CS60203: Design Optimization of Computing Systems *September 10, 2024*

Lock Free Programming Part 2

Instructor: Mainack Mondal *Scribe-By: <Bhavani Kalal >*

1 Introduction

Lock-free programming is a concurrency technique that allows multiple threads to access and modify shared data without traditional locks like mutexes. Instead of waiting for locks, threads use atomic operations to ensure correct updates, which eliminates risks like deadlocks and livelocks. This approach ensures that at least one thread always makes progress, even if others are delayed. Lock-free programming is ideal for high-performance systems, as it reduces contention and improves scalability.

2 Recap of Lock-Free Programming (Part 1)

What we have discussed:

- Locks are mechanisms used in concurrent programming to ensure that multiple threads or processes do not simultaneously access shared resources, preventing race conditions. They are necessary for maintaining data consistency and avoiding conflicts in multi-threaded environments.
- Types of locks include mutexes, spinlocks, and read-write locks. Drawbacks of locks can include deadlocks, performance bottlenecks due to contention, and increased complexity in managing them.
- Lock-free programming is a technique that allows multiple threads to access shared data without using locks, improving performance and avoiding deadlock risks. It is beneficial for reducing latency and increasing throughput in concurrent applications.
- The ABA problem occurs when a value in memory changes from A to B and then back to A, causing confusion in certain algorithms (e.g., CAS-based algorithms), as it appears unchanged even though intermediate changes occurred.

3 Outline

The main topics covered in this lecture are:

- Lock-Free Primitives (Hardware and Software)
- Read-Copy-Update (RCU) in Linux
- Lock-Free APIs in Java and C/C++
- Problems with Lock-Free Programming
- Uses of Lock-Free Programming

4 Lock-Free Primitives

Lock-free primitives, such as **compare-and-swap (CAS)** and **fetch-and-add**, are atomic operations that enable threads to safely manipulate shared memory without using locks. These primitives enable the creation of **lock-free data structures** like queues and stacks, where multiple threads can operate concurrently without blocking. Lock-free data structures help boost performance in multi-threaded environments. As with lock-free programming, challenges such as memory management and addressing the ABA problem must be carefully managed

4.1 Lock-Free Primitives in Hardware

The **Compare-And-Swap (CAS)** instruction is the key hardware primitive. On x86 architectures, this is implemented using the `CMPXCHG` instruction, which operates as follows:

```
CMPXCHG reg/mem32, reg32
```

Here, the instruction compares the accumulator (%eax) with the value in memory (destination). If they are equal, the destination is updated with the source register, otherwise the accumulator is updated with the destination's value.
The instruction is atomic on a single core. For multiple cores, a special prefix (`LOCK`) is used:

```
LOCK CMPXCHG reg/mem32, reg32
```

4.2 Lock-Free Primitives in Software

In software, compilers provide primitives that rely on hardware instructions like CAS. For instance, GCC provides atomic built-in functions such as:

- **__atomic_compare_exchange**: Atomically compares the value of an object to an expected value and, if equal, replaces it with a new value. This is used in constructing lock-free algorithms like compare-and-swap (CAS).
- `__atomic_fetch_add`: Atomically adds a specified value to the current value of the atomic variable, ensuring that no race conditions occur when multiple threads increment the same counter.
- `__atomic_test_and_set`: Sets a flag to true and returns its previous value. This is often used in spinlock implementations and ensures atomic checking and setting of flags.
- `__atomic_is_lock_free`: Returns whether the type of atomic object is lock-free on the system, allowing developers to choose more efficient data structures where possible.

These functions allow programmers to build lock-free data structures and systems.

5 Read-Copy-Update (RCU) in Linux

Read-Copy-Update (RCU) is a synchronization mechanism in the Linux kernel that enables readers to access shared data without locks while writers update the data concurrently. This lock-free approach is especially useful in scenarios where there are many readers and few writers, as it allows for highly efficient and scalable read operations.

`How RCU works:`

- **Readers**: RCU allows multiple readers to access data concurrently without acquiring locks. This ensures low overhead for read-heavy workloads, as readers never blocks.
- **Writers**: When writers need to update data, they do not directly modify it. Instead, they create a new copy of the data, make the necessary updates, and then replace the old version with the new one. The old version remains available to readers until all in-progress readers have completed their access.
- **Grace Period**: RCU ensures that the old version of data remains available until a "grace period" has passed, during which all readers who may have been accessing the data complete their operations. Once the grace period ends, the old data can be safely reclaimed and deallocated.

5.1 Example of RCU List Management

This example demonstrates how Read-Copy-Update (RCU) is used to manage a list, allowing concurrent reads while writers update the list safely.

5.1.1 Initial List Setup

Let's assume we start with an empty list:

`RCU List: [ ]`

- **Readers** can access the list at this point, but since it's empty, they see no data. - **Writers** can add elements, which will be reflected in the RCU list after the update.

