

CPE 345: Operating Systems Laboratory

Lab04

IPC Using Shared Memory

Submitted by: Gianna Foti

Date of Experiment: 2 February 2025

Report Deadline: 5 February 2025

Lab Section: CPE345-03

Demonstration Deadline: 5 February 2025

Introduction

In this laboratory assignment, we explored Inter-Process Communication (IPC) using shared memory in Linux by implementing two independent C programs that communicate without a parent-child relationship. IPC allows processes to efficiently exchange data, and in this case, a shared memory segment serves as the communication medium, enabling direct access to a common data structure. The first process (Writer) prompts the user to enter two floating-point numbers, stores them in the shared memory segment, and sets a flag to indicate that new data is available. The second process (Reader) monitors this flag, retrieves the numbers, computes their sum, displays the result, and resets the flag to indicate completion. This process continues in a loop until the user sets the flag to -1, signaling termination. By implementing real-time process synchronization using shared memory and flags, we gained a deeper understanding of efficient memory sharing in IPC and learned how to handle process coordination in a Linux environment. Additionally, working with system calls such as `shmget()`, `shmat()`, `shmdt()`, and `shmctl()` reinforced our knowledge of shared memory management and its role in process communication.

Theory Topics

1. `shmget()`: This stands for Shared Memory Get. The `shmget()` system call is used to create a new shared memory segment or retrieve an existing one. It returns a shared memory identifier (`shmid`), which is required for further operations like attaching or controlling the memory.
2. `shmat()`: This stands for Attach Shared Memory. The `shmat()` system call attaches a shared memory segment to the address space of the calling process. This allows the process to read and write to the shared memory just like normal variables.
3. `shmctl()`: This stands for Control Shared Memory. The `shmctl()` system call is used to control and manage shared memory segments. It allows processes to modify permissions, retrieve segment status, and remove the segment.

4. `shmdt()`: This stands for Detach Shared Memory. The `shmdt()` system call detaches a shared memory segment from the address space of the calling process. This does not delete the segment, but only disconnects the process from it.

Part 1 (Writer):

Program Description: This program is the writer process, responsible for receiving user input, storing two floating-point numbers in a shared memory segment, and signaling the reader process that new data is available. It repeatedly prompts the user for input and waits for the reader to process the data. The user can choose to continue entering values or exit the program. When exiting, it signals the reader to terminate and then exits silently without printing any messages.

Process:

1. Create a shared memory segment using `shmget()`.
2. Attach to the shared memory segment using `shmat()`.
3. Initialize the flag to 0 to indicate that no new data is available yet.
4. Enter a loop for user input:
 - a. Prompt the user for the first number.
 - b. Prompt the user for the second number.
 - c. Set the flag to 1 to notify the reader process that data is ready.
5. Wait until the reader process resets the flag to 0, indicating the sum has been computed.
6. Ask the user if they want to continue or exit:
 - a. If the user enters 1, the loop repeats with new inputs.
 - b. If the user enters -1, it sets the flag to -1 (to signal the reader process to exit), detaches from shared memory using `shmdt()`, and exits silently (without printing messages).

Program Output:

```
✓ TERMINAL

● bash-4.4$ gcc lab004_P1.c -o lab004_P1
● bash-4.4$ ./lab004_P1
---- Process 1 (Writer) ----

Enter first number: 1
Enter second number: 3
Continue? Yes(1) No(-1): 1

Enter first number: 9
Enter second number: 7
Continue? Yes(1) No(-1): -1
○ bash-4.4$ □
```

Figure 1: Program 1

Part 2 (Reader):

