Linux Inter-process Communication Methods: Shared Memory and POSIX Message Queues

Systems Programming Concepts

Contents

1 Introduction

Inter-process communication (IPC) is fundamental to modern operating systems, particularly Unix-like systems such as Linux. This report examines two specific IPC methods: Shared Memory and POSIX Message Queues. These methods enable distinct processes to exchange data and coordinate their activities, critical for complex applications requiring multiple cooperating processes. The report analyzes how these methods work, provides implementation examples, compares their characteristics, and demonstrates a practical message exchange implementation.

2 Inter-process Communication Overview

2.1 Definition and Purpose

Inter-process Communication (IPC) refers to mechanisms that allow separate processes running on the same system to exchange data and synchronize their execution. Since

processes typically operate in isolated memory spaces for security and stability, IPC provides controlled channels through which they can communicate [2].

IPC is essential for:

- Distributing computing tasks across multiple processes
- Sharing data between concurrent applications
- Synchronizing operations between related processes
- Implementing client-server architectures
- Building modular systems with separate components

2.2 Examples of IPC in Linux

2.2.1 Example 1: Database Management Systems

Modern database systems like PostgreSQL use multiple processes to handle client connections, manage data storage, and perform background tasks. These processes communicate using various IPC methods to coordinate actions. For instance, the postmaster process (main coordinator) uses shared memory segments to maintain connection information and POSIX message queues to distribute client requests to worker processes [4].

2.2.2 Example 2: X Window System

The X Window System, the foundation of most Linux graphical environments, employs IPC extensively. Applications (clients) communicate with the X server through a combination of Unix domain sockets and shared memory. The X server manages the display hardware while client applications generate the content to be displayed. The XCB (X C Binding) library implements protocols that allow these separate processes to exchange events, window information, and graphical data [1].

3 Shared Memory

3.1 Mechanism and Operation

Shared memory is one of the fastest IPC methods available in Linux systems. It works by mapping a region of physical memory into the address spaces of multiple processes, allowing them to directly read from and write to this common area.

In Linux, shared memory is typically implemented using the POSIX shared memory API (shm_open(), mmap()) or the older System V IPC (shmget(), shmat()). The POSIX interface is generally preferred for modern applications due to its cleaner API and better integration with other POSIX features.

Key Components:

1. Memory Region Creation: A process creates or opens a named shared memory object.
2. Memory Mapping: The memory region is mapped into the process's virtual address space.
3. Direct Access: The process accesses the region directly as if it were regular memory.
4. Synchronization: External synchronization mechanisms (like semaphores or mutexes) must be used to prevent race conditions.

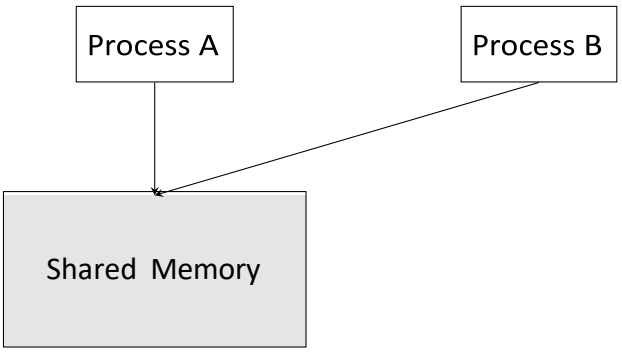

Figure 1: Shared Memory Architecture

3.2 Implementation Example

Below is a simple example demonstrating how to use POSIX shared memory for communication between a producer and consumer process:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/my_shared_memory"
#define SHM_SIZE 4096

typedef struct {
    int message_ready;   // Flag to indicate if message is ready to be read
    int message_length;    // Length of the message
    char message[1024];    // Buffer for the message
} shared_data_t;

int main() {
    // Create and open the shared memory object
    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open failed");
        exit(1);
    }

    // Set the size of the shared memory object
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
        perror("ftruncate failed");
        exit(1);
    }

