CS60203: Design Optimization of Computing Systems *September 8, 2024*

Lock Free Programming Part-1

Instructor: Mainack Mondal *Scribe-By: Arup Sau*

1 Summary of The Topic

Locks are synchronization methods employed in operating systems to regulate access to shared resources, guaranteeing that only a single thread can access a vital portion at any given moment. In environments that support multiple threads, it is possible for many threads to simultaneously attempt to modify shared data, which can result in inconsistencies in the data and race conditions.

2 Synchronization Problem

The Synchronization Problem involves guaranteeing that only one thread can execute its important portion at any given time in order to prevent data corruption.

2.1 Race Condition:

A race problem arises in concurrent programming when many threads attempt to concurrently access and modify shared data, and the final result of the program execution is influenced by the specific timing or order in which these threads are executed. This condition can be prevented by locks.

Example: Consider a shared variable balance being updated by two thread: If both threads execute these operations concurrently, the following sequence might occur:

- Thread A reads the balance as Rs 100.
- Thread B reads the balance as Rs 100.
- Thread A updates the balance to Rs 150.
- Thread B updates the balance to Rs. 50.

As a result of a race condition, the ultimate balance is Rs 50 instead of Rs 150, since Thread B replaces the modification made by Thread A. So to prevent this we use locks.

3 Types of Locks:

There are different types of lock like Mutex, Spin Lock, Semaphore etc. These locks are used depending on the specific requirements of application. Some of the locks are discussed in Table 1.

Table 1: Different types of lock and their pros and cons

Feature	**Mutex**	**Spin Lock**	**Semaphore**
Definition	A basic lock where only one thread can hold the lock and access the resource, while others are blocked.	Threads continuously check if the lock is available, spinning in a loop. They are lightweight and used when the wait time is expected to be short.	It allows multiple threads or processes to synchronize their access to a critical section.
Ownership	Can be owned by one thread at a time	No ownership; any thread can attempt to acquire it	No ownership; any thread can signal or wait
Pros	• Simplicity • Prevents Race Condition	• Efficient for short waiting time • Avoid Context Switch	• Controlled Access • Blocking and Waking • Flexibility
Cons	• Deadlock • Priority Inversion • Performance Overhead(Context Switch)	• Busy waiting • Not suitable for high contention • Starvation	• Complexity • Deadlocks • Starvation

4 Lock Free Programming:

Designing lock-free data structures is challenging. Rather we design lock free data structure. Lock-free data structures are designed to provide thread-safe operations without traditional locks. Examples Stack, Queue, Buffer. So to use lock free data structure we use lock free primitives.

4.1 Compare and Swap(CAS):

It is a cricial atomic operation that compares the value at a memory location to an expected value. If the current value matches the expected value, it swaps the value with a new one automatically.

4.2 Fetch and Add :

Atomically increments or decrements a value at a memory location ensuring that no other thread ca update it concurrently

```
compare-and-swap(T* location, T cmp, T new){
    // do atomically (in hardware)
    {
        T val = *location;
        if (cmp == val)
            *location = new;
        return val;
    }
}
```

Figure 1: Compare and Swap

```
fetch-and-add(T* location, T x)
{
  // do atomically (in hardware)
  {
   T val = *location;
   *location = val + x;
   return val;
  }
}
```

Figure 2: Fetch and Add

4.3 Load-Linked (LL) and Store-Conditional (SC):

Reads the value from a specified memory location. Saves the memory location's address so that ensuing SC actions can confirm if the value there has changed. If the value is the same as the one read by the previous LL operation, writes the new value to the memory address. If the value has been changed the SC operation fails, and the write does not occur.

```
store-conditional(T* ptr, T value){
   if (no update to *ptr since LL to this addr) {
       *ptr = value;
        return 1; // success!
   } else {
        return 0; // failed to update
   }
}
```

Figure 3: Load-Linked and Store Conditional

5 ABA Problem:

When using atomic operations like as Compare-And-Swap (CAS) or Load-Linked/Store-Conditional (LL/SC), the ABA Problem frequently arises in concurrent programming and lock-free data structures. It occurs when a value at a certain memory address shifts from A to B and back again to A, raising questions about whether atomic operations are performed correctly.

5.1 Understanding the Problem:

Step 1: Thread 1 reads shared variable (Initial State="A")

Step 2: Thread 2 execute and changes the variable(State="B"). While Thread 1 is busy with its calculation, Thread 2 wakes up and modifies the shared variable from A to B.

Step 3: Thread 2 changes the variable back to A(State="A" again)

Step 4: Thread 1 wake up and tries to compare and set.

Thread 1, which has not been aware of the changes made by Thread 2, proceeds with CAS operation, thinking that the variable's value is still the same as it was when it first observed it.

5.2 Solution:

Keep an Update Count: Maintain a counter or version number alongside the value at the memory location. The value by itself does not indicate if it has been altered, as this counter is incremented with each update.

Don't Recycle Memory Too Soon: After memory locations are released, do not deallocate or reuse them right away. Make sure that no memory is recycled or recovered until no processes that use it remain unfinished. This helps prevent other threads from unintentionally reuse memory and result in strange behaviours.

6 Advantages of Lock-Free Programming

- No/Less Context Switches.
- Higher CPU Frequency and Throughput
- No Deadlocks or Priority Inversions
- Faster Multicore Programming

Lock-free programming offers significant performance benefits in scenarios where high concurrency is essential. By eliminating locks, it improves scalability on multicore systems by lowering the possibility of problems like deadlocks and priority inversions and enabling thread progress independently. However, Adopting lock-free algorithms is difficult, because of the designing and hardware dependencies.