K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: D-2	**Roll No.:** 16010122151
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:

Implementation details:

Producer-Consumer Problem

The **Producer-Consumer problem** is an example of synchronization where two types of threads are involved:

- **Producer**: Produces data (items) and stores them in a shared buffer.
- **Consumer**: Consumes data (items) from the buffer.

The main challenge is that the producer and consumer must synchronize their actions because:

- The producer should not add data to the buffer if it is full.
- The consumer should not consume data from the buffer if it is empty.

This creates the need for synchronization mechanisms like semaphores or mutexes to ensure mutual exclusion and proper coordination.

Reader-Writer Problem

The **Reader-Writer Problem** involves two types of threads:

- **Readers**: Threads that read shared data.
- **Writers**: Threads that write to shared data.

The challenge here is that multiple readers can read the data simultaneously without any issues, but when a writer writes to the data, it must have exclusive access to it, meaning no other reader or writer can access the data at the same time.

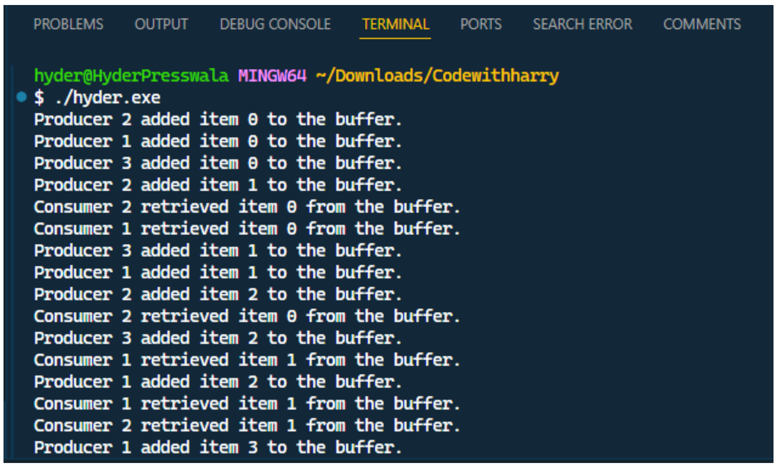
The main synchronization issues arise when:

- Writers must be given exclusive access to the shared resource.
- Multiple readers can access the resource concurrently, but no writer should be allowed to access the data while it is being read or written to by others.

Producer-Consumer Problem

```cpp
#include <iostream>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <vector>

std::queue<int> buffer;
const unsigned int MAX_SIZE = 5; // Maximum buffer size
std::mutex mtx;
std::condition_variable cv_empty, cv_full;

// Producer function
void producer(int id) {
    for (int i = 0; i < 5; ++i) { // Each producer produces 5 items
        std::this_thread::sleep_for(std::chrono::milliseconds(500));

        // Locking the mutex before accessing the shared buffer
        std::unique_lock<std::mutex> lock(mtx);

        // Wait if buffer is full
        cv_full.wait(lock, []() { return buffer.size() <
MAX_SIZE; });

        // Produce an item
        buffer.push(i);
        std::cout << "Producer " << id << " added item "
<< i << " to the buffer.\n";

        // Notify consumer that the buffer is not empty
        cv_empty.notify_all();
    }
}

// Consumer function
void consumer(int id) {
    for (int i = 0; i < 5; ++i) { // Each consumer
consumes 5 items
        std::this_thread::sleep_for(std::chrono::millisec
onds(1000));

        // Locking the mutex before accessing the shared
buffer
        std::unique_lock<std::mutex> lock(mtx);

        // Wait if buffer is empty
        cv_empty.wait(lock, []() { return
!buffer.empty(); });

        // Consume an item
        int item = buffer.front();
        buffer.pop();
        std::cout << "Consumer " << id << " retrieved
item " << item << " from the buffer.\n";

        // Notify producer that the buffer is not full
        cv_full.notify_all();
    }
}

int main() {
    int num_producers = 3; // Number of producers
    int num_consumers = 2; // Number of consumers

    std::vector<std::thread> producers;
    std::vector<std::thread> consumers;

    // Launch producer threads
    for (int i = 0; i < num_producers; ++i) {
        producers.push_back(std::thread(producer, i +
1)); // Producer IDs start from 1
    }

    // Launch consumer threads
    for (int i = 0; i < num_consumers; ++i) {
        consumers.push_back(std::thread(consumer, i +
1)); // Consumer IDs start from 1
    }

    // Join all threads
    for (auto& th : producers) {
        th.join();
    }
    for (auto& th : consumers) {
        th.join();
    }

    return 0;
}
```

