

Report: Comparing Obstruction-Free and Wait-Free Snapshot Algorithms

Author: Om Dave (CO22BTECH11006)

Introduction

The goal of this assignment is to compare the Wait-Free and Obstruction-Free snapshot algorithms for Multiple Reader Multiple Writer (MRMW) registers. The Wait-Free Snapshot algorithm ensures that all threads complete their tasks in a finite number of steps regardless of the actions of others, while the Obstruction-Free Snapshot algorithm guarantees completion only in the absence of contention. This comparison aims to evaluate the performance differences between these two algorithms, focusing on average-case and worst-case scenarios in concurrent systems.

Program Design and Implementation

System Specifications

The experiments were conducted on a system with the following specifications:

Detail	Specification
CPU(s)	12
Model Name	AMD Ryzen 5 5600H
Threads per Core	2
Cores per Socket	6

Code Structure and Workflow

Main Program Flow

The program design consists of writer and snapshot threads interacting with a shared snapshot object that implements either the Obstruction-Free or Wait-Free snapshot algorithm. Below is a generalized description of the main workflow:

1. Initialization:
- **Input Parameters:** The program reads parameters (*nw*, *ns*, *M*, *μw*, *μs*, *k*) from an input file (*input.txt*), where:

▪ *nw*: Number of writer threads

▪ *ns*: Number of snapshot collector threads

▪ *M*: Number of registers

▪ *μw*: Average delay (in microseconds) between writer operations

▪ *μs*: Average delay (in microseconds) between snapshot operations

- **k**: Number of snapshots each collector will take
- **Distributions**: `exponential_distribution` objects are used to model the delays for writers (`distribution_writer`) and snapshot collectors (`distribution_snap`).
- **Snapshot Object Creation**: Depending on the experiment, either an **Obstruction-Free** or **Wait-Free Snapshot** object (`snapshotObj`) is created to manage the shared registers.

2. Thread Creation:

- **Writer Threads**: `nw` writer threads are created to continuously write random values to the shared registers. Each writer thread runs until a termination signal is received.
- **Snapshot Threads**: `ns` snapshot collector threads are created to periodically take snapshots of the shared registers. Each snapshot collector takes `k` snapshots before finishing.

3. Thread Operations:

- **Writer Threads**:
 - Each writer thread writes a random value to a random location in the shared registers.
 - The write operation is logged, including the timestamp and value written.
 - After writing, the thread sleeps for an exponentially distributed period based on `μw`.
- **Snapshot Threads**:
 - Each snapshot collector thread takes `k` snapshots of the shared registers.
 - The snapshot operation is logged, including the values read and the time taken.
 - After each snapshot, the thread sleeps for an exponentially distributed period based on `μs`.

4. Termination and Log Collection:

- Once all snapshot threads have completed their operations, a termination signal (`term = true`) is sent to all writer threads.
- Logs are collected from both writer and snapshot threads. The logs include details about the operations performed, such as timestamps, values written, and snapshot results.
- Logs are sorted by timestamp and written to an output file (`output.txt`).

5. Performance Metrics:

- **Average and Worst-case Times** for both update and snapshot operations are computed using the collected data.
- These metrics help in comparing the performance of the two algorithms in terms of average-case scalability, worst-case scalability, and the impact of updates on snapshot operations.

Changes Made to Construct MRMW Snapshot Algorithm from Given MRSW Implementation

The original implementations of the obstruction-free and wait-free snapshot algorithms, as described in the book, were designed for multi-reader single-writer (MRSW) registers. In this problem, we need to modify these algorithms to accommodate `nw` writers and `M` multi-reader multi-writer (MRMW) registers, where multiple writer threads can simultaneously write to the same location, and multiple snapshot threads can concurrently read the register values.

