Report

Lab 3: Implementation of Lamport Logical Clock

Aim:

To implement Lamport's logical clock algorithm in a multi-threaded environment where two processes (threads) exchange messages and perform internal events to synchronize logical clocks.

Theory:

In distributed systems, where multiple processes operate concurrently and communicate through message passing, the challenge of maintaining a consistent order of events across all processes arises. Unlike in centralized systems, where a single clock can keep time, distributed systems lack a global clock, making it difficult to determine the exact order of events. This is where Lamport's logical clocks come into play.

Lamport Logical Clocks are a mechanism introduced by Leslie Lamport in 1978 to order events in a distributed system. The key idea is to assign a logical timestamp to each event, ensuring that if one event causally affects another, the first event is assigned a smaller timestamp than the second. This way, the system can maintain a consistent order of events across different processes without relying on synchronized physical clocks.

How Lamport Logical Clocks Work:

1. **Initialization**: Each process in the system maintains its own logical clock, initialized to 0.
2. **Internal Events**: When a process performs an internal event (e.g., computation or data processing), it increments its logical clock by 1.
3. **Sending Messages**: When a process sends a message to another process, it increments its logical clock by 1, timestamps the message with the updated clock value, and then sends the message.
4. **Receiving Messages**: Upon receiving a message, a process compares its own logical clock with the timestamp of the received message. The receiving process updates its clock to the maximum of its current clock value and the received timestamp and then increments its clock by 1. This ensures that the event of receiving the message is ordered after the event of sending it.

Example Scenario:

Consider two processes, **P1** and **P2**, communicating in a distributed system:

1. **P1** starts with a logical clock value of 0 and performs an internal event, incrementing its clock to 1.
2. **P1** sends a message to **P2** with the timestamp 1. **P2** receives this message when its clock is at 0.

3. **P2** compares its clock with the message's timestamp. Since the message's timestamp (1) is greater, **P2** updates its clock to 1 and increments it to 2, ensuring that the receive event is ordered after the send event.
4. **P2** then performs an internal event, incrementing its clock to 3.
5. **P2** sends a message back to **P1** with the timestamp 3. **P1** receives this message, compares its current clock (still at 1) with the received timestamp (3), and updates its clock to 3, then increments it to 4.

By the end of this exchange, both processes have logically ordered the events:

- **P1**'s events: Internal (1), Send (1), Receive (4)
- **P2**'s events: Receive (2), Internal (3), Send (3)

This logical ordering respects the causality of events: the send from **P1** occurs before the receive by **P2**, and similarly, the send from **P2** occurs before the receive by **P1**. This example illustrates how Lamport logical clocks ensure that all processes in a distributed system can agree on the order of events, even in the absence of synchronized physical clocks.

Significance:

The use of Lamport logical clocks is crucial in various distributed system algorithms, including mutual exclusion, snapshot algorithms, and consistent state recording. They provide a foundation for understanding and reasoning about the order of events in systems where physical clock synchronization is impractical or impossible. By using logical timestamps, processes can ensure that causally related events are correctly ordered, preventing inconsistencies in the system's behavior.

Function & Modules Used:

std::mutex: Ensures that shared data structures like message buffers are safely accessed by multiple threads.

std::condition_variable: Manages the waiting and notification between threads to handle the receipt of messages.

std::queue<int>: Stores messages in a buffer for each thread.

std::thread: Represents each process and allows them to run concurrently.

std::vector<int> logical_clocks: Stores the logical clock values for each thread.

std::map<int, std::queue<int>> message_buffers: Buffers that hold messages to be processed by each thread.

