COP 4600 Project 2

Semaphores

Diya Jain[†]
CSE Department
University of South Florida
Tampa FL USA
diyajain@usf.edu

ABSTRACT

This program demonstrates the use of process synchronization mechanisms in a multi-process environment. It uses shared memory and semaphores to synchronize four child processes, each tasked with incrementing a shared variable to reach specific target values. The program showcases the coordination of these processes and ensures proper resource cleanup upon completion.

KEYWORDS

Process synchronization, shared memory, semaphores, IPC, critical section, multi-process

ACM Reference format:

Diya Jain. 2023. COP 4600 Project 2: Semaphores

1 Introduction

Process synchronization is crucial in multi-process systems to prevent race conditions and ensure orderly access to shared resources. This program exemplifies process synchronization using shared memory and semaphores in a scenario where four child processes independently increment a shared variable to predefined target values. The primary objective is to achieve coordination among these processes to reach the specified targets.

2 Design and Implementation

The program is structured as follows:

1. Constants and Function Prototypes:
    - The program defines constants for shared memory and semaphore keys, the number of child processes, and the total target value.
    - It declares function prototypes for creating shared memory, creating a semaphore, and performing P and V operations (wait and signal) on the semaphore.
    - The `process` function encapsulates the critical section logic for each child process.
2. Main function:
    - The main function initializes shared memory and semaphore.
    - It forks child processes, assigning each a specific target value.
    - Child processes execute the `process` function to increment the shared variable.
    - The parent process waits for the child processes to complete and prints their exit status.
3. Shared Memory and Semaphore Initialization:
    - The `createSharedMemory` function creates a shared memory segment for the shared variable.
    - The `createSemaphore` function creates a semaphore and initializes its value to 1.
4. Process Synchronization:
    - Child processes use semaphores to enter and exit the critical section, ensuring exclusive access to the shared variable.
    - They increment the shared variable within the critical section until the target value is reached.

3 Results and Observation

The program was executed multiple times, and the following results were observed:

1. Each child process independently increments the shared variable to its designated target.
2. The program prints the process ID and the current value of the shared variable as each child process exits.
3. The parent process waits for all child processes to complete and then releases shared memory and semaphore resources.
4. The final value of the shared variable is correctly computed as the sum of the individual targets.

Sample Output:

```
From Process 1: counter = 2750013.
Child with ID 6880 has just exited.
From Process 2: counter = 5500022.
Child with ID 6881 has just exited.
From Process 3: counter = 8250002.
Child with ID 6882 has just exited.
From Process 4: counter = 11000000.
Child with ID 6883 has just exited.
End of Simulation.
```

Figure 1: Sample Output from the ./executable file.

4 Conclusion

This program successfully demonstrates process synchronization using shared memory and semaphores in a multi-process environment. It showcases the importance of proper synchronization mechanisms to avoid race conditions and ensure orderly access to shared resources. The program's design and implementation provide a clear example of coordinating multiple processes to achieve a common goal while maintaining data integrity and proper resource cleanup.