

# CPE 345: Operating Systems Laboratory

## Lab05

### IPC Message Queues

**Submitted by:** Gianna Foti

**Date of Experiment:** \_\_\_\_\_ 10 February 2025 \_\_\_\_\_

**Report Deadline:** \_\_\_\_\_ 12 February 2025 \_\_\_\_\_

**Lab Section:** \_\_\_\_\_ CPE345-03 \_\_\_\_\_

**Demonstration Deadline:** \_\_\_\_\_ 12 February 2025 \_\_\_\_\_

## Introduction

In this lab, we explored Inter-Process Communication (IPC) using message queues to facilitate asynchronous communication between processes. We implemented two processes: Process A, which creates a message queue, sends messages, and waits for responses, and Process B, which receives messages and responds accordingly. Key system calls used included `msgget()` to create/access a queue, `msgsnd()` to send messages, `msgrcv()` to receive messages, and `msgctl()` to manage and remove the queue. Proper synchronization and cleanup were emphasized to prevent resource leaks. This lab reinforced our understanding of IPC mechanisms in Linux and the importance of efficient resource management in concurrent systems.

## Theory Topics

1. `msgget()`
  - a. The `msgget()` system call is used to create or access a message queue. It takes a key and flags as arguments, where `IPC_CREAT` is used to create the queue if it does not exist. The function returns a message queue identifier (`msqid`), which is used for further operations like sending and receiving messages.
2. `msgsnd()`
  - a. The `msgsnd()` function is responsible for sending messages to a message queue. It requires a message queue ID, a pointer to the message structure, the size of the message content, and optional flags. The message must have a message type (`msg_type`), which helps in retrieving specific messages using `msgrcv()`. It allows processes to communicate by sending structured messages through a kernel-managed queue.

### 3. `msgrcv()`

- a. The `msgrcv()` function retrieves messages from a message queue. It allows processes to read messages of a specified type or receive any available message. The function blocks execution until a message is received unless the `IPC_NOWAIT` flag is set, in which case it returns immediately if no message is available.

### 4. `msgctl()`

- a. The `msgctl()` function performs various control operations on a message queue, including retrieving information, modifying attributes, and setting permissions. It is essential for managing the lifecycle of a message queue, ensuring that system resources are properly allocated and released. Proper use of `msgctl()` prevents message queues from persisting unnecessarily, reducing the risk of resource leaks in the system.

## **Part 1 (Writer):**

**Program Description:** This program is responsible for creating and managing the message queue used for communication between two processes. It prompts the user for input, sends the message to the queue, and waits for a response from Process B. The process continues exchanging messages until the user types “Exit”, at which point it sends the termination command, waits for acknowledgment from Process B, and then properly deallocates the message queue using `msgctl()`. This ensures that system resources are freed, preventing lingering queues. Process A plays a crucial role in initiating and managing inter-process communication (IPC) while ensuring clean termination.

Process:

1. I defined a struct `msg_buffer` with two fields:
  - a. `long msg_type` to differentiate between sent and received messages.
  - b. `char msg_text[MSG_SIZE]` to store the actual message content.
2. I created a message queue using `msgget(KEY, IPC_CREAT | 0666)` to establish communication between Process A and Process B.

3. I implemented message sending by prompting the user for input, removing the newline character, setting `msg_type = 1`, and using `msgsnd()` to send messages to Process B.
4. I handled message reception using `msgrcv()`, waiting for and displaying responses from Process B while ensuring continuous message exchange.
5. I managed exit conditions by checking if the user typed "Exit", sending it to Process B, waiting for acknowledgment, and ensuring both processes terminated properly.
6. I cleaned up resources by calling `msgctl(msgid, IPC_RMID, NULL)`, removing the message queue before exiting to prevent system resource leaks.