5.1.2 Writer Adds an Element

When a writer adds a new element 'A', they create a new node and insert it into the list:
Operation: `list_add_rcu(A)`
State:

- List seen by readers (before writer insertion): `[ ]`
- List after writer insertion: `[ A ]`

Readers' View: The readers can now see the new node 'A' immediately after the writer uses `list_add_rcu()`. RCU ensures that this addition does not block any ongoing readers.

5.1.3 Concurrent Readers Access the List

While readers are iterating over the list, they access the version containing 'A'.
State:

- Reader 1 sees: `[ A ]`
- Reader 2 sees: `[ A ]`

These readers do not need locks and access the list concurrently without blocking.

5.1.4 Writer Removes an Element

If a writer decides to remove the element 'A', it is marked for removal, but the old version of the list is still available for readers who are in the middle of their operations.

Operation: `list_del_rcu(A)`
State:

- List seen by active readers: `[ A ]`
- New list for future readers: `[ ]`

Readers' View: Existing readers (e.g., Reader 1 and Reader 2) continue to see the old version with 'A', while new readers will see an updated list without 'A'.

5.1.5 RCU Synchronization (Grace Period)

After removing 'A', the writer calls `rcu_synchronize()` to ensure that all readers accessing the old version have finished.

Operation: `rcu_synchronize()`
State:

- During the grace period: Readers continue to see the old list, `[ A ]`, if they began before the removal.
- After the grace period: All readers have completed, and the list becomes `[ ]`.

5.1.6 Final Cleanup

After the grace period, once all readers are done accessing the old version, the memory for 'A' can be safely reclaimed.

Operation: `kfree_rcu(A)`
State:

- Final state of the list: `[ ]`
- Node 'A' is now safely removed, and its memory is reclaimed.

5.1.7 Functions:

- `list_add_rcu()` adds a node to the list, which is immediately visible to readers.
- `list_del_rcu()` marks a node for removal, but ongoing readers continue to see the old version.
- `rcu_synchronize()` ensures that all readers have finished accessing the old version before it is removed.
- After the grace period, `kfree_rcu()` safely reclaims memory for the old data.

5.1.8 Example Timeline

- **T0**: List = `[ ]` (empty)
- **T1**: Writer adds ‘A‘ → List = `[ A ]`
- **T2**: Reader 1 and Reader 2 start reading the list → List (to readers) = `[ A ]`
- **T3**: Writer removes ‘A‘, but readers still see old version → List (to new readers) = `[ ]`, old readers still see `[ A ]`
- **T4**: `rcu_synchronize()` waits for Reader 1 and Reader 2 to finish
- **T5**: After the grace period ends, ‘A‘ is fully removed, and its memory is reclaimed → List = `[ ]`

5.2 Benifits of RCU

- **Scalability**: RCU is highly efficient for read-heavy workloads because readers are never blocked, making it suitable for systems with many concurrent readers.
- **Lock-Free Reads**: Readers do not need locks, avoiding the overhead and contention that can occur in lock-based synchronization mechanisms.
- **Efficient Memory Reclamation**: RCU ensures safe memory reclamation by deferring the freeing of old data until all readers are done, avoiding the risks of use-after-free bugs.

5.3 Limitations of RCU

- **Concurrent Writers**: Atmost only one writer can update the data structure at a time, we have to use separate mechanism for blocking writers.
- **Memory Usage**: RCU can increase memory consumption due to maintaining multiple versions of data, leading to higher memory overhead.
- **Complexity**: Correctly implementing RCU is complex and requires careful management of grace periods and memory reclamation, which can introduce bugs and performance issues.
- **Delayed Cleanup**: Memory reclamation is delayed until a grace period has passed, which can result in prolonged memory usage and potential delays in releasing unused memory.
- **Not Ideal for Frequent Updates**: RCU is optimized for scenarios with many reads and infrequent writes. For workloads with frequent updates, the overhead of maintaining multiple versions can outweigh the benefits.

5.4 RCU in User Space

The RCU library provides functions to implement RCU synchronization, allowing efficient concurrent access to shared data. You use functions like `rcu_read_lock()` and `rcu_read_unlock()` to manage read-side critical sections, `rcu_assign_pointer()` to safely update pointers, and `rcu_synchronize_rcu()` to wait for all read-side operations to complete before making changes. This setup ensures that multiple threads can read shared data concurrently without locking, while updates are handled in a way that maintains consistency and avoids data races.

6 Lock-Free APIs

Lock-free programming provides efficient concurrency control without traditional locking mechanisms. Both C++ and Java offer built-in support for lock-free operations through their respective standard libraries.

6.1 Lock-Free API in C++

In C++, lock-free programming is supported by the ‘atomic‘ header, which provides several atomic types and operations:

- `std::atomic<T>`: A template class that provides atomic operations on types. Key operations include:
  - `load()`: Atomically reads the value.
  - `store()`: Atomically writes a value.
  - `exchange()`: Atomically sets the value and returns the old value.
  - `compare_exchange_weak()` and `compare_exchange_strong()`: Atomically compares and exchanges values if they match.
- `std::atomic_flag`: Provides atomic operations for simple flags. Supports:
  - `test_and_set()`: Atomically sets the flag and returns its previous state.
  - `clear()`: Clears the flag.
- `std::shared_mutex`: While not strictly lock-free, it provides shared and exclusive locking mechanisms which can be used to implement lock-free structures under certain conditions.

