****

**Assignment on Process Synchronizations**

**Submitted To**

**Faculty Name :** **Prof. Dr. Md. Motaharul Islam,**

**Professor, Department of CSE**

**Submitted By**

**Name : Sheikh Sarafat Hossain**

**Id No : 2022-3-60-109**

**Course Title :** Operating System

**Course Code :** CSE325

**Section :** 07

**Semester :** Fall 24

**Date of Submission:** 12 January 2025

**Process Synchronization**

**Overview**

This report analyzes the implementation and output of a shared memory-based program to demonstrate process synchronization. The program utilizes shared memory for communication between processes and employs the fork() system call to create multiple processes.

**Program Description**

The program creates a shared memory segment that is accessed by two child processes, Process1 and Process2. The shared memory is initialized to 1. Each process performs the following tasks:

* **Process1** increments the shared variable.
* **Process2** decrements the shared variable.

The program simulates potential race conditions by introducing delays (sleep calls), which emulate context switching and uncoordinated access to the shared memory.

**Key Steps in the Program**

1. **Shared Memory Initialization**:
   * A unique key is generated using ftok().
   * Shared memory is created with shmget() and attached to the processes with shmat().
2. **Process Creation**:
   * Two processes are created using fork().
   * Process1 executes the increment operation.
   * Process2 executes the decrement operation.
3. **Synchronization Issue Simulation**:
   * Both processes perform local operations on the shared variable with delays to showcase the effects of unsynchronized access.
4. **Final Output**:
   * The parent process waits for the child processes to complete and prints the final value of the shared variable.

**Observed Output**

The output of the program is as follows:

A screenshot of a computer program

Description automatically generated

**Explanation of the Output**

1. **Initial Reads**:
   * Both processes read the initial value of the shared variable (1).
2. **Local Updates**:
   * Process2 decrements its local copy of the shared variable to 0.
   * Simultaneously, Process1 increments its local copy to 2.
3. **Shared Memory Updates**:
   * Process2 writes 0 to the shared variable.
   * Subsequently, Process1 overwrites this with 2.
4. **Final Value**:
   * The final value of the shared variable is 2, reflecting the last write operation by Process1.

**Analysis of Race Condition**

The output illustrates a race condition caused by unsynchronized access to the shared memory. Both processes independently read and update the shared variable without coordination, leading to inconsistent intermediate states. The observed behavior depends on the execution order of processes, which is non-deterministic.

**Recommendations for Synchronization**

To avoid race conditions and ensure consistency in shared memory operations, synchronization mechanisms can be employed:

1. **Mutex Locks**:
   * Use a mutex to allow only one process to access the shared variable at a time.
2. **Semaphores**:
   * Implement semaphores to control access to the critical section.
3. **Atomic Operations**:
   * Utilize atomic operations to ensure that reads, updates, and writes occur as indivisible actions.

**Conclusion**

The program effectively demonstrates the challenges of process synchronization using shared memory. The observed race condition highlights the importance of implementing synchronization mechanisms to achieve consistent and predictable results in concurrent programming.