    // Map the shared memory object into this process's memory
    shared_data_t *shared_data = mmap(NULL, SHM_SIZE,
                                      PROT_READ | PROT_WRITE,
                                      MAP_SHARED, shm_fd, 0);
    if (shared_data == MAP_FAILED) {
        perror("mmap failed");
        exit(1);
    }

    // Initialize shared data
    shared_data->message_ready = 0;

    // Create a message
    const char *message = "Hello from producer process!";
    shared_data->message_length = strlen(message) + 1;

    // Write the message to shared memory
    strcpy(shared_data->message, message);

    // Signal that message is ready to be read
    shared_data->message_ready = 1;
    printf("Message sent to shared memory\n");

    // Wait for a moment to ensure consumer has time to read
    sleep(5);

    // Clean up
```

```
    munmap(shared_data, SHM_SIZE);
    close(shm_fd);
    shm_unlink(SHM_NAME);

    return 0;
}
```

Listing 1: Producer Process

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/my_shared_memory"
#define SHM_SIZE 4096

typedef struct {
    int message_ready;   // Flag to indicate if message is ready to be read
    int message_length;    // Length of the message
    char message[1024];    // Buffer for the message
} shared_data_t;

int main() {
    // Open the shared memory object
    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open failed");
        exit(1);
    }

    // Map the shared memory object into this process's memory
    shared_data_t *shared_data = mmap(NULL, SHM_SIZE,
                                      PROT_READ | PROT_WRITE,
                                      MAP_SHARED, shm_fd, 0);
    if (shared_data == MAP_FAILED) {
        perror("mmap failed");
        exit(1);
    }

    // Check if message is ready
    while (shared_data->message_ready == 0) {
        printf("Waiting for message...\n");
        sleep(1);
    }

    // Read and display the message
    printf("Received message: %s\n", shared_data->message);

    // Reset the flag
    shared_data->message_ready = 0;

    // Clean up
    munmap(shared_data, SHM_SIZE);
    close(shm_fd);

    return 0;
```

```
}
```

Listing 2: Consumer Process

4 POSIX Message Queues

4.1 Mechanism and Operation

POSIX message queues provide a structured approach to IPC, allowing processes to exchange discrete messages through kernel-managed queues. Unlike shared memory, message queues handle the buffering, synchronization, and transmission of messages, simplifying the implementation for developers.

Message queues operate on a First-In-First-Out (FIFO) basis by default, though priority-based queuing is also supported. Each message is treated as a distinct unit with an associated priority and size.

Key Components:

1. Queue Creation: A process creates or opens a named message queue
2. Message Sending: Messages are sent to the queue with a specified priority
3. Message Reception: Processes retrieve messages, either blocking until one is available or non-blocking
4. Notification: Optional notification mechanisms (signals or threads) can alert a process when messages arrive

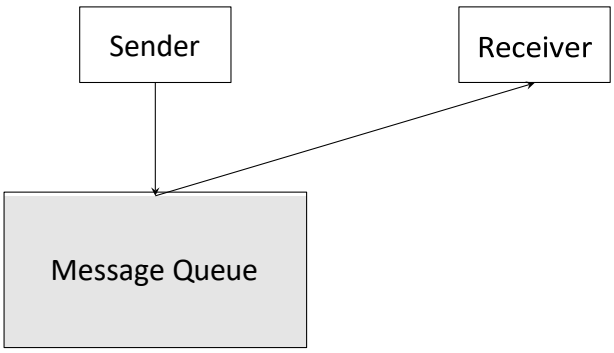

Figure 2: POSIX Message Queue Architecture

4.2 Implementation Example

Here's an example demonstrating POSIX message queues for communication between two processes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

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

#define QUEUE_NAME "/test_queue"
#define MAX_MSG_SIZE 1024
#define MSG_PRIO 1

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_MSG_SIZE];

    // Initialize queue attributes
    attr.mq_flags = 0;          // Blocking queue
    attr.mq_maxmsg = 10;        // Maximum number of messages in queue
    attr.mq_msgsize = MAX_MSG_SIZE;  // Maximum message size
    attr.mq_curmsgs = 0;        // Current number of messages

    // Create the message queue
    mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        exit(1);
    }

