Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12

Project Document: Inter-Process Communication (IPC) Framework

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12

Contents

Copyright © Mahima Bhardwaj

bhardwamahima81@gmail.com

Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Copyright © Mahima Bhardwaj

bhardwamahima81@gmail.com

Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12

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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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.

Inter-Process Signaling:

- **Feature**: Signal Handling.
- **Description**: Processes can send signals to each other to trigger specific actions. The framework includes custom signal handlers to manage signals such as `SIGINT`, `SIGTERM`, etc.

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.

Error Handling and Logging:

- **Feature**: Robust Error Handling and Logging Mechanisms.
- **Description**: The framework includes comprehensive error handling for IPC operations, with logs that track failures, retries, and the status of communication channels.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Custom Protocols:

- **Feature**: Design of Custom Protocols for IPC.
- **Description**: Users can design and implement custom protocols to define how messages are formatted, transmitted, and interpreted between processes.

Security Features:

- **Feature**: Access Control for Shared Resources.
- **Description**: Implement access control mechanisms to restrict which processes can read from or write to shared memory or message queues.

Timeout and Retransmission:

- **Feature**: Support for Timeouts and Retransmission.
- **Description**: Implement timeout mechanisms for IPC operations and automatic retransmission of messages if no acknowledgment is received within a specified period.

Dynamic Configuration:

- **Feature**: Runtime Configuration of IPC Parameters.
- **Description**: The framework allows dynamic configuration of parameters like message queue size, shared memory size, and semaphore values, enabling the system to adapt to changing workload requirements.

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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Functional Requirements

1. 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
2. Semaphore-Based Synchronization:
    - **Description**: The system must implement semaphores to control access to shared resources and ensure that processes are synchronized correctly.
    - **Priority**: High
3. 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
4. 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
5. Inter-Process Signaling:
    - **Description**: The system should support signaling between processes, allowing processes to send and handle signals like `SIGINT` and `SIGTERM`.
    - **Priority**: Medium
6. Pipe Communication:
    - **Description**: The system should support unidirectional and bidirectional pipes for streaming data between processes.
    - **Priority**: Medium
7. Error Handling:
    - **Description**: The system must include robust error handling mechanisms to handle IPC failures, including retries and logging.
    - **Priority**: High
8. Custom Protocols Support:
    - **Description**: Users should be able to define and use custom communication protocols for their specific needs.
    - **Priority**: Medium

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

9. Security Mechanisms:
    - **Description**: Implement security features to control access to shared resources and ensure that only authorized processes can communicate.
    - **Priority**: Medium

Non-Functional Requirements

1. 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
2. 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
3. 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
4. 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
5. Security:
    - **Description**: The framework should protect against unauthorized access to shared resources and ensure data integrity and confidentiality.
    - **Priority**: High
6. Maintainability:
    - **Description**: The codebase should be well-documented and modular, allowing for easy updates and extensions to the system.
    - **Priority**: Medium
7. Compatibility:
    - **Description**: The system should be compatible with different operating systems and environments, ensuring that it can be deployed in various contexts.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

  - **Priority**: Low
8. Portability:
  - **Description**: The IPC framework should be portable across different hardware and software platforms, allowing it to be used in a wide range of applications.
  - **Priority**: Low
9. Resource Efficiency:
  - **Description**: The system should minimize the use of system resources such as memory and CPU, ensuring that it runs efficiently even on resource-constrained devices.
  - **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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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.

