Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam

Department of Computer Science & Engineering

M.Tech. CSE - III Semester (2025-26) Assignment Report - Experiment 7

Course: ICS 1313 - Operating System Practices Laboratory

Name: Simiyon Vinscent Samuel L

Reg No: 31222247001062

Academic Year: 2025-26 (ODD) Chennai -

603110

Experiment No: 7

Title: Inter-process Synchronization using Semaphores

1 Objective

To develop applications that use inter-process synchronization concepts using semaphores. The program should:

- Implementsemaphore-basedsynchronizationusingPOSIXAPIs: `sem_init()`, `sem_wait()`, `sem_post()`, and `sem_destroy()`.
- Use shared memory for data buffer combined with semaphores for synchronization.
- Create two applications:
    1. Parent-child process for producer-consumer with string input.
    2. Client-serverproducer-consumerforgeneratingandconsumingNrandom numbers.
- Demonstrate proper synchronization to avoid race conditions in producerconsumer problem.
- Handle inter-process communication with semaphores without polling.

2 Program Design

The programs implement semaphore synchronization with the following components:

2.1 Semaphore System Calls

- `sem_init()`: Initializes a semaphore with shared flag for inter-process use.
- `sem_wait()`: Decrements (waits on) a semaphore.
- `sem_post()`: Increments (signals) a semaphore.
- `sem_destroy()`: Destroys a semaphore.

2.2 Shared Memory Integration

- Uses shared memory for buffer and places semaphores in shared memory for inter-process access.

2.3 Key Features

- Uses `ftok()` to generate unique keys for shared memory segments.
- Implementsproducer-consumersynchronizationusingempty, full, andmutex semaphores.
- Handles process communication with proper wait and signal operations.
- Provides cleanup mechanisms for semaphores and shared memory to prevent leaks.

2.4 API Requirements

The programs implement the following key functions:

- `create_shared_memory`: Createssharedmemorysegmentusing`shmget()`.
- `attach_shared_memory`: Attaches segment to process using `shmat()`.
- `detach_shared_memory`: Detaches segment using `shmdt()`.
- `cleanup_shared_memory`: Removessegmentusing`shmctl()`with`IPC_RMID`.
- Semaphore operations: `sem_init` with `pshared=1`, `sem_wait`, `sem_post`, `sem_destroy`.

2.5 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.
- `<semaphore.h>`: For POSIX semaphore functions.

- `<pthread.h>`: For thread-related types (required for semaphores).
- `<stdlib.h>`: For random number generation.

2.6 Input and Constraints

Application 1 - String Producer-Consumer:

- Input: String from parent process (e.g., ”HELLO”).
- Constraint: Processes character by character; buffer size 1.
- Process: Parent produces characters, child consumes and displays.

Application 2 - Random Numbers Producer-Consumer:

- Input: Number N from producer (client).
- Constraint: N up to 100; buffer size 10 for numbers.
- Process: Client generates N random numbers, server consumes and displays.
- Exit condition: After consuming all N numbers.

3 Program Implementation

```
#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 <semaphore.h>
#include <fcntl.h>
#define BUFFER_SIZE 1
#define SHM_SIZE sizeof(char) + 3 *
sizeof(sem_t) typedef struct { char
buffer[BUFFER_SIZE]; sem_t empty; sem_t full;
sem_t mutex; } SharedData; int main() { key_t
key; int shmid;
    SharedData *shared_data;
    pid_t pid;
    char input[] = ”HELLO”;
    // Generate a key for shared
    memory key = ftok(”/tmp”, 67); //
    Create shared memory segment
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror(”shmget failed”);
        exit(1);
    }
    // Attach shared memory
    shared_data = (SharedData *)shmat(shmid, NULL,
    0); if (shared_data == (SharedData *)-1) {
    perror(”shmat failed”); exit(1);
    }
    // Initialize semaphores (pshared=1 for inter-process)
    sem_init(&shared_data->empty, 1, 1); // Buffer empty
    initially sem_init(&shared_data->full, 1, 0); // No items
    initially sem_init(&shared_data->mutex, 1, 1); // Mutex
    unlocked pid = fork();
```

