**Batch: D - 1 Roll No.: 16010122096**

**Experiment No. 07**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **TITLE:** Implementation of Process synchronization algorithms using thread - producer consumer problem , reader-writers problem**.** |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**AIM:** Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

**CO 3.** To understand the concepts of process synchronization and deadlock.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. “Operating Systems Principles”, Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate “Operating Systems”, McGraw Hill Third Edition.**
3. **Sumitabha Das “ UNIX Concepts & Applications”, McGraw Hill Second**

**Edition.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Knowledge of Concurrency, Synchronization, threads.

# Description of the chosen process synchronization algorithm:

# The Producer-Consumer Problem is a synchronization problem where producers generate data and place it in a shared buffer, while consumers retrieve and use the data. The buffer has limited capacity, so the producer must wait if it's full, and the consumer must wait if it's empty. To handle synchronization, semaphores or locks are used: a mutex to ensure mutual exclusion when accessing the buffer, and two additional semaphores—full and empty—to track the state of the buffer. This ensures that the producer and consumer don't interfere with each other, preventing race conditions and deadlocks.

**Implementation details:**

*#include* <iostream>

*#include* <vector>

*#include* <thread>

*#include* <cstdlib>

*#include* <chrono>

using namespace std;

const int sz = 5;

const int mx = 10, mn = 1;

vector<int> v(sz);

int in = 0, out = 0;

void produce() {

*while* (true) {

        int num = rand() % (mx - mn + 1) + mn;

*while* ((in + 1) % sz == out) {

            cout << "Buffer Full" << endl;

            this\_thread::sleep\_for(chrono::milliseconds(100));

        }

        v[in] = num;

        cout << num << " Produced" << endl;

        in = (in + 1) % sz;

        this\_thread::sleep\_for(chrono::milliseconds(rand() % 7000));

    }

}

void consume() {

*while* (true) {

*while* (in == out) {

            cout << "Buffer Empty" << endl;

            this\_thread::sleep\_for(chrono::milliseconds(100));

        }

        int x = v[out];

        cout << "Consumed " << x << endl;

        out = (out + 1) % sz;

        this\_thread::sleep\_for(chrono::milliseconds(rand() % 7000));

    }

}

int main() {

    thread prod\_thread(produce);

    thread cons\_thread(consume);

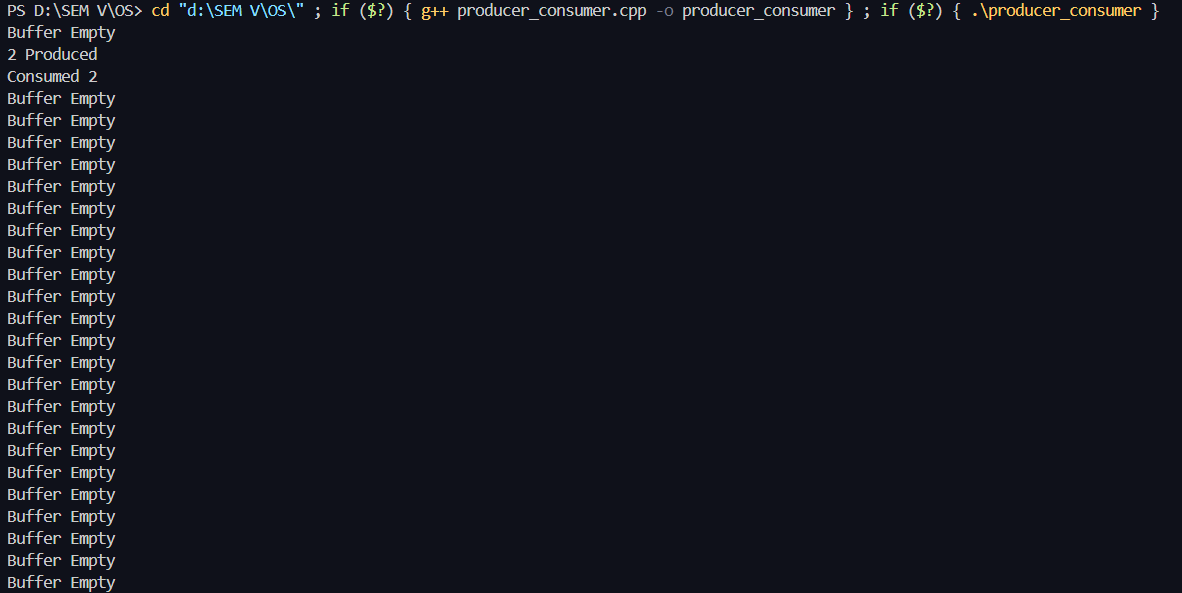
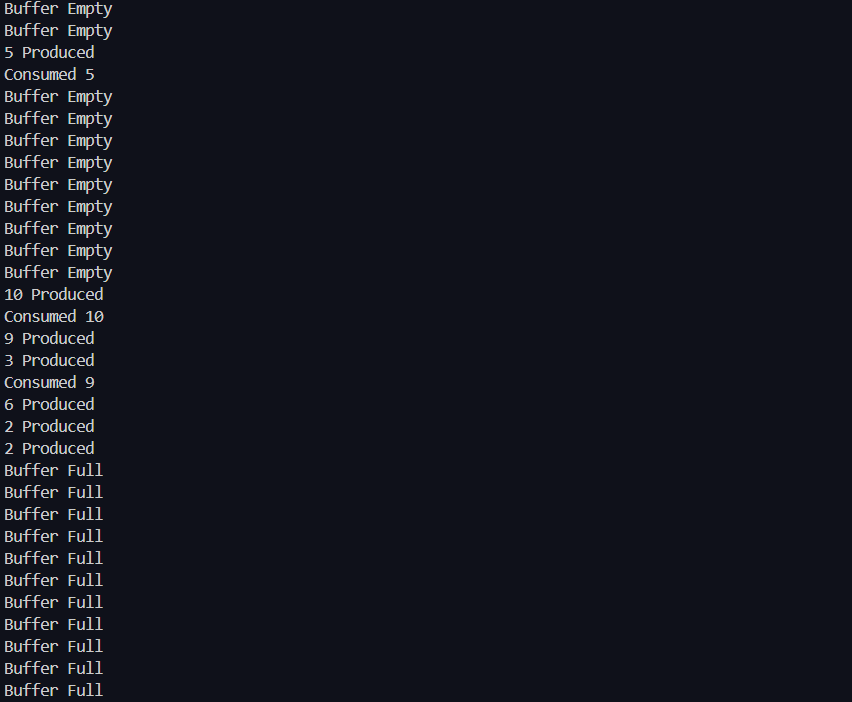
    prod\_thread.join();

