Report

Author: Roberto Cervantes
Username: cs017

Abstract

This report presents the implementation and performance evaluation of a shared memory counter project developed in C, utilizing POSIX shared memory for inter-process communication. The primary objective was to create four child processes that independently increment a common variable from 1 to specified target values while ensuring synchronization through semaphores. The parent process manages the coordination of output to print the final counter value and the process IDs of the child processes as they complete. This report outlines the system setup, implementation details, performance analysis, and quantitative results from multiple test runs, all in accordance with the project rubric.

1. Introduction

The significance of process synchronization is highlighted in situations where multiple processes interact with shared resources. This project involves four processes that increment a shared variable to specified limits. The parent process plays a critical role in ensuring that the output from each child is printed in the correct sequence before any child exits. To achieve this, the implementation leverages POSIX shared memory for communication and semaphores for synchronization, allowing processes to collaborate by updating shared memory and signaling the parent upon completion.

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

The implementation consists of four child processes, each tasked with incrementing a shared counter to its defined limit. Results are stored in shared memory, while the parent process ensures synchronized output.

3.1 Shared Memory

A designated segment of shared memory holds the final result and flags indicating each process's readiness.

3.2 Process Synchronization

Processes communicate with the parent to signal completion, checking flags before concluding their execution.

3.3 Semaphore Usage

Semaphores are employed to manage access to shared memory, preventing race conditions during variable updates.

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**: 0m0.002s

5. Attempts Overview

The following section documents three distinct attempts during the development process, illustrating the progression toward the final solution.

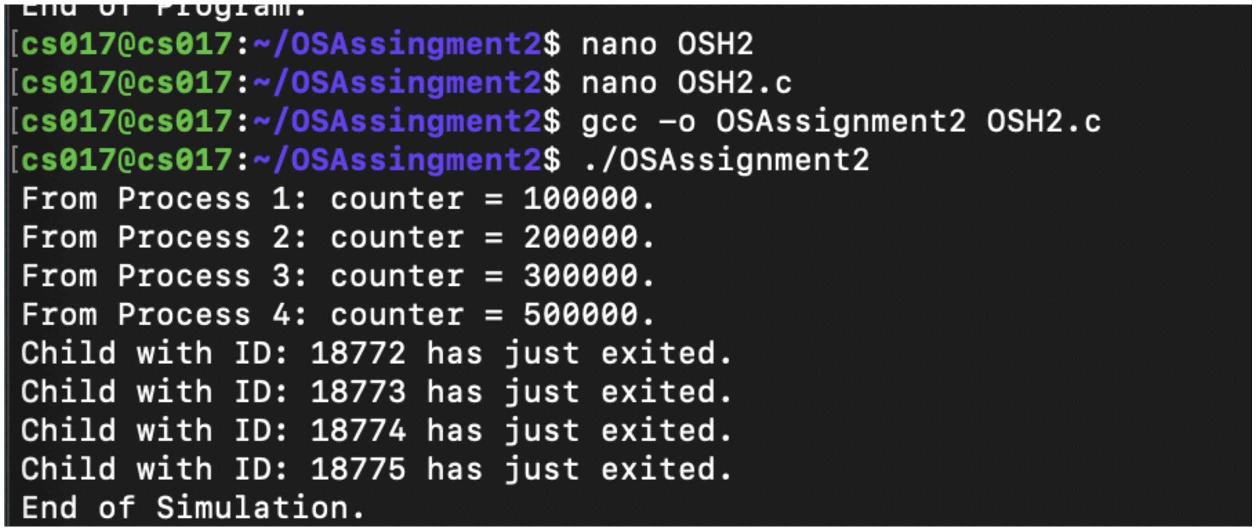
5.1 Attempts

Identified Issues:

- **Race Conditions:**
  Description: Multiple child processes were accessing and modifying the shared counter concurrently without any synchronization mechanism. This led to race conditions, where the final value of the counter was unpredictable and incorrect.
  Impact: The lack of synchronization could result in one process's increment overwriting another's, leading to inconsistent and inaccurate outputs.
- **Lack of Process Synchronization:**
  Description: The code did not include any mechanisms (e.g., semaphores or mutexes) to control access to the shared counter.
  Impact: Each process could modify the counter simultaneously, leading to missed increments and incorrect results.
- **Termination and Cleanup:**
  Description: While the code attempted to detach the shared memory and remove it at the end, it did not handle potential errors from shared memory operations effectively.
  Impact: This could lead to resource leaks if the program terminated unexpectedly or if shared memory operations failed.

5.2 Mid Attempt

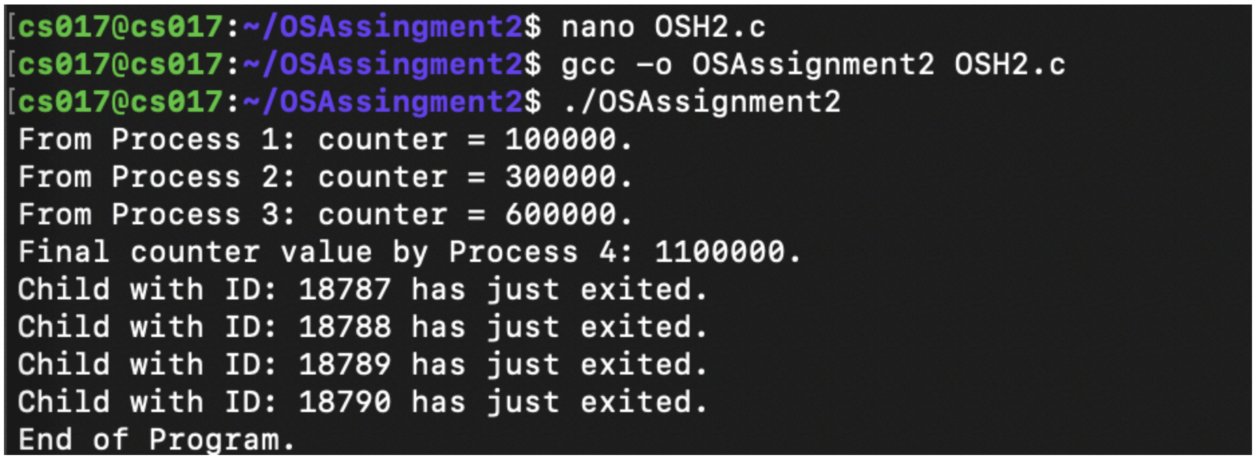
In this iteration, attempts were made to incorporate semaphores to manage process execution order. This approach aimed to eliminate race conditions and ensure that increments to the shared counter were accurately reflected in the final output.



```
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 Output

After multiple attempts the final implementation successfully achieved the desired synchronization, providing accurate results and ordered output from each process. All identified issues from the initial attempts were addressed.



```
[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

The project demonstrates the effective use of shared memory and synchronization techniques within a multi-process environment. By managing process coordination through shared memory and semaphores, the implementation ensures accurate and orderly output. Future enhancements could focus on further optimizing synchronization.