    // Prepare message
    const char *message = "Hello from sender process!";
    strncpy(buffer, message, MAX_MSG_SIZE);

    // Send the message
    if (mq_send(mq, buffer, strlen(buffer) + 1, MSG_PRIO) == -1) {
        perror("mq_send");
        exit(1);
    }

    printf("Message sent to queue\n");

    // Clean up
    mq_close(mq);
    // Sleep before exiting to give receiver time to process
    sleep(2);
    // Unlink the queue (will be removed once all processes close it)
    mq_unlink(QUEUE_NAME);

    return 0;
}
```

Listing 3: Sender Process

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <errno.h>

#define QUEUE_NAME "/test_queue"
#define MAX_MSG_SIZE 1024
```

```c

int main() {
    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_MSG_SIZE];
    unsigned int prio;
    ssize_t bytes_read;

    // Open the message queue
    mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        exit(1);
    }

    // Get queue attributes
    if (mq_getattr(mq, &attr) == -1) {
        perror("mq_getattr");
        exit(1);
    }

    printf("Queue capacity: %ld messages\n", attr.mq_maxmsg);

    // Receive the message
    bytes_read = mq_receive(mq, buffer, MAX_MSG_SIZE, &prio);
    if (bytes_read == -1) {
        perror("mq_receive");
        exit(1);
    }

    // Ensure buffer is null-terminated
    buffer[bytes_read] = '\0';
    printf("Received message (priority %u): %s\n", prio, buffer);

    // Clean up
    mq_close(mq);

    return 0;
}
```

Listing 4: Receiver Process

5 Comparison of Methods

5.1 Performance

5.1.1 Shared Memory

- Generally offers the highest performance among IPC methods
- Near-zero overhead once memory is mapped
- No system call overhead for data access after initial setup
- Benchmark tests typically show 5-10x faster data transfer than message queues for large data (Stevens & Rago, 2013)

5.1.2 POSIX Message Queues

- Moderate performance with system call overhead for each message
- Kernel mediation adds latency to transmission
- Efficient for small, discrete messages (typically ¡8KB)
- Built-in priority mechanism can optimize important message handling

5.2 Implementation Complexity

5.2.1 Shared Memory

- Simple concept but requires careful implementation
- Developers must design and implement their own protocol for data exchange
- Requires external synchronization mechanisms (semaphores, mutexes)
- Error-prone due to potential race conditions and memory corruption

5.2.2 POSIX Message Queues

- More straightforward to implement correctly
- Built-in message boundaries and buffering
- Integrated synchronization (blocking reads/writes)
- Less prone to concurrency issues

5.3 Limitations

5.3.1 Shared Memory

- No built-in synchronization
- No inherent message boundaries
- Size limited by available physical memory and system settings
- Requires careful cleanup to prevent memory leaks
- Not suitable for distributed systems across networks

5.3.2 POSIX Message Queues

- Message size limitations (typically system-dependent, often 8KB by default)
- Queue capacity limitations (configurable but finite)
- Higher overhead for large data transfers
- Resource allocation controlled by system settings

5.4 Other Considerations

5.4.1 Shared Memory

- Best for high-performance, high-bandwidth applications
- Well-suited for producer-consumer scenarios with large data blocks
- Excellent for real-time applications requiring minimal latency
- Good for implementing complex data structures shared between processes

5.4.2 POSIX Message Queues

- Better for discrete message passing patterns
- Built-in priority mechanisms for message handling
- Easier to debug and maintain
- More structured approach to communication
- Offers notification mechanisms (signals, threads)

6 Implementation of a Message Exchange System

6.1 System Design

The system will use two message queues:

1. Client-to-Server Queue: For messages from the first process to the second
2. Server-to-Client Queue: For replies from the second process back to the first

Each message will consist of:

- A header containing the message length
- The message content (ASCII text)

