Department of Computer Engineering

Batch: A1	**Roll No.: 16010123012**
Experiment No. 07	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: Readers-Writers problem using semaphore/monitors.

AIM: Implementation of Process synchronization algorithm using semaphore to solve Reader-writers problem

Expected Outcome of Experiment:

CO3: 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. **William Stallings, "Operating System Internal & Design Principles", Pearson.**

4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

The readers-writer problem is about managing access to shared data. It allows multiple readers to read data at the same time without issues but ensures that only one writer can write at a time, and no reader can read while the writer is writing. This helps prevent data corruption and ensures smooth concurrent operation of multiple processes.

The readers/writers problem is stated as follows:

There is a data area shared among a number of processes.

The data area could be a file, a block of main memory or even a bank of processor registers.

There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.

2. Only one writer at a time may write to the file.

3. If a writer is writing to the file, no reader may read it.

The reader-writer problem can be thought of for two different scenarios; priority to reader or priority to writer. The solution to the problem can be proposed using semaphore or monitors.

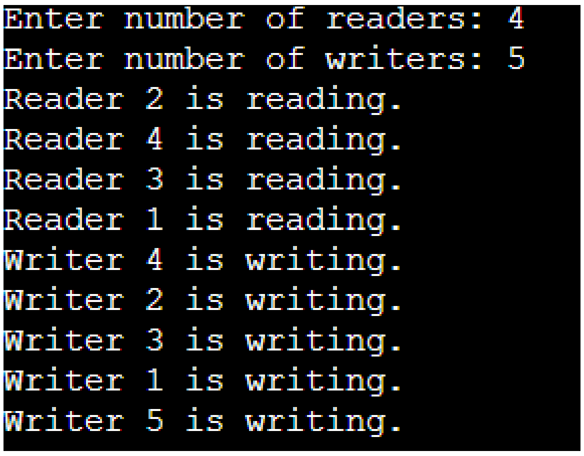
Implementation details:

DATA STRUCTURES Used/ Code
int (readCount)
bool (writerActive)
std::mutex (mtx)
std::condition_variable (cv)
std::vector<std::thread>

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <vector>
#include <chrono>
using namespace std;

mutex mtx;
condition_variable cv;
int readCount = 0;
bool writerActive = false;

void reader(int id)
{
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, []
            { return !writerActive; });
    readCount++;
    lock.unlock();
```

```cpp
    cout << "Reader " << id << " is reading." << endl;
    this_thread::sleep_for(chrono::seconds(1));

    lock.lock();
    readCount--;
    if (readCount == 0)
        cv.notify_all();
    lock.unlock();
}

void writer(int id)
{
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, []
            { return readCount == 0 && !writerActive; });
    writerActive = true;
    lock.unlock();

    cout << "Writer " << id << " is writing." << endl;
    this_thread::sleep_for(chrono::seconds(1));

    lock.lock();
    writerActive = false;
    cv.notify_all();
    lock.unlock();
}

int main()
{
    int r, w;
    cout << "Enter number of readers: ";
    cin >> r;
    cout << "Enter number of writers: ";
    cin >> w;

    vector<thread> readers;
    vector<thread> writers;

    for (int i = 0; i < r; ++i)
    {
        readers.emplace_back(reader, i + 1);
    }
    for (int i = 0; i < w; ++i)
    {
        writers.emplace_back(writer, i + 1);
    }

    for (auto &t : readers)
    {
        t.join();
    }

    for (auto &t : writers)
    {
        t.join();
    }
    return 0;
}
```

Output:



```
Enter number of readers: 4
Enter number of writers: 5
Reader 2 is reading.
Reader 4 is reading.
Reader 3 is reading.
Reader 1 is reading.
Writer 4 is writing.
Writer 2 is writing.
Writer 3 is writing.
Writer 1 is writing.
Writer 5 is writing.
```

Conclusion:
I have successfully completed the implementation of the Readers-Writers problem using monitor-based synchronization with mutex and condition_variable in C++. This experiment helped me understand the core concepts of process synchronization, particularly how multiple readers can safely access shared data concurrently while ensuring that writers gain exclusive access when writing. The use of condition variables effectively managed thread coordination, preventing race conditions and maintaining data consistency.

Post Lab Objective Questions

1) A semaphore which takes only binary values is called

a) Counting semaphore
b) Mutex
c) Weak semaphore
d) None of the above
Ans: b) Mutex

2) Mutual exclusion can be provided by the
a) Mute locks
b) Binary semaphores
c) Both a and b
d) None of these
Ans: c) Both a and b

3) A monitor is a module that encapsulates
a) Shared data structures
b) Procedures that operate on shared data structure
c) Synchronization between concurrent procedure invocation
d) All of the above
Ans: d) All of the above

4) To enable a process to wait within the monitor
a) A condition variable must be declared as condition
b) Condition Variables must be used as Boolean objects
c) Semaphore must be used
d) All of the above
Ans: a) A condition variable must be declared as condition

Post Lab Subjective Questions

1. What is Monitor? Explain how the monitor supports synchronization by the use of condition variables.

A monitor is a high-level synchronization construct used in concurrent programming to manage access to shared resources safely and efficiently. It encapsulates shared data and the procedures that operate on it, ensuring that only one process or thread can execute within the monitor at any given time, thus providing automatic mutual exclusion. To support synchronization among concurrent threads, monitors use condition variables, which allow threads to wait for specific conditions to be met before proceeding. The wait() operation causes a thread to suspend its execution and release the monitor lock, enabling other threads to enter the monitor. Once the condition is fulfilled, another thread can use the signal() operation to wake up one of the waiting threads, allowing it to re-enter the monitor and resume execution. This mechanism ensures coordinated access to shared resources and prevents race conditions, making monitors a powerful abstraction for safe concurrent programming.

Date: 15 / 04 / 2025 **Signature of faculty in-charge**