Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam

Department of Computer Science & Engineering

M.Tech. CSE - III Semester (2025-26) Assignment Report - Experiment 6

Course: ICS 13 13 - Operating System Practices Laboratory

Name: Simiyon Vinscent Samuel L

Reg No: 31222247001062

Academic Year: 2025-26 (ODD)

Chennai - 603110

Experiment No: 6

Title: Inter-process Communications using Shared Memory

Objective

To develop applications that use inter-process communication concepts using shared memory. The program should:

• Implement shared memory IPC using system calls: shmget(), shmat(), shmdt(), and shmctl()

• Create two applications:

1. Parent-child process for name conversion to uppercase
2. Client-server chat application using shared memory
   • Demonstrate efficient data exchange between processes without kernel involvement
   • Handle synchronization and communication between separate processes

Program Design

The programs implement shared memory IPC with the following components:

Shared Memory System Calls:

• **shmget()**: Creates or accesses a shared memory segment

• **shmat()**: Attaches the shared memory segment to the process address space

- **shmdt()**: Detaches the shared memory segment from the process
- **shmctl()**: Performs control operations on shared memory (including cleanup)

Key Features:

- Uses ftok() to generate unique keys for shared memory segments
- Implements proper synchronization using flags and polling
- Handles process communication without data copying through kernel
- Provides cleanup mechanisms to prevent memory leaks

API Requirements

The programs implement the following key functions:

- **create_shared_memory**: Creates shared memory segment using shmget()
- **attach_shared_memory**: Attaches segment to process using shmat()
- **detach_shared_memory**: Detaches segment using shmdt()
- **cleanup_shared_memory**: Removes segment using shmctl() with IPC_RMID

Header Files Used

- `<sys/ipc.h>`: Provides IPC constants and ftok() function
- `<sys/shm.h>`: Contains shared memory system calls and constants
- `<sys/wait.h>`: For process synchronization using wait()
- `<unistd.h>`: Provides fork(), sleep(), and other system calls
- `<ctype.h>`: For character manipulation functions like toupper()

Input and Constraints

Application 1 - Name Conversion:

- Input: Name string from parent process
- Constraint: Maximum name length of 100 characters
- Process: Child converts to uppercase using shared memory

Application 2 - Chat Application:

- Input: Text messages from client and server
- Constraint: Maximum message length of 256 characters
- Process: Real-time bidirectional communication
- Exit condition: Either party types "bye"

Program Implementation

Application 1: Parent-Child Name Conversion

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <ctype.h>

#define SHM_SIZE 100

int main() {
    key_t key;
    int shmid;
    char *shared_memory;
    pid_t pid;

    // Generate a key for shared memory
    key = ftok("/tmp", 65);

    // Create shared memory segment
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }

    // Attach shared memory
    shared_memory = (char *)shmat(shmid, NULL, 0);
    if (shared_memory == (char *)-1) {
        perror("shmat failed");
        exit(1);
    }

    pid = fork();
```

```
    if (pid == 0) {
        // Child process - convert to uppercase
        printf("Child process: Waiting for input from parent...\n");

        // Wait until parent writes the name
        while (strlen(shared_memory) == 0) {
            sleep(1);
        }

        printf("Child process: Converting '%s' to uppercase\n", shared_memory);

        // Convert to uppercase
        for (int i = 0; shared_memory[i]; i++) {
            shared_memory[i] = toupper(shared_memory[i]);
        }

        printf("Child process: Name in Uppercase: %s\n", shared_memory);

        // Detach shared memory
        shmdt(shared_memory);
        exit(0);

    } else if (pid > 0) {
        // Parent process - get name input
        char name[100];

        printf("Parent Process: Enter a name to convert into uppercase: ");
        fgets(name, sizeof(name), stdin);

        // Remove newline character
        name[strcspn(name, "\n")] = 0;

        // Write to shared memory
        strcpy(shared_memory, name);

        // Wait for child to complete
        wait(NULL);

        printf("Parent Process: Final result: %s\n", shared_memory);
```

```
        // Detach and remove shared memory
        shmdt(shared_memory);
        shmctl(shmid, IPC_RMID, NULL);