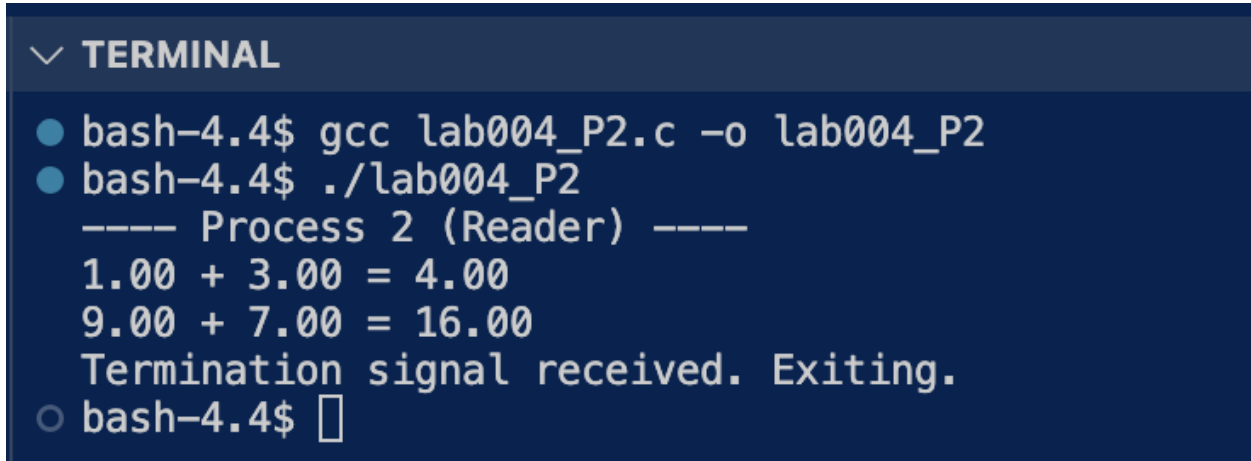
Program Description: This program is the reader process, responsible for detecting when new data is available in shared memory, computing the sum of two numbers, and displaying the result. It continuously monitors the shared memory until the writer process signals termination. When the writer process exits, the reader detects the termination flag (-1) and exits gracefully.

Process:

1. Attach to the existing shared memory segment using `shmat()`.
2. Enter an infinite loop to monitor shared memory:
 - a. If `flag != 1`, it waits (continuously checking every 500ms).
 - b. If `flag == -1`, it detaches (`shmdt()`) and exits, indicating that the writer has terminated.
3. Read `value1` and `value2` from shared memory.
4. Compute the sum and store the result in shared memory.

5. Display the output in the format:
 - a. $3.00 + 2.00 = 5.00$
6. Reset the flag to 0, signaling the writer that it can enter the next input.
7. Repeat steps 2-6 until termination (flag == -1).

Program Output:

A terminal window with a dark blue background and white text. The title bar says '✓ TERMINAL'. The prompt is 'bash-4.4\$'. The user enters 'gcc lab004_P2.c -o lab004_P2'. The prompt is 'bash-4.4\$'. The user enters './lab004_P2'. The program output is: '---- Process 2 (Reader) ----', '1.00 + 3.00 = 4.00', '9.00 + 7.00 = 16.00', 'Termination signal received. Exiting.'. The prompt is 'bash-4.4\$' followed by a cursor.

```
✓ TERMINAL
● bash-4.4$ gcc lab004_P2.c -o lab004_P2
● bash-4.4$ ./lab004_P2
---- Process 2 (Reader) ----
1.00 + 3.00 = 4.00
9.00 + 7.00 = 16.00
Termination signal received. Exiting.
○ bash-4.4$
```

Figure 2: Program 2

Conclusion

In conclusion, this laboratory assignment provided valuable hands-on experience with Inter-Process Communication (IPC) using shared memory in Linux. By implementing two independent C programs, a writer process and a reader process, we successfully demonstrated how processes can efficiently share data without a parent-child relationship. The use of shared memory allowed for direct data exchange, reducing the overhead associated with other IPC mechanisms such as pipes or message queues. Through real-time process synchronization using a shared flag, we ensured that data was correctly written, read, and processed in a structured manner. The implementation reinforced our understanding of key system calls including `shmget()`, `shmat()`, `shmdt()`, and `shmctl()`, which are essential for managing shared memory in Linux. Additionally, debugging synchronization issues and ensuring proper flag handling deepened our knowledge of process coordination and resource management. Overall, this lab

enhanced our understanding of efficient memory sharing in IPC, demonstrating its importance in multi-process environments. By successfully implementing the required functionalities, we gained practical insight into how shared memory facilitates fast and efficient communication between processes in a Linux-based system.