Changes in Obstruction-Free MRSW Snapshot Algorithm

- **Handling Multiple Writers:** In the MRSW implementation, each register stores a value along with its timestamp. However, in the MRMW case, where multiple writers may modify the same location concurrently, two different threads could obtain the same timestamp and write the same value to that location, leading to an incorrect double-collect of an invalid state. To solve this issue, we store not only the timestamp and value in each register but also the `thread_id` of the thread that modified the value. This additional identifier helps prevent false collects of invalid states when two threads write with the same timestamp and value.
- **Modification of StampedValue Class:** The `StampedValue` class is modified to hold three entities: `timestamp`, `value`, and `thread_id`. We also use C++ atomics for the MRMW register locations to ensure thread safety.
- **Managing Concurrent Writes:** In the MRSW scenario, a writer thread updates the timestamp of the location after reading it, and since only one writer modifies a location at a time, the timestamps are safe. However, in the MRMW case, when multiple writers write to the same location concurrently, the timestamps are no longer safe. Consider the following scenario:
 1. Initially, a particular location has a timestamp of 0.
 2. Two threads, A and B, try to write to the same location concurrently and read the initial timestamp of 0.
 3. Thread A writes to the location with timestamp 1.
 4. Thread B attempts to write to the location but halts.
 5. A new thread, C, reads the timestamp 1, increments it, and writes to the location with timestamp 2 and a value `v`.
 6. Thread B wakes up and writes to the location with timestamp 1.
 7. Thread C then comes again, reads the timestamp 1, increments it, and writes back to the location with timestamp 2 and the same value `v`.

As a result, we end up with the same `{timestamp, value, thread_id}` at two different times, which could lead to an incorrect double-collect of an invalid state.

- **Thread-Local Sequence Numbers:** To solve this issue, we use thread-local sequence numbers for timestamps. Each thread maintains its own sequence number, which it increments each time it writes to a location. This ensures that if the same thread writes the same value to the same location again, it will increment its own previous sequence number, thus preventing identical timestamps. While two different threads could still write with the same timestamp, the `thread_id` differentiates them. Since timestamps are specific to each thread, they could potentially decrease compared to others, but the algorithm remains correct due to the unique combination `{timestamp, value, thread_id}`. Therefore, instead of referring to these as timestamps, they are more appropriately called **sequence numbers** for each thread.
- **Summary of Update Process:** Each writer thread has its own sequence number starting from 0. When a thread writes to a location, it increments its sequence number and writes it to the timestamp field of that location.

Changes in Wait-Free MRSW Snapshot Algorithm

- **Incorporating Obstruction-Free Changes:** All the changes made in the obstruction-free MRSW algorithm are also applied in the wait-free implementation.

- **Handling `helpsnap` Array:** In the MRSW wait-free case, the snapshot of the entire register is stored during an update operation within the `StampedValue` class itself, as the number of writers and locations are the same. However, in the MRMW scenario, we have `M` locations and `nw` writers. To handle this, we create a `helpsnap` array with `nw` elements, where each element can store a snapshot of the entire register. When a writer updates any location, it takes a snapshot of the entire register and stores it in its respective position in the `helpsnap` array. During a snapshot operation, if a writer has moved twice, the snapshot collector can refer to the most recent snapshot taken by that writer thread from the `helpsnap` array.
- **Ensuring Wait-Free Guarantees:** This mechanism ensures that the snapshot algorithm maintains wait-free properties even when there are multiple writers, allowing for consistent and reliable collection of register values without risking an incorrect or incomplete state.

Complications Encountered During Programming

- **Building Logic for MRMW Case:** Translating the logic from MRSW to MRMW required a deep understanding of potential edge cases where the original algorithm could fail. Handling concurrent writes from multiple threads and ensuring the correctness of snapshot operations was particularly challenging. Identifying these edge cases and devising effective solutions, such as using `thread_id` and **thread-local sequence numbers** to uniquely identify modifications, was essential to the success of the algorithm.
- **Atomic Implementation of `StampedValue` Class:** Implementing atomic operations for the `StampedValue` class required a thorough understanding of **C++ atomics**. This involved extensive reading of the C++ atomic documentation to ensure thread safety and overcome compilation issues. We also had to link with the `-latomic` flag to successfully compile the program, ensuring that the atomic operations functioned correctly.
- **Custom Comparators for Complex Classes:** Writing custom comparators was necessary in several parts of the program. For example, comparing `StampedValue` objects that contain multiple fields (`timestamp`, `value`, and `thread_id`) required specific comparators to ensure accurate sorting and comparisons. Similarly, for log entries, we needed to sort logs accurately based on their timestamps, which involved creating custom comparison functions to handle the sorting logic properly.