    cons\_thread.join();

*return* 0;

}

**Output:**

**Conclusion:**

Proper synchronization in concurrent programming is essential to prevent data corruption, deadlocks, and ensure efficient resource access among threads.

**Post Lab Descriptive Questions**

1. Differentiate between a monitor, semaphore and a binary semaphore?

 **Monitor**: A monitor is a higher-level synchronization construct that encapsulates shared variables, the procedures that operate on those variables, and the synchronization required to protect them. It allows only one thread to execute a monitor procedure at a time, automatically handling mutual exclusion and condition synchronization.

 **Semaphore**: A semaphore is a low-level synchronization primitive that uses integer values to control access to shared resources. It can be classified as a counting semaphore, which allows a certain number of threads to access the resource concurrently, or a binary semaphore, which can only take values 0 or 1.

 **Binary Semaphore**: A binary semaphore is a specific type of semaphore that can only have two values: 0 and 1. It is often used to implement mutual exclusion, functioning similarly to a mutex. A binary semaphore allows one thread to access the shared resource while others wait.

1. Producer-Consumer Problem:
   1. What would happen if the mutex semaphore was not used in the producer-consumer implementation?
   2. How can the buffer size affect the performance of the producer-consumer system?
   3. What are the potential issues if the producer and consumer threads are not properly synchronized?

**a.** If the mutex semaphore was not used, both the producer and consumer could access the shared buffer concurrently. This could lead to race conditions where multiple threads modify the buffer simultaneously, resulting in inconsistent or corrupted data. For example, a producer might overwrite data that a consumer is currently reading, leading to data loss and unpredictable behavior.

**b.** The buffer size directly impacts the efficiency of the producer-consumer system. A small buffer can lead to frequent blocking of the producer when it is full and of the consumer when it is empty, causing underutilization of resources. Conversely, a larger buffer may reduce blocking but can increase latency in processing as data takes longer to travel through the system. Thus, the optimal buffer size balances throughput and latency, maximizing performance.

**c.** If the producer and consumer threads are not properly synchronized, several issues may arise:

* **Data Corruption**: Race conditions can lead to data being incorrectly written or read.
* **Deadlocks**: Improper synchronization can cause threads to block indefinitely, waiting for each other to release resources.
* **Starvation**: One thread may monopolize access to the buffer, preventing the other from proceeding.
* **Inconsistent State**: The system may reach an inconsistent state where the data in the buffer does not reflect the expected results of producer and consumer operations.

1. Reader-Writers Problem:
   1. Explain the importance of the rw\_mutex semaphore in the reader-writers problem.
   2. How does the implementation ensure that writers get exclusive access to the shared resource?
   3. What modifications would you make to prioritize writers over readers?

**a.** The rw\_mutex semaphore is crucial in the reader-writers problem as it allows multiple readers to access the shared resource simultaneously while ensuring that writers have exclusive access. This prevents data inconsistency that could arise from simultaneous write and read operations. The rw\_mutex effectively manages read and write locks, ensuring that the reading and writing processes do not interfere with each other.

**b.** The implementation ensures that writers get exclusive access by using the rw\_mutex to block all readers when a writer is active. When a writer wants to access the resource, it acquires the write lock, which prevents any readers from accessing the resource until the writer has finished. This guarantees that the writer can modify the resource without being interfered with by any reader threads.

**c.** To prioritize writers over readers, the following modifications can be made:

* Introduce a mechanism to count the number of waiting writers and allow them to acquire the lock before any new readers are granted access.
* Modify the condition checks in the reader and writer implementations to ensure that if a writer is waiting, new readers should be blocked until the writer has completed its operation.
* Implement a waiting queue for readers and writers, where writers are given preference in acquiring the lock if there are both readers and writers waiting. This can prevent reader starvation and ensure that critical writes are not delayed by ongoing reads.

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**