.
  + **Priority**: High

### Usability:

* + **Description**: The framework should be easy to use, with clear documentation and examples, making it accessible for developers to implement IPC in their applications.
  + **Priority**: Medium

### Security:

* + **Description**: The framework should protect against unauthorized access to shared resources and ensure data integrity and confidentiality.
  + **Priority**: High

### Maintainability:

* + **Description**: The codebase should be well-documented and modular, allowing for easy updates and extensions to the system.
  + **Priority**: Medium

# **5. User Roles**

### System Administrator

* Full access to create, manage, and terminate IPC objects.

### Regular User

* Restricted access based on permissions, primarily focused on using the IPC mechanisms.

# **6. Input/Output Specifications**

### Input:

* Messages, data segments, and synchronization signals from various processes.

### Output:

* Messages delivered to intended recipients, synchronized access to shared resources.

# **7. Challenges**

### Resource Management:

* Efficiently managing and cleaning up IPC objects to prevent resource leakage.

### Concurrency Control:

* Implementing robust synchronization mechanisms to avoid race conditions and deadlocks.

### Error Handling:

* Providing clear error messages and recovery options in case of failures.

# 

# **8. System Design**

### Process A

* Send a message to **Message Queue**.
* Writes data to **Shared Memory**.
* Signals **Semaphore** after writing to **Shared Memory**.

**Process A**

**Message Queue**

**Shared Memory**

**Semaphore**

### **Process B**

* Waits for a signal from **Semaphore** before accessing **Shared Memory**.
* Reads data from **Shared Memory**.
* Receives a message from **Message Queue**

**Semaphore**

**Shared Memory**

**Message Queue**

**Process B**

### **Process A**

It is responsible for sending data to the Message Queue and writing data to Shared Memory. Once it has written to shared memory, it signals the Semaphore to notify Process B that the data is ready.

### **Process B**

**It** waits for the semaphore signal, ensuring that it only accesses the Shared Memory when it is safe to do so. After reading the data, Process B can also receive messages from the Message Queue.

# 

# 

# **9. Skeleton Code Structure**

### ipc.h

#ifndef IPC\_H // Check if IPC\_H is not defined

#define IPC\_H // Define IPC\_H

#include <iostream>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <cstring>

#include <unistd.h>

// define Message Structure

struct Message {

long msg\_type;

char msg\_text[100];

};

// Message Queue Functions

int initMessageQueue(key\_t key);

int sendMessage(int msgid, const Message& message);

int receiveMessage(int msgid, Message& message, long msg\_type);

// Shared Memory Functions

int initSharedMemory(key\_t key, size\_t size);

void\* attachSharedMemory(int shmid);

// Semaphore Functions-

int initSemaphore(key\_t key, int num\_sems);

int semaphoreWait(int semid, int sem\_num);

int semaphoreSignal(int semid, int sem\_num);

// IPC Framework Initialization

void initIPCFramework();

// Testing Functions

void test\_message\_queue();

void test\_shared\_memory();

void test\_semaphore();

void test\_concurrency();

void test\_integration();

#endif // IPC\_H

### ipc.cpp

#include <sys/wait.h>

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

// Message Queue Functions

int initMessageQueue(key\_t key) {

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

if (msgid == -1) {

perror("msgget");

exit(EXIT\_FAILURE);

}

return msgid;

}

int sendMessage(int msgid, const Message& message) {

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

perror("msgsnd");

return -1;

}

return 0;

}

int receiveMessage(int msgid, Message& 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 initSharedMemory(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\* attachSharedMemory(int shmid) {

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

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

perror("shmat");

exit(EXIT\_FAILURE);

}

return shmaddr;

}

// Semaphore Functions

int initSemaphore(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 to 1 (binary semaphore)

semctl(semid, 0, SETVAL, 1);

return semid;

}

int semaphoreWait(int semid, int sem\_num) {

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

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

perror("semop wait");

return -1;

}

return 0;

}

int semaphoreSignal(int semid, int sem\_num) {

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

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

perror("semop signal");

return -1;

}

return 0;

}

// IPC Framework Initialization

void initIPCFramework() {

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

int msgid = initMessageQueue(key);

int shmid = initSharedMemory(key, 1024);

int semid = initSemaphore(key, 1);