### Program Output:

```
▼ TERMINAL
● bash-4.4$ gcc lab005_P1.c -o lab005_P1
● bash-4.4$ ./lab005_P1
Process A: Chat Started. Type 'Exit' to quit.

You: Hey!
=====
Message from Process B: 238947 Hello!!?? >>
=====

You: Im going to exit next
=====
Message from Process B: do it man
=====

You: Exit
Exit command sent. Waiting for acknowledgment...
=====
Exit command received. Ending chat.
=====
Message queue removed. Chat Ended.
○ bash-4.4$ █

● bash-4.4$ gcc lab005_P2.c -o lab005_P2
● bash-4.4$ ./lab005_P2
Process B: Chat Started. Waiting for messages...

=====
Message from Process A: Hey!
=====
You: 238947 Hello!!?? >>
=====

Message from Process A: Im going to exit next
=====
You: do it man
=====

Message from Process A: Exit
=====

Process A exit command received. Ending chat...
Process B: Chat Ended.
○ bash-4.4$ █
```

Figure 1: Program 1 with Writer Exit

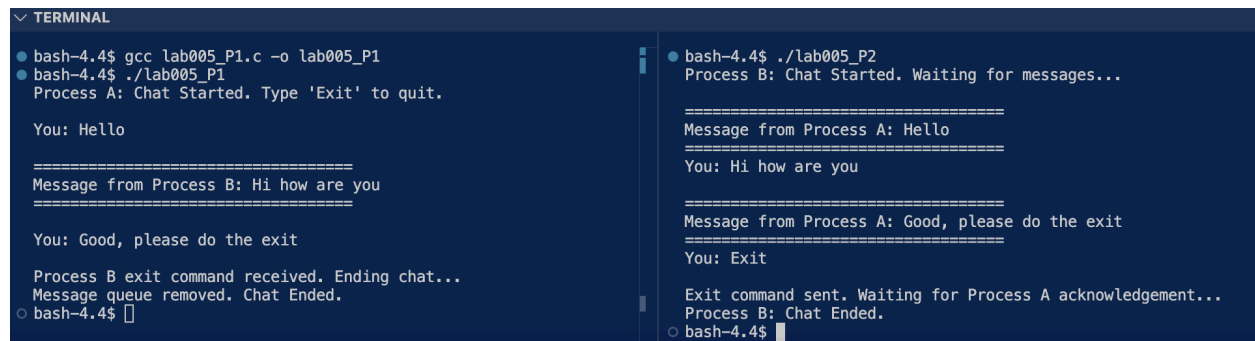
## Part 2 (Reader):

**Program Description:** This program acts as the receiver process, continuously waiting for messages from Process A via a message queue. When a message is received, it is displayed, and the user is prompted to send a response back to Process A. If the received message is “Exit”, Process B acknowledges the termination request by sending a confirmation message before exiting. Similarly, if the user types “Exit”, it sends the termination command to Process A, ensuring a controlled shutdown. Process B plays a critical role in maintaining bidirectional communication while ensuring proper synchronization and message handling.

Process:

1. I defined a struct `msg_buffer` with two fields:
  - a. `long msg_type` to differentiate between sent and received messages.
  - b. `char msg_text[MSG_SIZE]` to store the actual message content.
2. I accessed the existing message queue using `msgget(KEY, 0666)`, ensuring Process B could communicate with Process A.
3. I implemented message reception using `msgrcv()`, allowing Process B to continuously receive and display messages from Process A.
4. I handled user responses by prompting for input, removing the newline character, setting `msg_type = 2`, and using `msgsnd()` to send the response back to Process A.
5. I managed exit conditions by checking if the received message was “Exit”, sending an acknowledgment message back to Process A before terminating.
6. I cleaned up resources by calling `msgctl(msgid, IPC_RMID, NULL)`, removing the message queue before exiting to prevent system resource leaks.