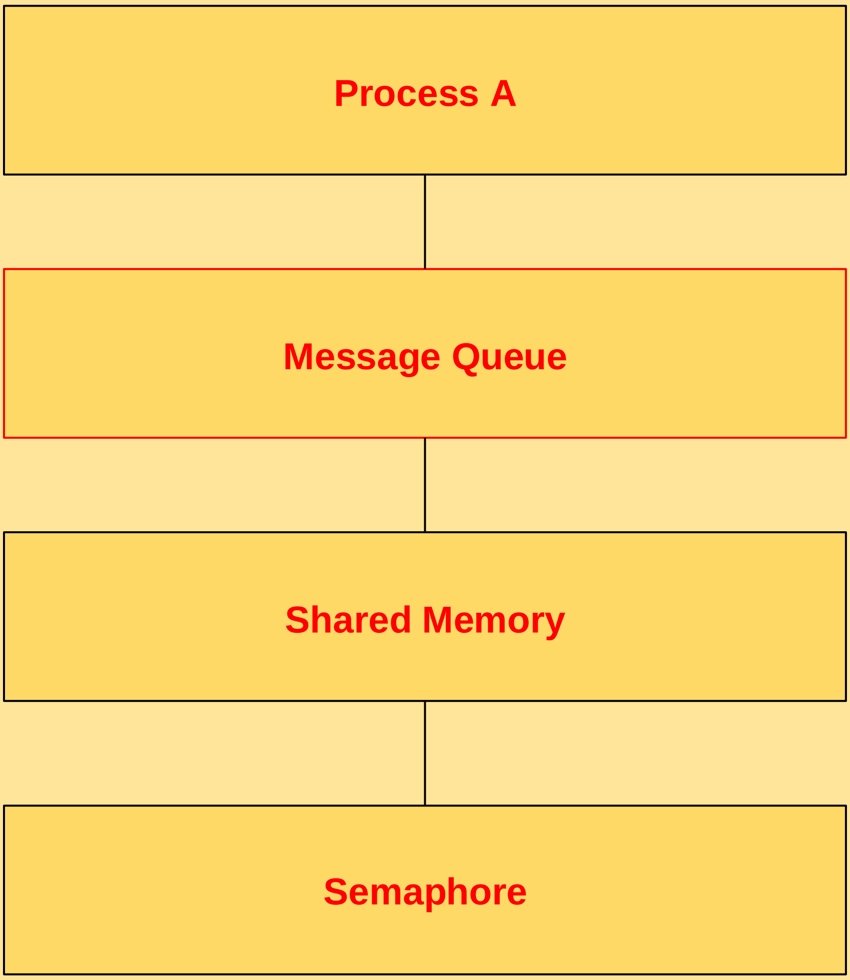
8. System Design

Process A

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

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12

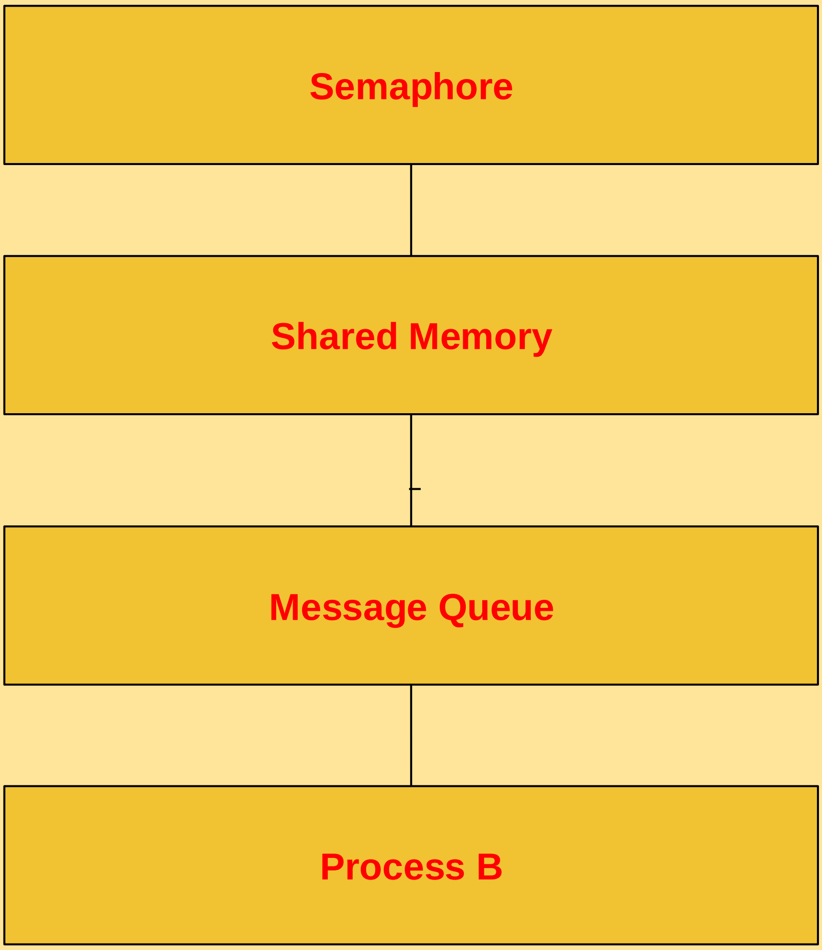


Process B

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

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

Docid	Language	Author	Subject	Page
Proj/Wip	En	Mahima	LSP	12



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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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);
```


Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
// 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 "../include/ipc.h"
#include <sys/wait.h> //wait()
#include <iostream>
#include <cstring> // strcpy, strlen
#include <cstdlib> //exit(), malloc()

// Message Queue Functions
int initMessageQueue(key_t key) {
```

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    int msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    return msgid;
}

// to send msg into queue
int sendMessage(int msgid, const Message& message) {
    if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) == -1) {
        perror("msgsnd");
        return -1;
    }
    return 0;
}

// to receive msg from queue
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) {
```

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    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);
    }
    return semid;
}

int semaphoreWait(int semid, int sem_num) {
```


Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    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);
```

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    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);
```

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    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);

    semaphoreWait(semid, 0);
    std::cout << "Semaphore Test - Entered Critical Section" << std::endl;
    semaphoreSignal(semid, 0);
}

void test_concurrency() {
    pid_t pid = fork();
    if (pid == 0) {
        test_message_queue();
        test_shared_memory();
        test_semaphore();
        exit(0);
    } else {
        test_message_queue();
        test_shared_memory();
        test_semaphore();
        wait(NULL);
```


Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

```
    }
}

void test_integration() {
    initIPCFramework();
}
```

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;
}
```

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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.

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com

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.

Copyright © Mahima Bhardwaj

bhardwamahima81@gmail.com

THANK YOU

- WIPRO TEAM
- RPS TEAM
- SHWEATK SIR
- KIRAN VVN SIR

k

Copyright © Mahima Bhardwaj
bhardwamahima81@gmail.com