Message message;

message.msg\_type = 1;

strcpy(message.msg\_text, "Hello from Message Queue!");

sendMessage(msgid, message);

receiveMessage(msgid, message, 1);

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

char\* shared\_data = static\_cast<char\*>(attachSharedMemory(shmid));

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

semaphoreWait(semid, 0);

std::cout << "Shared Memory Content: " << shared\_data << std::endl;

semaphoreSignal(semid, 0);

}

// Testing Functions

void test\_message\_queue() {

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

int msgid = initMessageQueue(key);

Message message;

message.msg\_type = 1;

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

sendMessage(msgid, message);

receiveMessage(msgid, message, 1);

std::cout << "Message Queue Test - Received: " << message.msg\_text << std::endl;

}

void test\_shared\_memory() {

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

int shmid = initSharedMemory(key, 1024);

char\* shared\_data = static\_cast<char\*>(attachSharedMemory(shmid));

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

std::cout << "Shared Memory Test - Content: " << shared\_data << std::endl;

}

void test\_semaphore() {

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

int semid = initSemaphore(key, 1);

std::cout << "Waiting on semaphore." << std::endl;

if (semaphoreWait(semid, 0) == -1) {

std::cerr << "Semaphore wait failed." << std::endl;

return;

}

std::cout << "Semaphore acquired, entering critical section." << std::endl;

// Simulate some work in the critical section

sleep(1); // Replace with actual work

if (semaphoreSignal(semid, 0) == -1) {

std::cerr << "Semaphore signal failed." << std::endl;

return;

}

std::cout << "Semaphore released, exiting critical section." << std::endl;

}

void test\_concurrency() {

pid\_t pid = fork();

if (pid == 0) {

std::cout << "Child process starting tests." << std::endl;

test\_message\_queue();

test\_shared\_memory();

test\_semaphore();

std::cout << "Child process tests completed." << std::endl;

exit(0);

} else if (pid > 0) {

std::cout << "Parent process waiting for child." << std::endl;

test\_message\_queue();

test\_shared\_memory();

test\_semaphore();

wait(NULL);

std::cout << "Parent process tests completed." << std::endl;

} else {

perror("fork");

exit(EXIT\_FAILURE);

}

}

void test\_integration() {

std::cout << "Integration test starting." << std::endl;

initIPCFramework();

std::cout << "Integration test completed." << std::endl;

### **ipc\_test.cpp**

#include "../include/ipc.h"

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <sys/wait.h>

#include <unistd.h>

int main() {

test\_message\_queue();

test\_shared\_memory();

test\_semaphore();

test\_concurrency();

test\_integration();

return 0;

}

### **Makefile**

**# Default target**

all: $(TARGET)

**# Rule to build the executable**

$(TARGET): $(OBJ\_FILES)

$(CC) $(CFLAGS) -o $@ $^

**# Rule to build object files**

$(BUILD\_DIR)/%.o: $(SRC\_DIR)/%.cpp

mkdir -p $(BUILD\_DIR)

$(CC) $(CFLAGS) -c $< -o $@

**# Clean rule to remove build artifacts**

clean:

rm -rf $(BUILD\_DIR) $(TARGET)

### **OUTPUT**

### 

rm -rf $(BUILD\_DIR) $(TARGET)