## Program Output:



```

▼ TERMINAL
● bash-4.4$ gcc lab005_P1.c -o lab005_P1
● bash-4.4$ ./lab005_P1
Process A: Chat Started. Type 'Exit' to quit.

You: Hello

=====
Message from Process B: Hi how are you
=====

You: Good, please do the exit

Process B exit command received. Ending chat...
Message queue removed. Chat Ended.
○ bash-4.4$

● bash-4.4$ ./lab005_P2
Process B: Chat Started. Waiting for messages...

=====
Message from Process A: Hello
=====
You: Hi how are you

=====
Message from Process A: Good, please do the exit
=====
You: Exit

Exit command sent. Waiting for Process A acknowledgement...
Process B: Chat Ended.
○ bash-4.4$
```

Figure 2: Program 2 with Reader Exit

## Research Online:

### Questions:

1. How do you make a process wait to receive a message and not return immediately?
  - a. A process can wait for a message using the `msgrcv()` function without the `IPC_NOWAIT` flag. By default, `msgrcv()` blocks execution until a message of the requested type becomes available in the queue. This ensures that the process does not return immediately if no message is present, effectively making it wait until data is received.
2. Message Queue vs Shared Memory (discuss use and differences).
  - a. Message queues are ideal for structured, message-based communication, as the kernel manages message delivery, ensuring reliability and simplicity. In contrast, shared memory enables faster data exchange by allowing direct access to a common memory region but requires explicit synchronization to prevent conflicts. While message queues are better for discrete, asynchronous communication, shared memory is preferred for high-speed data transfer in performance-critical applications.
3. Research use of function `ftok()`, what is its use?
  - a. The `ftok()` function (File-to-Key) is used to generate a unique key for IPC mechanisms like message queues and shared memory. It takes a file path and a project identifier (integer) as inputs and produces a key that different processes can use to access the same IPC resource.

#### 4. What does IPC\_NOWAIT do?

- a. The IPC\_NOWAIT flag is an optional parameter for `msgrcv()` and `msgsnd()` that prevents blocking behavior.
  - i. When used with `msgrcv()`, the function returns immediately if no message of the specified type is available, instead of waiting indefinitely. If no message is present, it fails and returns -1.
  - ii. When used with `msgsnd()`, the function fails immediately if the message queue is full, rather than blocking until space becomes available. If the queue is full, it returns -1, indicating that the message could not be sent.

## Conclusion

In this lab, we successfully implemented Inter-Process Communication (IPC) using message queues to facilitate bidirectional communication between two processes. Process A acted as the sender, creating the message queue, sending user-inputted messages, and handling queue removal, while Process B received messages, responded accordingly, and acknowledged termination requests. We utilized key Linux system calls such as `msgget()`, `msgsnd()`, `msgrcv()`, and `msgctl()` to manage the queue and ensure proper synchronization between processes. By handling exit conditions correctly and ensuring resource cleanup, we reinforced our understanding of message passing mechanisms in operating systems and the importance of efficient process coordination.