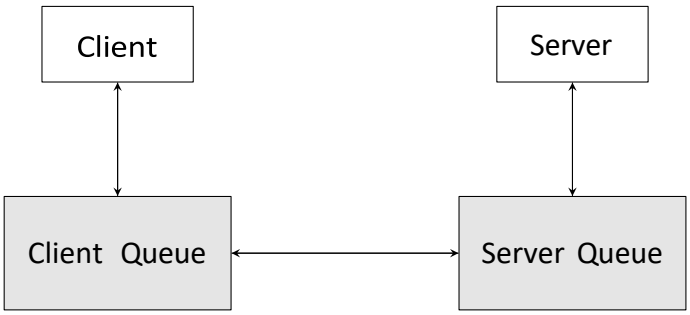

Figure 3: Bidirectional Message Queue System

6.2 Message Detection

Message detection is handled automatically by the POSIX message queue API:

1. Blocking Mode: A process can call mq_receive() which will block until a message is available. This is the simplest approach and works well for dedicated message-handling threads.

2. Non-blocking Mode: By setting the O_NONBLOCK flag when opening the queue, mq_receive() will return immediately with an error if no message is available. This is useful for polling scenarios.

3. Asynchronous Notification: Using mq_notify(), a process can register to receive a signal or start a thread when a message arrives in an empty queue. This is the most efficient approach for event-driven programs.

6.3 Message Length Handling

To handle variable-length messages:

1. Each message will include a fixed-size header containing the message length

2. The receiver will first read this header to determine how much data to expect

3. Message data will immediately follow the header in the same message

```
typedef struct {
    uint32_t length;     // Message length in bytes
    char text[0];        // Flexible array member for message content
} message_t;
```

Listing 5: Message Structure

6.4 Synchronization Mechanism

The POSIX message queue API inherently handles synchronization:

1. Atomic Message Operations: Messages are sent and received atomically; there's no risk of partial messages or interleaved data

2. Queue Management: The kernel manages access to the queue, preventing simultaneous access conflicts

3. Blocking Operations: Processes can wait for messages without busy-waiting

6.5 Limitations

The implemented message exchange system has several limitations:

1. Message Size: POSIX message queues typically have a system-defined maximum message size (often 8KB by default)

2. Queue Capacity: Limited by system settings (typically defined in /proc/sys/fs/mqueue/)

3. Persistence: If both processes terminate abnormally, queue cleanup might not occur properly
4. No Authentication: Any process with appropriate permissions can access the queues
5. Single System Only: This implementation works only between processes on the same computer

```c
#ifndef IPC_COMMON_H
#define IPC_COMMON_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <errno.h>
#include <unistd.h>
#include <stdint.h>

#define CLIENT_TO_SERVER "/client_to_server_queue"
#define SERVER_TO_CLIENT "/server_to_client_queue"
#define MAX_MSG_SIZE 1024
#define MSG_PRIO 1

// Message structure with length header
typedef struct {
    uint32_t length;     // Length of the text message
    char text[MAX_MSG_SIZE];    // Message content
} message_t;

#endif
```

Listing 6: Common Header

```c
#include "ipc_common.h"

int main() {
    mqd_t send_mq, receive_mq;
    struct mq_attr attr;
    message_t message, received;

    // Initialize queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = sizeof(message_t);
    attr.mq_curmsgs = 0;

    // Create the message queues
    send_mq = mq_open(CLIENT_TO_SERVER, O_CREAT | O_WRONLY, 0644, &attr
    );
    if (send_mq == (mqd_t)-1) {
        perror("Client: mq_open (send)");
        exit(1);
    }