# **9. Testing and Validation**

### Unit Testing

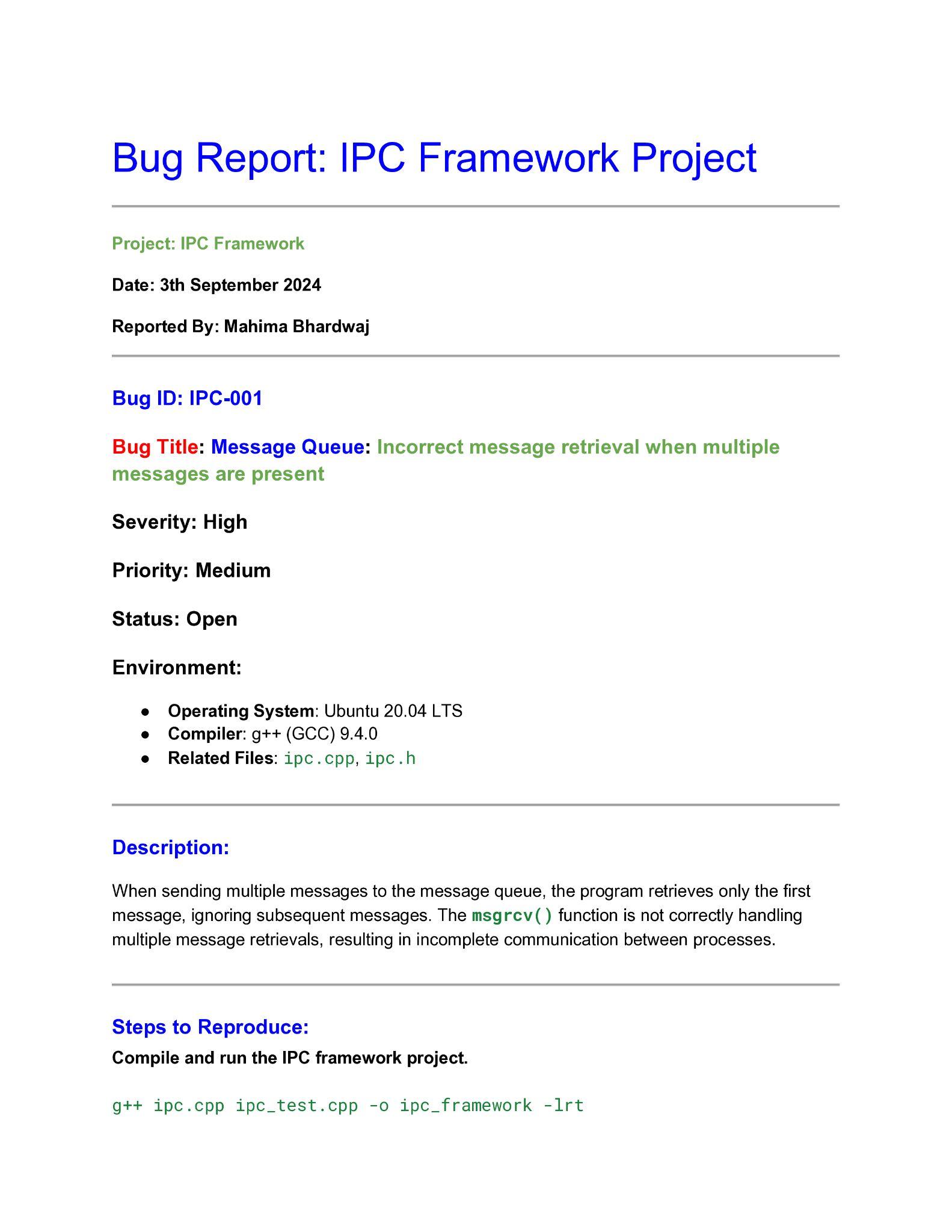
* Test individual IPC mechanisms (message queues, shared memory, semaphores) to ensure correctness.