## Appendix

Table 1: Program 1

```
// Author: Gianna Foti
// Teacher: Trevor H McClain
// Course: Operating Systems Laboratory - CPE 435-03
// Date: 12 February 2025
// Description: This program is the writer process. It takes a user-inputted message,
// stores it in a shared memory segment, and sets a flag indicating that new data is
// available. The process continues until the user types "Exit", signaling the reader
// process to terminate.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY 5555
#define MSG_SIZE 256

// Define message structure
struct msg_buffer {
    long msg_type;
    char msg_text[MSG_SIZE];
};

int main() {
    int msgid;
    struct msg_buffer msg;

    // Create message queue
    msgid = msgget(KEY, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget failed");
        exit(1);
    }

    printf("Process A: Chat Started. Type 'Exit' to quit.\n");

    while (1) {
        // Get user input
        printf("\nYou: ");
        fgets(msg.msg_text, MSG_SIZE, stdin);
        msg.msg_text[strcspn(msg.msg_text, "\n")] = 0; // Remove newline
        msg.msg_type = 1;
    }
}
```

```

// Send message to queue
if (msgsnd(msgid, &msg, sizeof(msg.msg_text), 0) == -1) {
    perror("msgsnd failed");
    exit(1);
}

// Exit condition
if (strcmp(msg.msg_text, "Exit") == 0) {
    printf("\nExit command sent. Waiting for acknowledgment...\n");

    // Wait for acknowledgment from Process B
    if (msgrcv(msgid, &msg, sizeof(msg.msg_text), 2, 0) == -1) {
        perror("msgrcv failed");
        exit(1);
    }

    // Display acknowledgment message
    printf("\n===== \n");
    printf("%s\n", msg.msg_text);
    printf("===== \n");
    break;
}

// Wait for response from Process B
if (msgrcv(msgid, &msg, sizeof(msg.msg_text), 2, 0) == -1) {
    perror("msgrcv failed");
    exit(1);
}

// If Process B sent "Exit", handle termination
if (strcmp(msg.msg_text, "Exit") == 0) {
    printf("\nProcess B exit command received. Ending chat...\n");
    break;
}

// Properly format received message display
printf("\n===== \n");
printf("Message from Process B: %s\n", msg.msg_text);
printf("===== \n");
}

msgctl(msgid, IPC_RMID, NULL);
printf("Message queue removed. Chat Ended.\n");
return 0;
}

```

Table 2: Program 2

```

// Author: Gianna Foti
// Teacher: Trevor H McClain
// Course: Operating Systems Laboratory - CPE 435-03
// Date: 12 February 2025
// Description: This program is the reader process. It waits for a message from
// Process A in a shared memory segment. Once a new message is available, it
// displays the message, acknowledges it by resetting the flag, and waits for
// the next message. The process continues until it receives "Exit", signaling
// that Process A has terminated, at which point Process B also exits.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY 5555
#define MSG_SIZE 256

// Define message structure
struct msg_buffer {
    long msg_type;
    char msg_text[MSG_SIZE];
};

int main() {
    int msgid;
    struct msg_buffer msg;

    // Access existing message queue
    msgid = msgget(KEY, 0666);
    if (msgid == -1) {
        perror("msgget failed");
        exit(1);
    }

    printf("Process B: Chat Started. Waiting for messages...\n");

    while (1) {
        // Receive message from Process A
        if (msgrcv(msgid, &msg, sizeof(msg.msg_text), 1, 0) == -1) {
            perror("msgrcv failed");
            exit(1);
        }

        // Properly format received message display
    }
}

```

```

printf("\n=====\\n");
printf("Message from Process A: %s\\n", msg.msg_text);
printf("=====\\n");

// Exit condition - Process A is terminating
if (strcmp(msg.msg_text, "Exit") == 0) {
    // Send acknowledgment to Process A
    strcpy(msg.msg_text, "Exit command received. Ending chat.");
    msg.msg_type = 2;

    if (msgsnd(msgid, &msg, sizeof(msg.msg_text), 0) == -1) {
        perror("msgsnd failed");
        exit(1);
    }

    printf("\\nProcess A exit command received. Ending chat...\\n");
    break;
}

// Get user response
printf("You: ");
fgets(msg.msg_text, MSG_SIZE, stdin);
msg.msg_text[strcspn(msg.msg_text, "\\n")] = 0; // Remove newline
msg.msg_type = 2;

// Send response back to Process A
if (msgsnd(msgid, &msg, sizeof(msg.msg_text), 0) == -1) {
    perror("msgsnd failed");
    exit(1);
}

// Exit condition - If user types Exit, send it to Process A
if (strcmp(msg.msg_text, "Exit") == 0) {
    printf("\\nExit command sent. Waiting for Process A acknowledgement...\\n");
    break;
}
}

msgctl(msgid, IPC_RMID, NULL);
printf("Process B: Chat Ended.\\n");
return 0;
}

```