Code with Comments:

```
#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

#include <condition_variable>

#include <random>

#include <chrono>

#include <queue>

#include <map>


std::mutex mtx;

std::condition_variable cv;

std::map<int, std::queue<int>> message_buffers;

std::vector<int> logical_clocks;

std::vector<std::mutex*> buffer_mutexes;


void print_event_order(int thread_id, const std::vector<int>& event_order) {

        std::cout << "Thread " << thread_id << " event order: ";

        for (int event : event_order) {

        std::cout << event << " ";

        }

        std::cout << std::endl;

}
```

```
void thread_func(int thread_id, int num_iterations) {
        std::vector<int> event_order;
        int& logical_clock = logical_clocks[thread_id];
        std::queue<int> local_buffer;

        std::default_random_engine generator;
        std::uniform_int_distribution<int> distribution(0, 2);

        for (int i = 0; i < num_iterations; ++i) {
        // Internal event
        std::this_thread::sleep_for(std::chrono::milliseconds(distribution(generator) * 100));
        logical_clock++;
        event_order.push_back(logical_clock);

        // Sending message
        int recipient_id = (thread_id + 1) % logical_clocks.size();
        {
        std::lock_guard<std::mutex> lock(*buffer_mutexes[recipient_id]);
        message_buffers[recipient_id].push(logical_clock);
        }
        cv.notify_all();

        // Receiving message
        std::unique_lock<std::mutex> lock(mtx);
```

```
            cv.wait(lock, [&] { return !message_buffers[thread_id].empty(); });


            std::lock_guard<std::mutex> buffer_lock(*buffer_mutexes[thread_id]);
            while (!message_buffers[thread_id].empty()) {
            int received_clock = message_buffers[thread_id].front();
            message_buffers[thread_id].pop();
            logical_clock = std::max(logical_clock, received_clock) + 1;
            event_order.push_back(logical_clock);
            }


            print_event_order(thread_id, event_order);
            }
}


int main() {
            int num_threads = 2;  // Updated to use 2 processes (threads)
            int num_iterations = 5;


            logical_clocks.resize(num_threads, 0);


            // Initialize message_buffers map with empty queues
            for (int i = 0; i < num_threads; ++i) {
            message_buffers[i] = std::queue<int>();
            }
```

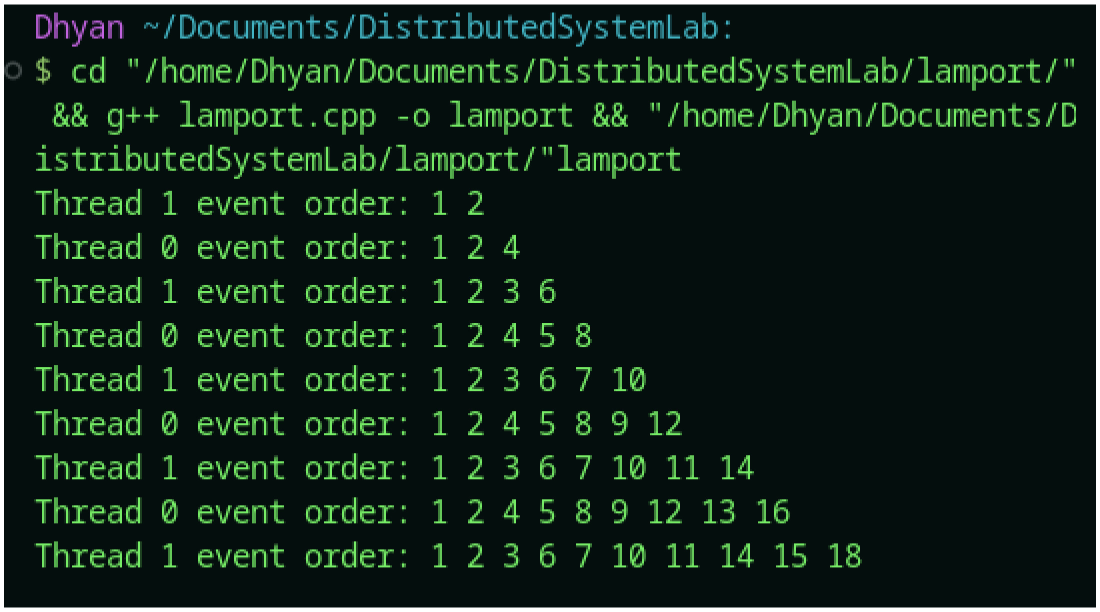
```
        // Initialize buffer_mutexes with pointers to std::mutex objects
        std::vector<std::mutex*> mutex_pointers;
        for (int i = 0; i < num_threads; ++i) {
        std::mutex* mutex_ptr = new std::mutex();
        mutex_pointers.push_back(mutex_ptr);
        }
        buffer_mutexes = mutex_pointers;

        std::vector<std::thread> threads;
        for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(thread_func, i, num_iterations);
        }

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

        // Clean up dynamically allocated mutexes
        for (auto mutex_ptr : buffer_mutexes) {
        delete mutex_ptr;
        }
        return 0;
}
```

Output:


```
Dhyan ~/Documents/DistributedSystemLab:
$ cd "/home/Dhyan/Documents/DistributedSystemLab/lamport/"
 && g++ lamport.cpp -o lamport && "/home/Dhyan/Documents/D
istributedSystemLab/lamport/"lamport
Thread 1 event order: 1 2
Thread 0 event order: 1 2 4
Thread 1 event order: 1 2 3 6
Thread 0 event order: 1 2 4 5 8
Thread 1 event order: 1 2 3 6 7 10
Thread 0 event order: 1 2 4 5 8 9 12
Thread 1 event order: 1 2 3 6 7 10 11 14
Thread 0 event order: 1 2 4 5 8 9 12 13 16
Thread 1 event order: 1 2 3 6 7 10 11 14 15 18
```