3.1 Application 1: Parent-Child String Producer-Consumer

```
    if (pid == 0) {
        // Child process - Consumer
        for (int i = 0; i < strlen(input); i++) {
            printf("Consumer acquired semaphore Full\n");
            sem_wait(&shared_data->full);
            printf("Consumer acquired semaphore Mutex\n");
            sem_wait(&shared_data->mutex);
            // Consume item
            printf("Consumer consumed item %c\n", shared_data->
               buffer[0]);
            sem_post(&shared_data->mutex);
            printf("Consumer released semaphore Mutex\n");
            sem_post(&shared_data->empty);
            printf("Consumer released semaphore empty\n");
        }
        printf("Consumer exited\n");
        // Detach shared memory
        shmdt(shared_data);
        exit(0);
    } else if (pid > 0) {
        // Parent process - Producer for (int i = 0; i <
        strlen(input); i++) { printf("Producer acquired
        semaphore Empty\n");
            sem_wait(&shared_data->empty);
            printf("Producer acquired
            semaphore                              Mutex\n");
            sem_wait(&shared_data->mutex);
            // Produce item
            shared_data->buffer[0] = input[i];
            printf("Producer produced the
            item:                                  %c\n", input[i]);
            sem_post(&shared_data->mutex);
            printf("Producer released
            semaphore                              Mutex\n");
            sem_post(&shared_data->full);
            printf("Producer released
            semaphore                              full\n");
```

```
        }
        printf("Producer exited\n");
        // Wait for child to complete
        wait(NULL);
        // Destroy semaphores
        sem_destroy(&shared_data->empty);
        sem_destroy(&shared_data->full);
        sem_destroy(&shared_data->mutex);
        // Detach and remove shared memory
        shmdt(shared_data);
        shmctl(shmid, IPC_RMID, NULL);
    } else {
        perror("fork failed");
        exit(1);
    } return
    0;
}
```

3.2 Application 2: Producer Client (producer.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <semaphore.h>
#include <time.h>
#define BUFFER_SIZE 10
#define SHM_SIZE BUFFER_SIZE * sizeof(int) + 3 * sizeof(sem_t)
+ sizeof(int) #define MAX_N 100 typedef struct { int
buffer[BUFFER_SIZE]; int n; // Number of items to produce sem_t
empty; sem_t full; sem_t mutex; int in; int out; } SharedData;
int main() { key_t key; int shmid;
    SharedData *shared_data;
    // Generate a key for shared
    memory key = ftok("/tmp", 68); //
    Create shared memory segment
    shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("Producer: shmget failed");
        exit(1);
    }
    // Attach shared memory
    shared_data = (SharedData *)shmat(shmid, NULL,
    0); if (shared_data == (SharedData *)-1) {
    perror("Producer: shmat failed"); exit(1);
    }
    // Initialize semaphores and indices
    sem_init(&shared_data->empty, 1,
    BUFFER_SIZE); sem_init(&shared_data->full, 1,
    0); sem_init(&shared_data->mutex, 1, 1);
    shared_data->in = 0; shared_data->out = 0; //
    Get N from user
    printf("Producer: Enter N (number of random numbers to generate)
       : "); scanf("%d",
    &shared_data->n);
    srand(time(NULL));
```

```
    int produced = 0;
    while (produced < shared_data->n) {
        sem_wait(&shared_data->empty);
        sem_wait(&shared_data->mutex);
        // Produce random number
        int num = rand() % 100;
        shared_data->buffer[shared_data->in] = num; shared_data-
        >in = (shared_data->in + 1) % BUFFER_SIZE;
        printf("Producer produced: %d\n", num);
        sem_post(&shared_data->mutex); sem_post(&shared_data-
        >full);
        produced++;
    }
    // Signal end by producing -1
    sem_wait(&shared_data->empty); sem_wait(&shared_data-
    >mutex);
    shared_data->buffer[shared_data->in] = -1; shared_data-
    >in = (shared_data->in + 1) % BUFFER_SIZE;
    sem_post(&shared_data->mutex);
    sem_post(&shared_data->full);
    // Cleanup (producer cleans up after consumer will exit)
    sleep(5); // Wait for consumer to finish
    sem_destroy(&shared_data->empty); sem_destroy(&shared_data-
    >full); sem_destroy(&shared_data->mutex);
    shmdt(shared_data);
    shmctl(shmid, IPC_RMID, NULL);
    printf("Producer: Shared memory cleaned up.\n");
    return 0;

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <semaphore.h>
#define BUFFER_SIZE 10
#define SHM_SIZE BUFFER_SIZE * sizeof(int) + 3 * sizeof(sem_t) +
   sizeof(int)
typedef struct { int
    buffer[BUFFER_SIZE];
    int n; sem_t empty;
    sem_t full; sem_t
    mutex; int in; int out;
```

3.3 Application 2: Consumer Server (consumer.c)

```
} SharedData;
int main() {
key_t key; int
shmid;
    SharedData *shared_data;
    // Generate the same key as producer
    key = ftok("/tmp", 68);
    // Get existing shared memory segment shmid = shmget(key,
    SHM_SIZE, 0666); if (shmid == -1) { perror("Consumer:
    shmget failed - make sure producer is running first");
        exit(1);
    }
    // Attach shared memory
    shared_data = (SharedData *)shmat(shmid, NULL,
    0); if (shared_data == (SharedData *)-1) {
    perror("Consumer: shmat failed"); exit(1);
    }
    printf("Consumer: Connected, waiting for %d numbers\n",
       shared_data->n);
    int consumed = 0;
    while (consumed < shared_data->n) { sem_wait(&shared_data-
        >full); sem_wait(&shared_data->mutex);
        // Consume number
        int num = shared_data->buffer[shared_data->out];
        if (num == -1) break; // End signal
        shared_data->out = (shared_data->out + 1) % BUFFER_SIZE;
        printf("Consumer consumed: %d\n", num);
        sem_post(&shared_data->mutex); sem_post(&shared_data-
        >empty);
        consumed++;
    }
    // Cleanup
    shmdt(shared_data);
    printf("Consumer: Disconnected.\n");
    return 0;
}
```

4 Sample Test Cases

4.1 Application 1: String Producer-Consumer Output