MRMW Snapshot Class

Stamped Value Class

```
template <typename T>
class StampedValue {
public:
    int stamp;
    T value;
    int tid;
    // Default constructor
    StampedValue() : stamp(0), value(T()) , tid(-1) {}

    // Parameterized constructor
    StampedValue(int ts, T v, int thread_id) : stamp(ts), value(v),
```

```

tid(thread_id) {}

//define custom comparator function
bool operator==(const StampedValue& other) const {
    return (stamp == other.stamp && value == other.value && tid ==
other.tid);
}
};

```

Obstruction Free Snapshot Class

```

template <typename T>
class ObstructionFreeSnapshot {
private:
    vector<atomic<StampedValue<T>>> a_table; // Array of atomic MRMW
registers
    vector<int> seq_nos; // Array of sequence numbers for each writer
thread

    vector<StampedValue<T>> collect() {
        vector<StampedValue<T>> copy(a_table.size());
        for (int i = 0; i < a_table.size(); i++) {
            copy[i] = a_table[i].load();
        }
        return copy;
    }

public:
    ObstructionFreeSnapshot(int capacity, int writer_threads) {
        a_table = vector<atomic<StampedValue<T>>>(capacity);
        for (int i = 0; i < capacity; i++) {
            a_table[i].store(StampedValue<T>(0, T(), -1));
        }
        seq_nos = vector<int>(writer_threads);
        for (int i = 0; i < writer_threads; i++) {
            seq_nos[i] = 0; //storing the local sequence number of each
threada
        }
    }

    void update(int thread_id, int location, T value) {
        int newstamp = seq_nos[thread_id] + 1;
        seq_nos[thread_id] = newstamp;
        a_table[location].store(StampedValue<T>(newstamp, value,
thread_id));
    }

    vector<T> scan() {
        vector<StampedValue<T>> oldCopy, newCopy;

        oldCopy = collect();
    }
}

```

```

        while (true) {
            newCopy = collect();
            bool flag = true;
            for (int i = 0; i < a_table.size(); i++) {
                if (oldCopy[i] == newCopy[i]) continue;
                flag = false;
                break;
            }
            if (flag) break; // Both copies are equal, double collect is
successful, break from the loop
            oldCopy = newCopy;
        }
        vector<T> result(a_table.size());
        for (int i = 0; i < a_table.size(); i++) {
            result[i] = newCopy[i].value;
        }
        return result;
    }
};

```

Wait Free Snapshot Class

```

template <typename T>
class WaitFreeSnapshot {
private:
    vector<atomic<StampedValue<T>>> a_table; // Array of atomic MRMW
registers
    vector<vector<T>> helpsnap; // Stores recently taken snapshots by each
writer thread
    vector<int> seq_nos; // Array of sequence numbers for each writer
thread
    int nw; // Number of writer threads

    vector<StampedValue<T>> collect() {
        vector<StampedValue<T>> copy(a_table.size());
        for (int i = 0; i < a_table.size(); i++) {
            copy[i] = a_table[i].load();
        }
        return copy;
    }

public:
    WaitFreeSnapshot(int writer_threads, int capacity) {
        a_table = vector<atomic<StampedValue<T>>>(capacity);
        for (int i = 0; i < capacity; i++) {
            a_table[i].store(StampedValue<T>(0, T(), -1));
        }
        helpsnap = vector<vector<T>>(writer_threads, vector<T>(capacity));
        nw = writer_threads;
        seq_nos = vector<int>(writer_threads, 0);
    }