6.2 Lock-Free API in Java

Java provides lock-free operations primarily through the ‘java.util.concurrent‘ package:

- `AtomicInteger`: Provides atomic operations on an integer value. Key methods include:
  - `get()`: Atomically retrieves the value.
  - `set()`: Atomically sets a new value.
  - `compareAndSet()` and `weakCompareAndSet()`: Atomically compares and updates the value if it matches.
  - `incrementAndGet()` and `decrementAndGet()`: Atomically increments or decrements the value.
- `AtomicReference<T>`: Provides atomic operations on a reference type. Includes methods similar to `AtomicInteger`, but for object references.
- `ConcurrentLinkedQueue`: A lock-free queue implementation that supports concurrent insertion and removal operations.
- `ConcurrentHashMap`: A thread-safe, lock-free hash map that allows concurrent read and write operations.

6.3 Examples of Lock-Free Code

6.3.1 C++ Example: Lock-Free Stack

Below is an example of a simple lock-free stack implementation in C++ using `std::atomic`:

```
#include <atomic>
#include <memory>

template <typename T>
class LockFreeStack {
public:
    LockFreeStack() : head(nullptr) {}

    void push(const T& value) {
        Node* new_node = new Node(value);
        new_node->next = head.load();
        while (!head.compare_exchange_weak(new_node->next, new_node)) {
            // Retry until successful
        }
    }

    bool pop(T& value) {
        Node* old_head = head.load();
        while (old_head && !head.compare_exchange_weak(old_head, old_head->next)) {
            // Retry until successful
        }
        if (old_head) {
            value = old_head->data;
            delete old_head;
            return true;
        }
        return false;
    }

private:
    struct Node {
        T data;
        Node* next;
        Node(const T& data) : data(data), next(nullptr) {}
    };

    std::atomic<Node*> head;
};
```

6.3.2 Java Example: Lock-Free Counter

Below is an example of a lock-free counter implementation in Java using `AtomicInteger`:

```java
import java.util.concurrent.atomic.AtomicInteger;

public class LockFreeCounter {
    private AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getValue() {
        return counter.get();
    }

    public static void main(String[] args) {
        LockFreeCounter counter = new LockFreeCounter();

        // Example usage
        counter.increment();
        System.out.println("Counter Value: " + counter.getValue());
    }
}
```

7 Problems of Lock-Free Programming

While lock-free programming offers several advantages, such as reduced contention and improved scalability, it also presents several significant challenges and potential issues:

- **Difficulty in Writing Correct Code**: Writing correct lock-free code is inherently complex due to the need to manage concurrency without traditional locking mechanisms. The complexity arises from ensuring atomic operations and consistency without introducing race conditions or subtle bugs.

- **High Memory Usage**: Some lock-free algorithms, such as those using spin-locks, can lead to high memory consumption. This occurs because these algorithms may involve creating and maintaining multiple versions of data structures to ensure that readers and writers do not interfere with each other.

- **Performance Issues**: In certain scenarios, lock-free programming can result in worse performance compared to traditional mutexes. This degradation can happen due to the overhead of atomic operations or if the workload is not read-heavy, making traditional locks more efficient in those cases.

- **ABA Problem**: The ABA problem is a specific issue in lock-free programming where a location is modified twice between reads without the algorithm detecting the intermediate modifications. This problem can lead to incorrect assumptions and inconsistent state if not properly handled.

Overall, while lock-free programming can offer performance benefits in specific situations, it requires careful consideration and handling of these challenges to ensure correct and efficient operation.

8 Lock-Based vs Lock-Free Programming

Choosing between lock-based and lock-free programming depends on various factors. Here, we highlight key considerations to help in making an informed decision:

- **Number of Threads**: For applications with a large number of threads, lock-free programming can offer better performance by reducing contention and avoiding the overhead associated with traditional locks. Lock-free algorithms, such as those using atomic operations, scale more effectively with the number of threads, as they avoid bottlenecks caused by lock contention.
- **Contention Period**: The choice between lock-based and lock-free methods also depends on the contention period:
  - **Low Contention**: When contention is low, lock-free methods might be preferable as they can offer improved performance and scalability without the overhead of acquiring and releasing locks. Lock-free algorithms minimize contention and can provide better throughput in scenarios where conflicts are rare.
  - **High Contention**: If contention is high, lock-based solutions might be more appropriate. Traditional locks, such as mutexes, can handle high contention scenarios more effectively by serializing access to shared resources. Lock-free methods might suffer from performance degradation due to frequent retries and contention overhead in such cases.

Rule of Thumb: Always test and measure the performance of your code to determine which approach is best.

9 References

- Paul Mckenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?*
- Anthony Williams, *C++ Concurrency in Action*
- https://arxiv.org/pdf/1701.00854
- https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html
- https://en.cppreference.com/w/cpp/atomic/atomic