Report

Author: Roberto Cervantes
Username: cs017

Abstract

This report outlines the enhanced use of semaphores in shared memory synchronization for concurrent processes in C. The objective was to manage concurrent access by four child processes updating a shared variable without race conditions. Semaphores were introduced as critical synchronization mechanisms, ensuring a safe increment operation and maintaining execution order among child processes. This report includes system setup, revised implementation details, and an analysis of semaphore utilization effectiveness.

1. Introduction

The importance of robust process synchronization is heightened when managing shared memory in multi-process environments. In this project, four child processes increment a shared memory variable to distinct target values, each attempting to reach a specific goal. The revised program utilizes POSIX semaphores to guard the critical section effectively, ensuring consistency in the shared memory variable's final value.

This project refines our previous implementation by enhancing the semaphore mechanisms to lock and unlock shared memory operations, thus preventing any simultaneous access issues that could compromise data integrity. Each process communicates completion to the parent process using the wait function, which also organizes the final output sequence before program termination.

2. System Setup

- **Programming Language:** C
- **Compiler:** GCC
- **Operating System:** Linux

- **Development Environment:** This project was executed on a Linux virtual machine
- **Development Tools:** Visual Studio Code, GitHub (for version control)

3. Implementation Details

3.1 Shared Memory

The shared memory block holds the counter and flags set by each process as they reach completion. This setup remains largely unchanged from Assignment 1, with added semaphore usage to enforce access control.

3.2 Process Synchronization via Semaphores

Each child process obtains exclusive access to the shared memory counter by "locking" a semaphore before modifying the value and "unlocking" it immediately after. This ensures that only one process can access and increment the counter at any time, effectively preventing race conditions.

3.3 Critical Section Protection

The use of a binary (or counting) semaphore restricts access to the critical section, thereby protecting the shared memory's integrity. When a process reaches its target count, it signals the parent, which waits for all child processes to finish before releasing shared resources and semaphores.

3.4 Semaphore Usage

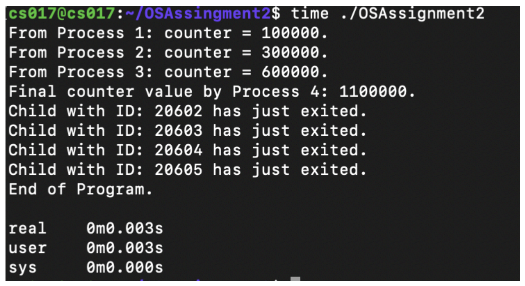
In this program, semaphores are used to enforce a specific order of execution among the four processes. Each process waits for a signal from the preceding process by calling sem_wait, which ensures that only one process increments the shared counter at a time. After completing its assigned increments, each process calls sem_post to signal the next process. This sequential signaling through semaphores ensures that the shared variable is accessed in a controlled, serialized manner, preventing race conditions.

This approach eliminates the need for mutex locking on shared memory because only one process accesses total at any given time, as enforced by the semaphores' order of signaling. The use of sem_open to create named semaphores allows each process to synchronize effectively even when created in different parts of the program.

4. Quantitative Data

- **Number of Child Processes:** 4
- **Shared Memory Size:** 128 bytes
- **Number of Iterations:**
    - Process 1: 100,000
    - Process 2: 200,000
    - Process 3: 300,000
    - Process 4: 500,000

Execution Time (for final run):

- **Real Time**: 0m0.003s
- **User Time**: 0m0.002s
- **System Time**: 0m0.00s
- 

```
cs017@cs017:~/OSAssingment2$ time ./OSAssignment2
From Process 1: counter = 100000.
From Process 2: counter = 300000.
From Process 3: counter = 600000.
Final counter value by Process 4: 1100000.
Child with ID: 20602 has just exited.
Child with ID: 20603 has just exited.
Child with ID: 20604 has just exited.
Child with ID: 20605 has just exited.
End of Program.

real    0m0.003s
user    0m0.003s
sys     0m0.000s
```

5. Attempt Overview

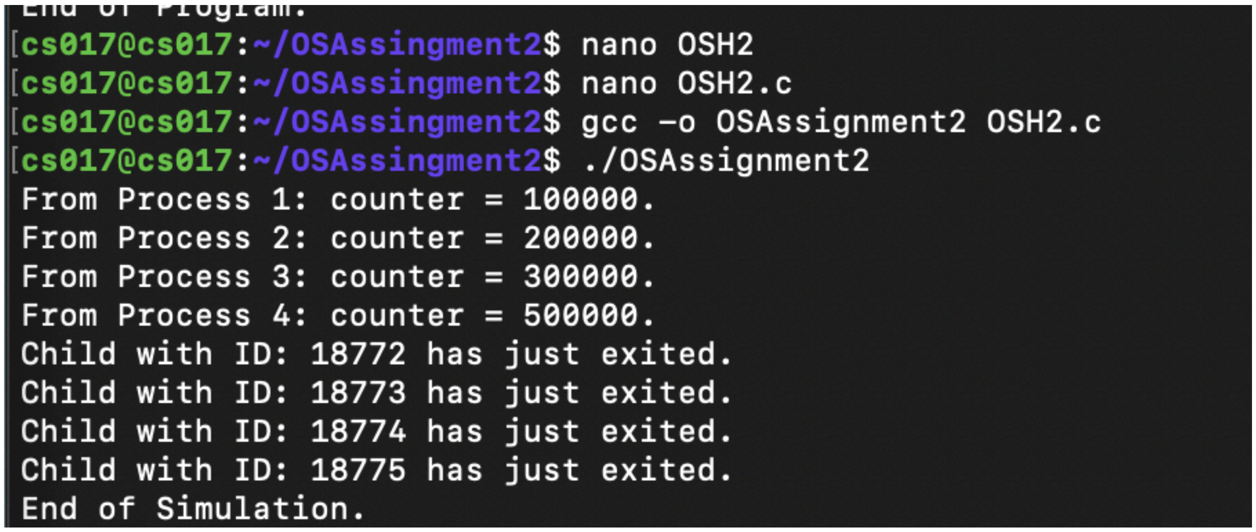
Each attempt illustrates progress in refining semaphore-based synchronization to manage process interactions effectively.