```
Enter the input string: HELLO
Producer acquired semaphore Empty
Producer acquired semaphore Mutex
Producer produced the item: H
Producer released semaphore Mutex
Producer released semaphore full
Consumer acquired semaphore Full
Consumer acquired semaphore Mutex
Consumer consumed item H
Consumer released semaphore Mutex
Consumer released semaphore empty
Producer acquired semaphore Empty
Producer acquired semaphore Mutex
Producer produced the item: E
Producer released semaphore Mutex
Producer released semaphore full
```

```
Consumer acquired semaphore Full
Consumer acquired semaphore Mutex
Consumer consumed item E
Consumer released semaphore Mutex
Consumer released semaphore empty
-- (continues for L, L, O) -Consumer
exited
Producer exited
```

4.2 Application 2: Random Numbers Producer-Consumer Output Producer Terminal:

```
Producer: Enter N (number of random numbers to generate): 5
Producer produced: 83
Producer produced: 86
Producer produced: 77
Producer produced: 15
Producer produced: 93
Producer: Shared memory cleaned up.
```

Consumer Terminal:

```
Consumer: Connected, waiting for 5 numbers
Consumer consumed: 83
Consumer consumed: 86
Consumer consumed: 77
Consumer consumed: 15
Consumer consumed: 93
Consumer: Disconnected.
```

5 Compilation and Execution Instructions

For Application 1 (String Producer-Consumer):

```
cc -o string_pc string_pc.c -lpthread
./string_pc
```

For Application 2 (Random Numbers System):

```
# Compile both programs cc -o
producer producer.c -lpthread cc -
o consumer consumer.c -lpthread
# Run producer in one terminal
./producer
# Run consumer in another terminal
./consumer
```

6 Performance Analysis

The semaphore synchronization implementation provides:

6.1 Efficiency

- **Proper synchronization**: Avoids busy-waiting with blocking `sem_wait`.
- **Directmemoryaccess**: Combinedwithsharedmemoryforzero-copytransfer.
- **Minimal overhead**: Semaphore operations are lightweight.

6.2 Synchronization

- **Semaphoremechanism**: Usesempty, full, mutexforclassicproducer-consumer.
- **Process synchronization**: Parent waits for child using `wait()`.
- **Clean termination**: Proper destruction prevents resource leaks.

6.3 Scalability

- **Multiple item support**: Buffer size can be adjusted.
- **Configurable N**: Handles variable number of items.
- **Persistent resources**: Shared memory and semaphores persist until removed.

7 Justification for Semaphore Synchronization

Semaphores are essential for synchronization in shared memory IPC because:

7.1 Performance Benefits

- **Efficient blocking**: Processes block instead of polling, saving CPU.
- **Atomic operations**: Prevent race conditions in critical sections.
- **Flexible counting**: Empty and full semaphores handle buffer capacity.

7.2 Practical Applications

- **Multi-threaded programs**: Synchronization in concurrent access.
- **Real-time systems**: Predictable blocking behavior.
- **Distributed computing**: Basis for more complex locks/mutexes.

7.3 Comparison with Other Synchronization Methods

- **Polling/Flags**: Wastes CPU cycles with busy-waiting.
- **Message queues**: Higher overhead for small data.
- **Conditionvariables**: Morecomplex, butsemaphoresaresimplerforcounting.

8 Learning Outcome

This exercise enhanced understanding of:

- **SemaphoreSynchronization**: Implementationusing`sem_init()`, `sem_wait()`, `sem_post()`, and `sem_destroy()`.
- **Producer-ConsumerProblem**: Classicsynchronizationwithsharedbuffer.
- **Resource Management**: Proper initialization and cleanup of semaphores and shared memory.
- **Inter-processCommunication**: Combiningsharedmemorywithsemaphores.
- **System Programming**: POSIX APIs for inter-process synchronization.
- **Concurrency Control**: Avoiding race conditions with mutex and counting semaphores.

Test case -3

Input: N = 0

producer: exit immediately

consumer: exit immediately

nothing is consumed and produced.

S.NO	Topic	Maximum mark	Mark obtained
1.	Aim and algorithm	4	4
2.	Testcases and output	4	4
3.	Best practices and creativity	2	2
			10

Learning outcome:-

*) gained practical knowledge in using posix semaphore.

*) deals with shared memory for IPC.

*) gained insight on full, empty, ~~and~~ circular queue and buffer.