Output :-



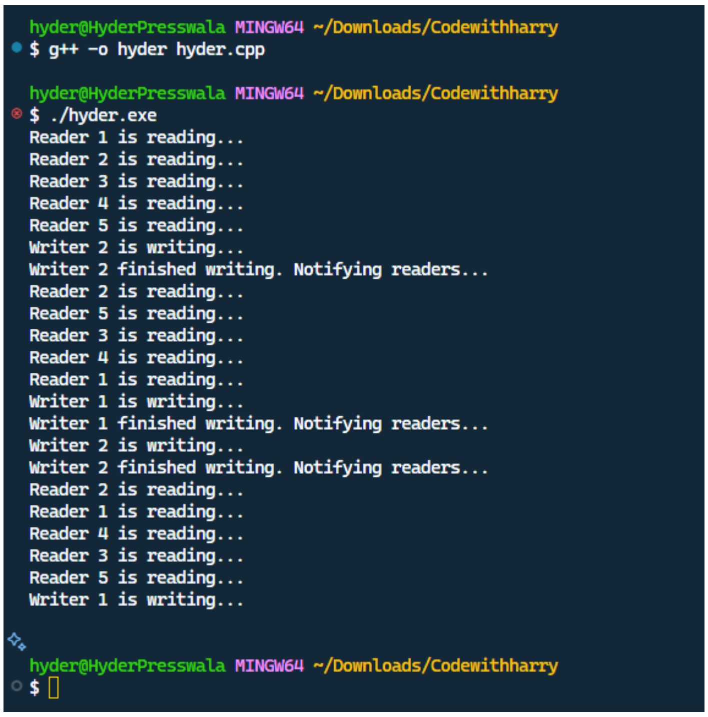
```
hyder@HyderPresswala MINGW64 ~/Downloads/Codewithharry
$ ./hyder.exe
Producer 2 added item 0 to the buffer.
Producer 1 added item 0 to the buffer.
Producer 3 added item 0 to the buffer.
Producer 2 added item 1 to the buffer.
Consumer 2 retrieved item 0 from the buffer.
Consumer 1 retrieved item 0 from the buffer.
Producer 3 added item 1 to the buffer.
Producer 1 added item 1 to the buffer.
Producer 2 added item 2 to the buffer.
Consumer 2 retrieved item 0 from the buffer.
Producer 3 added item 2 to the buffer.
Consumer 1 retrieved item 1 from the buffer.
Producer 1 added item 2 to the buffer.
Consumer 1 retrieved item 1 from the buffer.
Consumer 2 retrieved item 1 from the buffer.
Producer 1 added item 3 to the buffer.
```

Reader-Writer Problem [Using Mutex]

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <vector>

// Shared resources and synchronization mechanisms
std::mutex mtx;  // Mutex for protecting critical section
std::condition_variable cv;  // Condition variable to
manage synchronization
std::atomic<int> reader_count(0);  // Number of active
readers
bool writing = false;  // Flag to indicate if a writer is
writing

// Function for the Reader thread
```

```cpp
void reader(int id) {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);  // Lock the mutex

        // Wait if a writer is writing
        cv.wait(lock, []() { return !writing; });

        reader_count++;  // Increment the reader count
        std::cout << "Reader " << id << " is reading..." << std::endl;

        lock.unlock();  // Unlock the mutex to allow other threads

        // Simulate reading
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

        lock.lock();  // Lock again to decrement reader count
        reader_count--;

        if (reader_count == 0) {
            // If this is the last reader, notify any waiting writers
            cv.notify_all();
        }

        lock.unlock();

        // Simulate time between reading tasks
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }
}
```

```cpp
// Function for the Writer thread
void writer(int id) {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);  // Lock the mutex