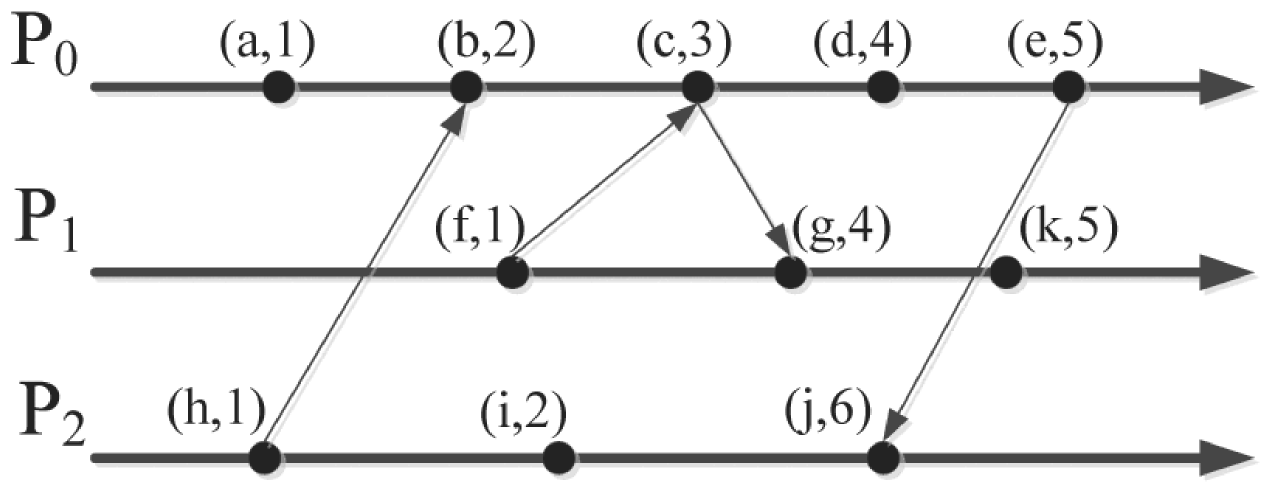
Analysis:

The implemented Lamport logical clock algorithm effectively models the ordering of events in a distributed system comprising two processes (threads). Each process maintains its logical clock, incrementing it with each internal event or message exchange. The key feature of this algorithm is its ability to maintain causality: when a process receives a message, it adjusts its clock to ensure that the event of receiving the message occurs after the event of sending it. This adjustment prevents causality violations and ensures that events are consistently ordered across the system.

In this implementation, each thread alternates between executing internal events and sending/receiving messages. The use of condition variables and mutexes ensures that messages are properly handled in a concurrent environment, preventing race conditions or lost updates. The result is that the logical clocks of the processes remain synchronized according to Lamport's rules, even though the processes themselves may operate asynchronously.

The observed output demonstrates that the event orders are consistent with the causality constraints of the system. The event orders printed by each thread show that all causally related events (like message send and receive pairs) are ordered correctly. The random delays introduced simulate the real-world asynchronous behavior of distributed systems, providing a robust test for the correctness of the algorithm.

This implementation also showcases the challenges of distributed systems, where maintaining a consistent view of event ordering across independent processes requires careful coordination. The Lamport clock provides a simple yet powerful method for achieving this, making it foundational in the study of distributed algorithms.



Conclusion:

The successful implementation of the Lamport logical clock algorithm in a multi-threaded environment demonstrates the effectiveness of the algorithm in maintaining a consistent and causally ordered sequence of events across distributed processes. This project highlights the importance of logical clocks in distributed systems, where physical clocks are not always sufficient or available.

By using threads to simulate concurrent processes, the implementation accurately models the challenges of maintaining causality in a distributed system. The use of synchronization mechanisms such as mutexes and condition variables ensures that message passing is handled correctly, avoiding common pitfalls like race conditions. The results confirm that the Lamport logical clock algorithm can reliably order events even in the presence of asynchronous execution, thereby preserving the causal relationships between events.

This experiment provides a clear understanding of how logical clocks work and why they are essential in distributed computing, particularly in scenarios where maintaining a consistent global state across multiple processes is critical. The insights gained here can be extended to more complex distributed systems, underscoring the relevance of Lamport clocks in both academic research and practical applications in distributed computing.