### Concurrency Testing

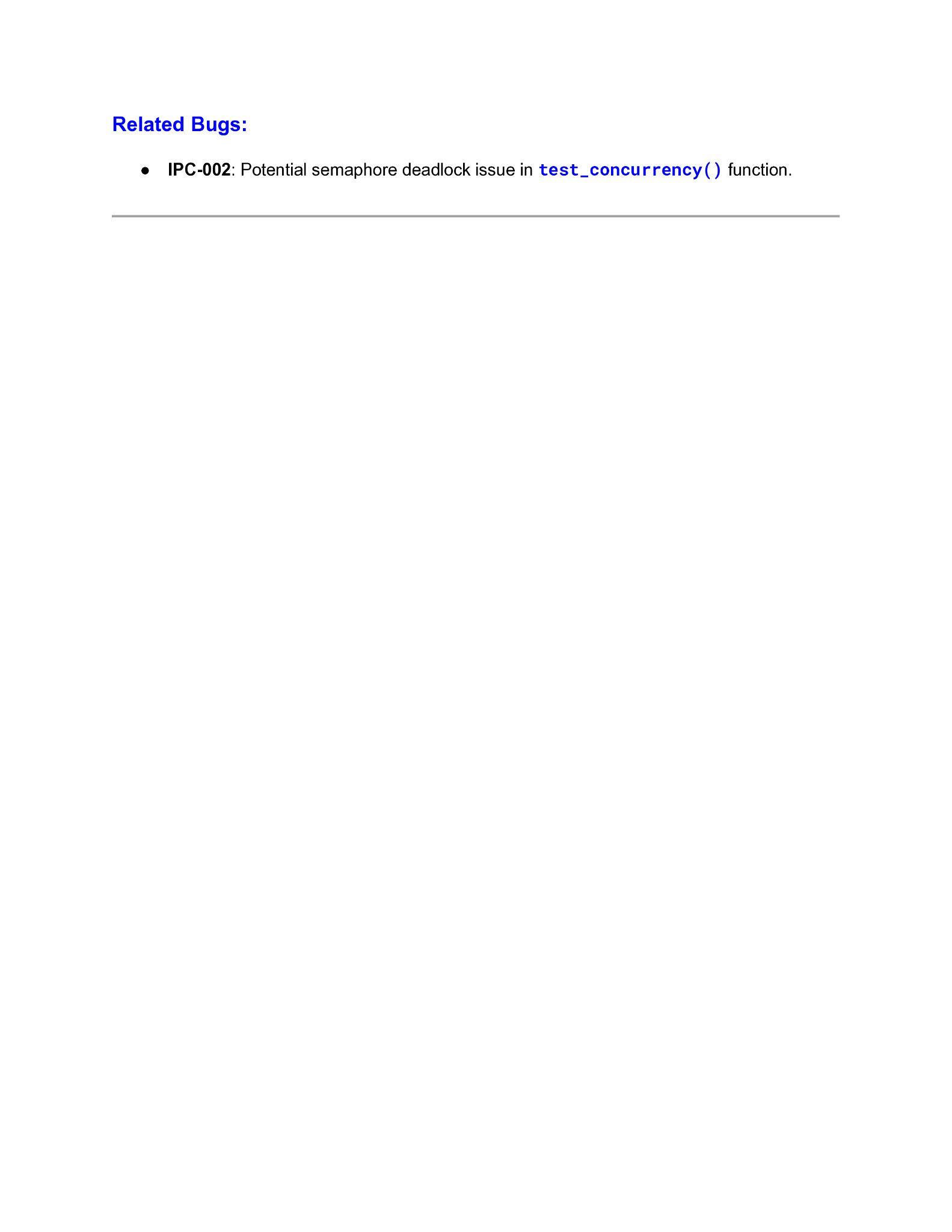
* Simulate multiple processes interacting with the framework to ensure synchronization and data integrity.

### Integration Testing

* Verify the framework works seamlessly as a unified IPC solution.

****





# **10. Documentation**

### Code Documentation:

Use inline comments and external documentation (e.g., Doxygen) to explain the purpose and usage of each module.

### User Guide:

Provide comprehensive instructions for developers on how to integrate and use the IPC framework in their applications.

# **11. Milestones**

### Phase 1

* Research and setup of the development environment.

### Phase 2

* Implementation of message queues.

### Phase 3

* Implementation of shared memory.

### Phase 4

* Implementation of semaphores.

### Phase 5

* Integration of IPC mechanisms into a unified framework.

### Phase 6

* Testing, debugging, and documentation.

# **12. References**

### POSIX IPC Documentation

* https://man7.org/linux/man-pages/man7/ipc.7.html

### Message Queues

* https://man7.org/linux/man-pages/man2/msgget.2.html

### Shared Memory

* https://man7.org/linux/man-pages/man2/shmget.2.html

### Semaphores

* <https://man7.org/linux/man-pages/man2/semget.2.html>

**This document outlines the requirements and provides a foundation for implementing an Inter-Process Communication (IPC) framework. The provided skeleton code offers a basic structure, which can be expanded to meet the full scope of the project.**

**THANK YOU**

* **WIPRO TEAM**
* **RPS TEAM**
* **SHWEATK SIR**
* **KIRAN VVN SIR**