        // Wait until there are no active readers and no writer is writing
        cv.wait(lock, []() { return reader_count == 0 && !writing; });

        writing = true;  // Mark that the writer is writing
        std::cout << "Writer " << id << " is writing..." << std::endl;

        lock.unlock();  // Unlock the mutex while writing

        // Simulate writing
        std::this_thread::sleep_for(std::chrono::milliseconds(1500));

        lock.lock();  // Lock the mutex again after writing
        writing = false;  // Mark that writing is finished
        std::cout << "Writer " << id << " finished writing. Notifying readers..." << std::endl;

        // Notify all readers that the writer is done
        cv.notify_all();

        lock.unlock();  // Unlock the mutex

        // Simulate time between writing tasks
        std::this_thread::sleep_for(std::chrono::millisec
onds(2000));
    }
}

int main() {
    int num_readers = 5;
    int num_writers = 2;

    // Create threads for readers and writers
    std::vector<std::thread> reader_threads;
    std::vector<std::thread> writer_threads;

    // Launch reader threads
    for (int i = 0; i < num_readers; ++i) {
        reader_threads.push_back(std::thread(reader, i +
1));
    }

    // Launch writer threads
    for (int i = 0; i < num_writers; ++i) {
        writer_threads.push_back(std::thread(writer, i +
1));
    }

    // Join the threads to the main thread
    for (auto& th : reader_threads) {
        th.join();
    }
    for (auto& th : writer_threads) {
        th.join();
    }

    return 0;
}
```



```
hyder@HyderPresswala MINGW64 ~/Downloads/Codewithharry
$ g++ -o hyder hyder.cpp

hyder@HyderPresswala MINGW64 ~/Downloads/Codewithharry
$ ./hyder.exe
Reader 1 is reading...
Reader 2 is reading...
Reader 3 is reading...
Reader 4 is reading...
Reader 5 is reading...
Writer 2 is writing...
Writer 2 finished writing. Notifying readers...
Reader 2 is reading...
Reader 5 is reading...
Reader 3 is reading...
Reader 4 is reading...
Reader 1 is reading...
Writer 1 is writing...
Writer 1 finished writing. Notifying readers...
Writer 2 is writing...
Writer 2 finished writing. Notifying readers...
Reader 2 is reading...
Reader 1 is reading...
Reader 4 is reading...
Reader 3 is reading...
Reader 5 is reading...
Writer 1 is writing...

hyder@HyderPresswala MINGW64 ~/Downloads/Codewithharry
$ 
```

Conclusion :- The Reader-Writer problem and the Producer-Consumer problem both teach us the importance of thread synchronization techniques, such as mutexes, semaphores, and condition variables

Post Lab Descriptive Questions

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

A monitor is a higher-level synchronization construct that encapsulates shared variables and operations with built-in mutual exclusion, a semaphore is a low-level synchronization primitive that controls access to shared resources through a counter (allowing multiple access), and a binary semaphore is a special type of semaphore that can only take values of 0 or 1, effectively functioning as a mutex to allow exclusive access.

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

a. Without the mutex semaphore, simultaneous access to the shared buffer could lead to race conditions, resulting in data corruption and unpredictable behavior.

b. The buffer size can significantly impact performance, as a larger buffer may increase throughput by allowing more items to be produced and consumed before blocking, while a smaller buffer may lead to increased waiting time and inefficiencies.

c. Improper synchronization between producer and consumer threads can result in potential issues such as buffer overflow, where producers attempt to add items to a full buffer, or underflow, where consumers try to remove items from an empty buffer.

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

a. The rw_mutex semaphore is crucial in the reader-writers problem because it manages the access control to the shared resource, ensuring that multiple readers can read concurrently while preventing writers from accessing the resource simultaneously.

b. The implementation ensures that writers get exclusive access by having them acquire the write_lock semaphore, which blocks all readers and other writers until the writing is complete. c. To prioritize writers over readers, you could implement a queue mechanism that allows waiting writers to be served first or modify the conditions for granting access such that writers are allowed to proceed when they request access, even if there are active readers

Date: 06-11-2024 **Signature of faculty in-charge**