Appendix

Table 1: Program 1

```
/*
Author: Gianna Foti
Teacher: Trevor H McClain
Course: Operating Systems Laboratory - CPE 435-03
Date: 5 February 2025
Description: This program is the writer process. It takes two floating-point numbers
from the user, stores them in a shared memory segment, and sets a flag indicating that
new data is available. The process continues until the user chooses to exit, signaling
the reader process to terminate.
*/

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

#define KEY ((key_t)(1234)) // Shared Memory Key from Lab Instructions
#define MSIZ sizeof(struct info)

// Define the shared memory structure
struct info {
    float value1, value2;
    float sum;
    int flag;
};

int main() {
    // Create shared memory
    int shmid = shmget(KEY, MSIZ, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }

    // Attach to shared memory
    struct info *shm = (struct info *)shmat(shmid, NULL, 0);
    if (shm == (void *)-1) {
        perror("shmat failed");
        exit(1);
    }

    shm->flag = 0; // Initialize flag
```

```

printf("---- Process 1 (Writer) ----\n");

while (1) {
    printf("\nEnter first number: ");
    scanf("%f", &shm->value1);

    printf("Enter second number: ");
    scanf("%f", &shm->value2);

    shm->flag = 1; // Signal Process 2 to compute sum

    while (shm->flag == 1) {
        usleep(500000); // Sleep for 500ms for better responsiveness
    }

    int choice;
    while (1) {
        printf("Continue? Yes(1) No(-1): ");
        if (scanf("%d", &choice) != 1) {
            printf("Invalid input. Enter 1 to continue, -1 to exit.\n");
            while (getchar() != '\n'); // Clear input buffer
            continue;
        }
        if (choice == -1) {
            shm->flag = -1; // Signal termination
            shmdt(shm); // Detach shared memory
            exit(0); // Exit silently (no message)
        } else if (choice == 1) {
            break;
        } else {
            printf("Invalid input. Enter 1 to continue, -1 to exit.\n");
        }
    }
}
return 0;
}

```

Table 2: Program 2

```

// Author: Gianna Foti
// Teacher: Trevor H McClain
// Course: Operating Systems Laboratory - CPE 435-03
// Date: 5 February 2025
// Description: This program is the reader process. It waits for two floating-point numbers
// to be available in a shared memory segment, computes their sum, and displays the result.
// The process continues to monitor the shared memory until the writer process signals termination.

```



```

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

#define KEY ((key_t)(1234))
#define MSIZ sizeof(struct info)

// Define shared memory structure
struct info {
    float value1, value2;
    float sum;
    int flag;
};

int main() {
    // Attach to existing shared memory
    int shmid = shmget(KEY, MSIZ, 0666);
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }

    struct info *shm = (struct info *)shmat(shmid, NULL, 0);
    if (shm == (void *)-1) {
        perror("shmat failed");
        exit(1);
    }

    printf("---- Process 2 (Reader) ----\n");

    while (1) {
        while (shm->flag != 1) {
            if (shm->flag == -1) {
                printf("Termination signal received. Exiting.\n");
                shmdt(shm);
                exit(0);
            }
            usleep(500000); // Sleep for 500ms
        }

        shm->sum = shm->value1 + shm->value2;
        printf("%.2f + %.2f = %.2f\n", shm->value1, shm->value2, shm->sum);

        shm->flag = 0; // Indicate sum is computed
    }
}

```

```
shmdt(shm);  
return 0;  
}
```