University of Boumerdes

Multi-Class Producer-Consumer Synchronization Using Shared Memory and Semaphores in C

Rapport SE 2

By

Mehdid Samy Abderraouf

Belhaddad Chawki Ahmed Ramzy

Course given by:

F. Sabba

Department of Computer science

09-12-2023

CONTENTS

1. INTRODUCTION

In concurrent and parallel computing, effective communication among processes is essential for efficient and synchronized systems. Managing interactions and shared resources is crucial to prevent conflicts and maintain data integrity.

1.1. PROBLEM

In this task, we aim to implement a solution for the multi-class producer-consumer synchronization problem. Four producers (P1, P2, P3, P4) and two consumers (C1, C2) share a common buffer.
The challenge is to synchronize access to the buffer, ensuring adherence to class-specific constraints (P1/P2 in CL1, P3/P4 in CL2, C1/C2 consume messages from their respective classes).

1.2. APPROACH

To tackle the multi-class producer-consumer synchronization problem, we adopted a solution leveraging shared memory and semaphores in the C programming language. The overall approach involves the creation of shared memory for the buffer and the use of semaphores to synchronize access to this shared resource.

Each producer process is responsible for generating messages, and each consumer process is tasked with consuming messages from the shared buffer. Semaphores are employed to coordinate the access of processes to the buffer, preventing conflicts and ensuring that the specified constraints are met.

2. IMPLEMENTATION

In this section, we provide a detailed explanation of the implementation of the multi-class producer-consumer synchronization solution using shared memory and semaphores in the C programming language.

2.1. SHARED MEMORY AND DATA STRUCTURES

For inter-process communication, we utilize shared memory with a message buffer and a class integer for the producer.

```c
typedef struct {
    char buffer[256];
    int class; // 1 for CL1, 2 for CL2
} shared_memory;
```

codeSamples/typedef.c

2.2. SEMAPHORE INITIALIZATION

```c
semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0644);
if (semid == -1) {
    perror("semget");
    exit(1);
}

// Initialize semaphore value
if (semctl(semid, 0, SETVAL, 1) == -1) {
    perror("semctl");
    exit(1);
}
```

codeSamples/semaphore_init.c

2.3. PRODUCER FUNCTION

The producer function generates messages and updates the shared memory. It utilizes semaphores for synchronization, ensuring mutual exclusion during critical sections.

```c
void producer(int class, int semid, shared_memory* shm) {
    struct sembuf sb;

    while (1) {
        sb.sem_num = 0;
        sb.sem_op = -1;
        sb.sem_flg = 0;
        semop(semid, &sb, 1);

        // Producer logic...

        sb.sem_op = 1;
        semop(semid, &sb, 1);

        sleep(1); // Introduce delay to simulate message
           production
    }
}
```

codeSamples/producer.c

2.4. CONSUMER FUNCTION

The consumer function consumes messages from the shared buffer based on the class. It also utilizes semaphores to ensure proper synchronization.

```c
void consumer(int class, int semid, shared_memory* shm) {
    struct sembuf sb;

    while (1) {
        sb.sem_num = 0;
        sb.sem_op = -1;
        sb.sem_flg = 0;
        semop(semid, &sb, 1);

        // Consumer logic...

        sb.sem_op = 1;
        semop(semid, &sb, 1);

        sleep(1); // Introduce delay to simulate message
           consumption
    }
}
```

codeSamples/consumer.c

2.5. FORKING PROCESSES

```c
for (int i = 1; i <= 4; i++) {
    if (fork() == 0) {
        producer(i <= 2 ? 1 : 2, semid, shm);
        exit(0);
    }
  }

for (int i = 1; i <= 2; i++) {
    if (fork() == 0) {
        consumer(i, semid, shm);
        exit(0);
    }
}
```

codeSamples/fork.c

3. CHALLENGES

During the implementation of the multi-class producer-consumer synchronization system, several challenges were encountered. In this section, we outline the key challenges faced and discuss the strategies employed to address them.

3.1. CHALLENGE 1: SYNCHRONIZATION

One of the primary challenges was achieving effective synchronization among the multiple processes, ensuring that producers and consumers operate correctly without data inconsistencies.

Solution: We implemented semaphore-based synchronization to control access to the shared memory. This approach helped in preventing race conditions and maintaining the integrity of the data.

3.2. CHALLENGE 2: SHARED MEMORY ACCESS

Managing shared memory access and preventing conflicts between processes accessing the shared buffer posed a significant challenge.

Solution: Careful consideration was given to the order of operations and the use of semaphores to regulate access. This helped in avoiding conflicts and maintaining the class-based constraints.

3.3. CHALLENGE 3: DEBUGGING FORKED PROCESSES

Debugging issues related to forked processes, especially in identifying the origin of errors and ensuring proper termination, presented a challenge.

Solution: Extensive logging and debugging statements were incorporated into the code. Additionally, systematic testing and monitoring were employed to trace the behavior of individual processes.

4. RESULTS

4.1. SCENARIO 1: CONCURRENT EXECUTION

Observations:

- Producers and consumers worked well together simultaneously.
- No issues observed with messages from the same class.
- The system handled multiple processes without errors.

4.2. SCENARIO 2: STRESS TESTING

Observations:

- The system handled stress conditions effectively.
- No message overflow or system instability during stress tests.

4.3. SCENARIO 3: CLASS-BASED CONSTRAINTS

Observations:

- Producers and consumers respected class constraints.
- Shared memory maintained proper classification.

4.4. SCENARIO 4: ERROR HANDLING

Observations:

- Graceful error handling observed in unexpected events.
- Proper cleanup ensured shared memory release on program termination.

5. CONCLUSION

The multi-class producer-consumer synchronization system successfully met the project requirements. Through testing and observation, several key conclusions can be drawn.

5.1. ACHIEVEMENTS

- The system effectively handled concurrent execution, ensuring proper coordination among producers and consumers.
- Stress testing demonstrated the robustness of the system, with no observable issues under high loads.
- Class-based constraints were maintained, preventing cross-class interactions and ensuring proper message handling.
- The system exhibited graceful error handling, responding well to unexpected events without compromising stability.

5.2. LESSONS LEARNED

This project provided valuable insights into process synchronization and shared memory management. Key takeaways include:

- The importance of using synchronization mechanisms, such as semaphores, to prevent race conditions.
- Proper design considerations for shared memory structures to maintain data integrity.
- The significance of error handling mechanisms for robust system behavior.