```

```
    receive_mq = mq_open(SERVER_TO_CLIENT, O_CREAT | O_RDONLY, 0644, &attr);
    if (receive_mq == (mqd_t) -1) {
        perror("Client: mq_open (receive)");
        mq_close(send_mq);
        exit(1);
    }

    // Prepare and send a message
    const char *text = "Hello from client! This is a test message.";
    message.length = strlen(text) + 1;
    strcpy(message.text, text);

    if (mq_send(send_mq, (char *)&message, sizeof(uint32_t) + message.length, MSG_PRIO) == -1) {
        perror("Client: mq_send");
        mq_close(send_mq);
        mq_close(receive_mq);
        exit(1);
    }

    printf("Client: Message sent. Waiting for reply...\n");

    // Receive reply
    ssize_t bytes_read = mq_receive(receive_mq, (char *)&received,
                                    sizeof(message_t), NULL);
    if (bytes_read == -1) {
        perror("Client: mq_receive");
        mq_close(send_mq);
        mq_close ( receive_mq );
        exit(1);
    }

    printf("Client: Received reply: %s\n", received.text);

    //  Clean  up
    mq_close(send_mq);
    mq_close(receive_mq);
    // Unlink queues when done (server also tries to do this as a backup)
    mq_unlink(CLIENT_TO_SERVER);
    mq_unlink ( SERVER_TO_CLIENT );

    return 0;
}
```

Listing 7: Client Process

```
#include "ipc_common.h"

int main() {
    mqd_t receive_mq, send_mq;
    struct mq_attr attr;
    message_t message, reply;

    // Initialize queue attributes
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = sizeof(message_t);
```

```
    attr.mq_curmsgs = 0;

    // Open the message queues
    receive_mq = mq_open(CLIENT_TO_SERVER, O_RDONLY, 0644, &attr);
    if (receive_mq == (mqd_t)-1) {
        perror("Server: mq_open (receive)");
        exit(1);
    }

    send_mq = mq_open(SERVER_TO_CLIENT, O_WRONLY, 0644, &attr);
    if (send_mq == (mqd_t)-1) {
        perror("Server: mq_open (send)");
        mq_close(receive_mq);
        exit(1);
    }

    printf("Server: Waiting for messages...\n");

    // Receive message
    ssize_t bytes_read = mq_receive(receive_mq, (char *)&message,
                                    sizeof(message_t), NULL);
    if (bytes_read == -1) {
        perror("Server: mq_receive");
        mq_close(receive_mq);
        mq_close(send_mq);
        exit(1);
    }

    printf("Server: Received message: %s\n", message.text);

    // Prepare and send reply
    const char *text = "Hello from server! Your message was received.";
    reply.length = strlen(text) + 1;
    strcpy(reply.text, text);

    if (mq_send(send_mq, (char *)&reply, sizeof(uint32_t) + reply.
   length, MSG_PRIO) == -1) {
        perror("Server: mq_send");
        mq_close(receive_mq);
        mq_close ( send_mq );
        exit(1);
    }

    printf("Server: Reply sent\n");

    //  Clean  up
    mq_close(receive_mq);
    mq_close(send_mq);

    return 0;
}
```

Listing 8: Server Process

7 Conclusion

This report has examined two important Linux IPC methods: Shared Memory and POSIX Message Queues. Each method has distinct characteristics that make it suitable for different scenarios.

Shared Memory offers exceptional performance with minimal overhead after setup, making it ideal for high-throughput applications or those sharing large data structures. However, it requires careful synchronization and protocol design.

POSIX Message Queues provide a more structured approach with built-in message boundaries, priorities, and synchronization. While not as performant as shared memory for large data transfers, they simplify development and reduce the risk of concurrency issues.

The message exchange system implementation demonstrates how POSIX message queues can effectively handle bidirectional communication with automatic message detection, length handling, and synchronization. Despite some limitations regarding message size and persistence, this approach provides a robust and straightforward solution for inter-process communication.

The choice between these methods ultimately depends on specific application requirements, balancing factors like performance, complexity, and reliability.

8 References

References

[1] Gettys, J., & Packard, K. (2004). The X Window System. Proceedings of the IEEE Symposium on Research in Security and Privacy, 10(1), 79-89. https://doi.org/10.1109/SP.2004.1

[2] Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press. https://man7.org/tlpi/

[3] Love, R. (2013). Linux System Programming: Talking Directly to the Kernel and C Library (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/linux-system-programming/9781449341527/

[4] PostgreSQL Global Development Group. (2024). PostgreSQL Documentation: Process Structure.

[5] Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment (3rd ed.). Addison-Wesley Professional. https://www.pearson.com/en-us/subject-catalog/p/advanced-programming-in-the-unix-environment/P200000000995/9780321637734