    } else {
        perror("fork failed");
        exit(1);
    }

    return 0;
}
```

Application 2: Chat Server (server.c)

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

#define SHM_SIZE 1024
#define MAX_MSG 256

typedef struct {
    char client_msg[MAX_MSG];
    char server_msg[MAX_MSG];
    int client_ready;
    int server_ready;
    int chat_active;
} ChatData;

int main() {
    key_t key;
    int shmid;
    ChatData *chat_data;
    char message[MAX_MSG];

    // Generate a key for shared memory
```

```c
    key = ftok("/tmp", 66);

    // Create shared memory segment
    shmid = shmget(key, sizeof(ChatData), 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("Server: shmget failed");
        exit(1);
    }

    // Attach shared memory
    chat_data = (ChatData *)shmat(shmid, NULL, 0);
    if (chat_data == (ChatData *)-1) {
        perror("Server: shmat failed");
        exit(1);
    }

    // Initialize shared data
    memset(chat_data, 0, sizeof(ChatData));
    chat_data->chat_active = 1;

    printf("Server: Chat server started. Waiting for client...\n");

    while (chat_data->chat_active) {
        // Wait for client message
        while (!chat_data->client_ready && chat_data->chat_active) {
            sleep(1);
        }

        if (!chat_data->chat_active) break;

        printf("Received from client: %s\n", chat_data->client_msg);

        // Check if client wants to exit
        if (strcmp(chat_data->client_msg, "bye") == 0 ||
            strcmp(chat_data->client_msg, "Bye") == 0) {
            printf("Server: Client disconnected.\n");
            chat_data->chat_active = 0;
            break;
        }
```

```
        // Reset client ready flag
        chat_data->client_ready = 0;

        // Get server response
        printf("Enter response: ");
        fgets(message, sizeof(message), stdin);
        message[strcspn(message, "\n")] = 0; // Remove newline

        // Copy to shared memory
        strcpy(chat_data->server_msg, message);
        chat_data->server_ready = 1;

        // Check if server wants to exit
        if (strcmp(message, "bye") == 0 || strcmp(message, "Bye") == 0) {
            printf("Server: Shutting down...\n");
            chat_data->chat_active = 0;
            break;
        }
    }

    // Cleanup
    shmdt(chat_data);
    shmctl(shmid, IPC_RMID, NULL);
    printf("Server: Shared memory cleaned up.\n");

    return 0;
}
```

Application 2: Chat Client (client.c)

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

#define SHM_SIZE 1024
#define MAX_MSG 256
```

```
typedef struct {
    char client_msg[MAX_MSG];
    char server_msg[MAX_MSG];
    int client_ready;
    int server_ready;
    int chat_active;
} ChatData;

int main() {
    key_t key;
    int shmid;
    ChatData *chat_data;
    char message[MAX_MSG];

    // Generate the same key as server
    key = ftok("/tmp", 66);

    // Get existing shared memory segment
    shmid = shmget(key, sizeof(ChatData), 0666);
    if (shmid == -1) {
        perror("Client: shmget failed - make sure server is running first");
        exit(1);
    }

    // Attach shared memory
    chat_data = (ChatData *)shmat(shmid, NULL, 0);
    if (chat_data == (ChatData *)-1) {
        perror("Client: shmat failed");
        exit(1);
    }

    printf("Client: Connected to chat server\n");
    printf("Client: Type 'bye' to exit\n");

    while (chat_data->chat_active) {
        // Get client message
        printf("Client: ");
        fgets(message, sizeof(message), stdin);
        message[strcspn(message, "\n")] = 0; // Remove newline
```

```c
        // Copy to shared memory
        strcpy(chat_data->client_msg, message);
        chat_data->client_ready = 1;

        // Check if client wants to exit
        if (strcmp(message, "bye") == 0 || strcmp(message, "Bye") == 0) {
            printf("Client: Disconnecting...\n");
            break;
        }

        // Wait for server response
        while (!chat_data->server_ready && chat_data->chat_active) {
            sleep(1);
        }

        if (!chat_data->chat_active) break;

        printf("Received from server: %s\n", chat_data->server_msg);