5.1 First Attempt

Identified Issues: Semaphore Initialization Errors
Description: Initially, the semaphores were not properly initialized, resulting in undefined behavior when multiple processes accessed shared memory.
Impact: Without correctly initialized semaphores, processes failed to obtain exclusive access, leading to occasional race conditions. As well as not showing combined shared memory.

5.2 Mid Attempt

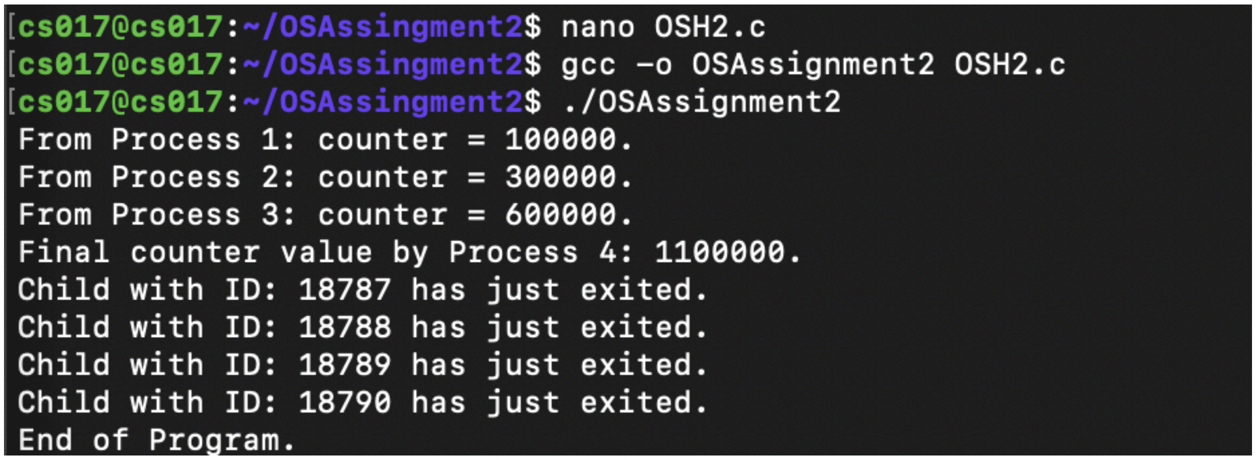
In this attempt, semaphores were fully implemented to control access. Processes correctly followed a wait-and-post sequence, allowing single access to the shared memory location per process cycle. This eliminated observable race conditions, providing consistent and repeatable results. But still not outputting the combined shared memory counter of 1100000 by the 4th counter



```
End of Program.
cs017@cs017:~/OSAssingment2$ nano OSH2
cs017@cs017:~/OSAssingment2$ nano OSH2.c
cs017@cs017:~/OSAssingment2$ gcc -o OSAssignment2 OSH2.c
cs017@cs017:~/OSAssingment2$ ./OSAssignment2
From Process 1: counter = 100000.
From Process 2: counter = 200000.
From Process 3: counter = 300000.
From Process 4: counter = 500000.
Child with ID: 18772 has just exited.
Child with ID: 18773 has just exited.
Child with ID: 18774 has just exited.
Child with ID: 18775 has just exited.
End of Simulation.
```

5.3 Final Attempt (CORRECT OUTPUT)

The final version achieved consistent protection of the shared memory counter, with each process incrementing independently within a semaphore-locked critical section. All previous issues were resolved, resulting in correct final outputs including shared memory of 1100000 by the 4th counter and structured termination of each process.



```
cs017@cs017:~/OSAssingment2$ nano OSH2.c
cs017@cs017:~/OSAssingment2$ gcc -o OSAssignment2 OSH2.c
cs017@cs017:~/OSAssingment2$ ./OSAssignment2
From Process 1: counter = 100000.
From Process 2: counter = 300000.
From Process 3: counter = 600000.
Final counter value by Process 4: 1100000.
Child with ID: 18787 has just exited.
Child with ID: 18788 has just exited.
Child with ID: 18789 has just exited.
Child with ID: 18790 has just exited.
End of Program.
```

6. Conclusion

This assignment effectively highlights the role of semaphores in preventing race conditions in shared memory interactions. By serializing access to a critical section, semaphores enforce data integrity and synchronize process operations. These modifications demonstrate the scalability of POSIX semaphores in managing multi-process environments, especially when critical resource access is required.

Through careful handling of semaphore operations (sem_wait and sem_post), each child process completes without data inconsistency, achieving a correct final count of 1,100,000 in the shared variable. Future optimizations could involve experimenting with alternative synchronization techniques, such as mutexes, to evaluate performance in high-load scenarios.