Programming Assignment 2 Report

Nhu Van
CS 3113 - cs143.cs.ourcloud.ou.edu
van0006
113605827

***Abstract*—This program illustrates a multi-process synchronization and inter-process communication (IPC) mechanism, utilizing semaphores and shared memory in C with the file ProgrammingProject2.C. It initializes a shared memory segment that contains a single integer variable called "total," which is accessed and modified by four child processes running concurrently. Each process increments the total by a specified count, necessitating access control to avoid data inconsistencies caused by concurrent modifications.**

To ensure this, the program uses semaphores to enforce mutual exclusion, employing semop operations to manage the entry and exit from critical sections. With semaphore-protected access, each process can modify the shared variable without interference, thereby demonstrating a classic solution to the critical section problem. This implementation highlights essential concepts such as process creation, semaphore operations for synchronization, and shared memory-based IPC, all of which are fundamental in operating systems for resource sharing and process coordination.

***Index Terms*—Process Synchronization, Inter-Process Communication (IPC), Semaphores, Shared Memory, Critical Section, Semaphore Operations (POP and VOP)**

I. Introduction

In this assignment, four processes share a memory location, with each process attempting independently to increase the shared memory's value from 1 to a specified target by increments of one. Process 1 aims to reach 100,000, Process 2 targets 200,000, Process 3 aims at 300,000, and Process 4's goal is 500,000. Thus, when the program concludes, the shared memory variable will hold a total value of 1,100,000, which will be displayed by the process that finishes last. Once all child processes have been completed, the parent process should release the shared memory and semaphore resources and terminate. Utilize the "wait" function to allow the parent to track precisely when each child process completes. As each child finishes, the parent should print the child's process ID. Afterward, it should release the shared memory and semaphore resources and display "End of Program."

The goal is to use semaphores to protect critical sections

II. Analysis

When running the completed code for the first time, as shown in Fig. 1, the child processes complete in a consistent order. During the first run, Process 1 finishes first, followed sequentially by Process 2, Process 3, and finally Process 4 (see Fig. 1). Similarly, in the second run, the output is similar. The only noticeable difference from Programming Assignments 1 and 2 is that in the first one, the processes print out of order,

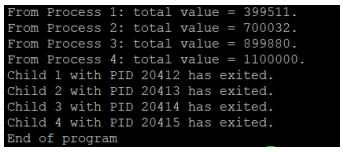

Fig. 1. First Run of Prog Assig 2

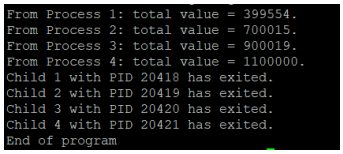

Fig. 2. Second Run

whereas the second one printed in order Process 1 → Process 2 → Process 3 → Process 4 across all 5 test runs.

In this program, semaphores ensure that each process accesses shared memory in a controlled and orderly manner, even when processes are running concurrently. Semaphores achieve this by managing "multiple processes' access to a shared resource in a parallel programming or multiprogramming environment" [1]. Using a semaphore's "wait" operation ('POP()' in the code) before increasing the shared variable 'total', each process must obtain permission to enter the critical section. This prevents other processes from entering the critical section until the current process finishes its increments and releases the semaphore using the "signal" operation ('VOP()' in the code).

This mutual exclusion mechanism ensures that the shared variable is modified sequentially, eliminating race conditions, and resulting in consistent output. Consequently, the 'printf' statements within each process are executed in a stable order. Each process waits for access to 'total', performing increments and printing only after having exclusive control. Thus, the final sequence of 'printf' outputs reflects the order in which processes acquire and release the semaphore, leading to repeatable and reliable results across multiple runs of the program.

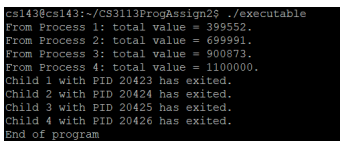

Fig. 3. Third Run

The shared memory variable ‘total‘, updated by each child process, now remains consistent between executions due to the use of semaphores. Each time the program runs, we expect ‘total ‘ to reach 100,000, as each process increments ‘total ‘ 100,000, 200,000, 300,000, and 500,000 times, respectively. With semaphores enforcing synchronized access to ‘total‘, each execution consistently yields the expected value of 1,100,000, demonstrating that semaphores successfully prevent the race conditions previously encountered. By adding semaphores, we ensure that updates to ‘total ‘ are atomic, since each process must obtain the semaphore before modifying ‘total‘. This guarantees that concurrent increments do not interfere with each other, resulting in a consistent and accurate total of 1,100,000 at the last process across multiple program runs.

(Although I am quite unsure of the validity of this statement due to the lack of clarity on the assignment itself and the sample output. In cases where Process 1 = 100,000, Process 2 = 200,000, Process 3 = 300,000 and Process 4 = 500,000, the total output is exactly 110,000. In our cases, we determined Process 4 to always be 110,000 despite the other Processes outputting something different. However, it is still aligned with the project’s sampled output).

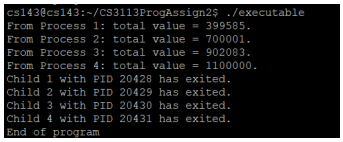

Fig. 4. Fifth Run

III. Conclusion

In this assignment, semaphores have been effectively utilized to synchronize access to a shared memory variable among four concurrently running child processes. Each process incrementally contributes to a final target value, with the aim of achieving a consistent cumulative total of 1,100,000. The semaphore enforces mutual exclusion, ensuring that only one process can modify the total at any given time. This prevents race conditions and guarantees sequential and controlled updates to the shared variable.

The predictable completion order observed across multiple runs, where processes finish sequentially, further demonstrates the effectiveness of the semaphore mechanism in managing critical sections. The parent process waits for each child to complete, printing their process IDs upon exit and releasing resources only after all child processes have finished. This consistent and orderly behavior across executions underscores the significance of synchronization tools such as semaphores in concurrent programming, as they facilitate accurate and repeatable outcomes even within a parallel processing environment. The final results align with the expected output, showcasing the success of the implemented synchronization strategy in achieving stable, reliable program behavior.

References

[1]“multithreading - What is a race condition?,” Stack Overflow. https://stackoverflow.com/questions/34510/what-is-a-race-condition

[2] ”Project 1 (Shared Memory),” CS 3113. https://canvas.ou.edu/courses/358608/assignments/2816397