        // Check if server wants to exit
        if (strcmp(chat_data->server_msg, "bye") == 0 ||
            strcmp(chat_data->server_msg, "Bye") == 0) {
            printf("Client: Server disconnected.\n");
            break;
        }

        // Reset server ready flag
        chat_data->server_ready = 0;
    }

    // Cleanup
    shmdt(chat_data);
    printf("Client: Disconnected from server.\n");

    return 0;
}
```

Sample Test Cases

Application 1: Name Conversion Output

```
Parent Process: Enter a name to convert into uppercase: john doe
Child process: Waiting for input from parent...
Child process: Converting 'john doe' to uppercase
Child process: Name in Uppercase: JOHN DOE
Parent Process: Final result: JOHN DOE
```

Application 2: Chat Application Output

Server Terminal:

```
Server: Chat server started. Waiting for client...
Received from client: Hello
Enter response: Hi
Received from client: This is a chat application
Enter response: Yes, it works great!
Received from client: bye
Server: Client disconnected.
Server: Shared memory cleaned up.
```

Client Terminal:

```
Client: Connected to chat server
Client: Type 'bye' to exit
Client: Hello
Received from server: Hi
Client: This is a chat application
Received from server: Yes, it works great!
Client: bye
Client: Disconnecting...
Client: Disconnected from server.
```

Compilation and Execution Instructions

For Application 1 (Name Conversion):

```
cc -o name_converter name_converter.c
./name_converter
```

For Application 2 (Chat System):

```
# Compile both programs
cc -o server server.c
cc -o client client.c

# Run server in one terminal
./server

# Run client in another terminal
./client
```

Performance Analysis

The shared memory IPC implementation provides:

Efficiency:

- **Zero-copy communication**: Data is not copied between kernel and user space
- **Direct memory access**: Processes directly read/write to shared segments
- **Minimal overhead**: No system call overhead for data transfer once attached

Synchronization:

- **Polling mechanism**: Uses flags and sleep() for coordination
- **Process synchronization**: Parent waits for child using wait()
- **Clean termination**: Proper cleanup prevents memory leaks

Scalability:

- **Multiple process support**: Can be extended to support multiple clients
- **Configurable memory size**: Shared memory size can be adjusted based on needs
- **Persistent communication**: Shared memory persists until explicitly removed

Justification for Shared Memory IPC

Shared memory is the most efficient IPC mechanism because:

Performance Benefits:

- **Fastest IPC method**: No kernel involvement in data transfer
- **Large data transfer**: Suitable for transferring large amounts of data
- **Low latency**: Direct memory access eliminates copying overhead

Practical Applications:

- **Database systems**: Multiple processes accessing common data structures
- **Real-time systems**: Low-latency communication requirements
- **Multimedia applications**: Sharing large buffers between processes

Comparison with other IPC methods:

- **Pipes/FIFOs**: Require data copying through kernel buffers
- **Message queues**: Limited message size and kernel overhead
- **Sockets**: Network protocol overhead even for local communication

Learning Outcome

This exercise enhanced understanding of:

- **Shared Memory IPC**: Implementation using shmget(), shmat(), shmdt(), and shmctl()
- **Process Synchronization**: Coordination between parent-child and client-server processes
- **Memory Management**: Proper attachment, detachment, and cleanup of shared segments
- **Inter-process Communication**: Efficient data sharing without kernel involvement
- **System Programming**: Understanding of Unix IPC mechanisms and their practical applications
- **Synchronization Techniques**: Implementation of polling-based coordination between processes

*) If message is "bye" break loop.
*) write a response to shared memory
*) Signal semaphore.

3. Terminate
*) clean up shared memory.

Test-case:-

T1 - Server	T2 - Client
Server: Hi	received: Hi
~~ ~~ recieved: hello	~~ser~~ Client: hello
Server: this is chat	received: this is chat
recieved: okay great	client: okay great
server: Bye	received: Bye

Evaluation

S.No	Topic	Maximum mark	mark obtained
1.	Aim & Algorithm	4.	4
2.	Test cases & output	4	4
3.	Best practice	2	2
			10