```

```

    }

    void update(int thread_id, int location, T value) {
        int newstamp = seq_nos[thread_id] + 1;
        seq_nos[thread_id] = newstamp;
        a_table[location].store(StampedValue<T>(newstamp, value,
thread_id));
        // Take snapshot for current writer thread
        helpsnap[thread_id] = scan();
    }

    vector<T> scan() {
        vector<StampedValue<T>> oldCopy, newCopy;
        vector<bool> can_help(nw, false);

        oldCopy = collect();

        while (true) {
            newCopy = collect();
            bool flag = true;
            for (int i = 0; i < a_table.size(); i++) {
                if (oldCopy[i] == newCopy[i]) continue;
                flag = false;
                // Check the latest writer thread that modified this
location
                int writer_thread = newCopy[i].tid;
                if(can_help[writer_thread]) {
                    // This writer thread has modified this location
during this snapshot
                    return helpsnap[writer_thread];
                }
                else can_help[writer_thread] = true;
            }
            if (flag) break; // Double collect is successful
            oldCopy = newCopy;
        }
        vector<T> result(a_table.size());
        for (int i = 0; i < a_table.size(); i++) {
            result[i] = newCopy[i].value;
        }
        return result;
    }
};

```

Snapshot Output

Sample output

```

Writer 0: Value 33 written to location 6 at 0.000118
Writer 1: Value 55 written to location 5 at 0.000176
Writer 2: Value 53 written to location 5 at 0.000233

```

```

Writer 0: Value 9 written to location 2 at 0.000236
Writer 1: Value 45 written to location 1 at 0.000273
Writer 3: Value 45 written to location 7 at 0.000321
Writer 0: Value 66 written to location 9 at 0.000354
Writer 2: Value 52 written to location 6 at 0.000359
Writer 1: Value 2 written to location 6 at 0.000398
Writer 2: Value 90 written to location 6 at 0.000438
Writer 3: Value 26 written to location 8 at 0.000448

Snapshot 0: [ 0 / 45 / 9 / 0 / 0 / 53 / 90 / 45 / 26 / 66 / ]
started at: 0.000497, ended at: 0.000501

Writer 1: Value 21 written to location 9 at 0.000554

Snapshot 1: [ 0 / 45 / 9 / 0 / 0 / 53 / 90 / 45 / 26 / 66 / ]
started at: 0.000552, ended at: 0.000555

Writer 0: Value 39 written to location 0 at 0.000572
Writer 2: Value 31 written to location 3 at 0.000575
Writer 3: Value 21 written to location 5 at 0.000584

Snapshot 2: [ 39 / 45 / 9 / 31 / 0 / 21 / 90 / 45 / 26 / 21 / ]
started at: 0.000598, ended at: 0.000603

```

- **Snapshot 0** and **Snapshot 1** are snapshots taken by threads 0 and 1 respectively. The start and end times of these snapshots can be considered by as bounds of the method call between which the snapshot can be linearized.
- We can see that **snapshot 0** and **snapshot 1** are the same, even though the writer 1 writes the value **21** to location **9** at time **0.000554**. Since this occurs between the start and end times of **snapshot 1**, the linearization point of **snapshot 1** can be said between **0.000552** and **0.000554**, which indicates that the current writer modification is not reflected in this snapshot.
- Similarly, **snapshot 2** can also be linearized as it reflects all the changes made by other writer threads.

Performance Analysis

Experiment 1: Average-case Scalability

Methodology

 Average-case Scalability

Observations and Reasoning

Experiment 2: Worst-case Scalability

Methodology

 Worst-case Scalability

Observations and Reasoning

Experiment 3: Impact of Update Operation on Scan (Average-case)

Methodology


This experiment analyzed the impact of varying the ratio of writer threads to snapshot threads (nw/ns) on the average-case performance.

 Impact of Update Operation (Average-case)

Observations and Reasoning

Experiment 4: Impact of Update Operation on Scan (Worst-case)

Methodology

 Impact of Update Operation (Worst-case)

Observations and Reasoning

Conclusion

Note

- The figures (Figure_1.png, Figure_2.png, etc.) are placeholders. Replace them with the actual paths to your images.
- Expand each section further as needed to include specific observations and detailed descriptions.