Reading Lock-Free Locks Revisited

Krr

Contents

1 Introduction

It starts with *lock-free* algorithms. From the Wikipedia page, *non-blocking* algorithms have many benefits, such as better performance on parallel algorithms, because coherent access doesn't need to be maintained by locks. Everyone has some unpleasant experience with locks, but *lock-free* algorithms can be troublesome too even in simple data-structures. Even a *lock-free* queue can be quite hard to implement. Further more, *lock-free* algorithms/data-structures can be less efficient, so there is a suggestion on Boost.Lockfree's webpage.

> In general we advise to consider if lock-free data structures are necessary or if concurrent data structures are sufficient. In any case we advice to perform benchmarks with different data structures for a specific workload.

Here is the case, this paper, Lock-Free Locks Revisited[1] introduces a way to write code in fine-grained locks while getting efficient *lock-free* behavior.

By combining these two idea, we achieve *performance* and *simplicity*. This paper comes with a implementation at CMUPARLAY/FLOCK which we can play with.

2 Term interpretation

It is my first time to this field, so it worth interpreting some terms here.

First, let me explain *Fine-grained synchronization* and *lock-free programming* here, using notes from CMU's Parallel Computer Architecture and Programming course.

Let's consider a simple list of links. Every CS student should be able to write a correct serial linked list. However, it is not that easy to make it thread-safe. The easiest way to think of it is to add a full lock to the list structure. Every operation acquires this lock, then release the lock when this operation terminates. This works, but all operations are reduced to serial. Intuitively, when we insert two nodes into completely different positions, these operations can be done in parallel.

Fine-grained lock can do a better job. By analyzing what your function might modify, with *extensive* correctness proof, we can transform a linked list using fine-grained lock, i.e., inserting after N0 requires modifying N0.next, so we only lock N0 and the inserted node.

The definition of *non-blocking* algorithm is weird in the first glance. *An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread.* But we are saying about performance, not about crashing. Well, a blocking algorithm acquiring lock can be viewed as preventing other thread from completing operations. Then when it crash, there is no easy way to restore, because the lock is acquired by the crashed thread. Then the definition of *non-blocking* is clear.

Atomic operations like atomicCAS or atomicAdd are basic building blocks for *non-blocking* algorithms. Using more tricks, e.g., wide-CAS, hazard pointers, we can achieve *lock-free* programming.

Idempotence is a property of operator. We have $(S, \cdot)$, where S is a set and $\cdot$ is a binary operator. $\cdot$ is said to be *idempotence* if

$$\forall x \in S, x \cdot x = x$$

its said that the central idea is to implement *idempotence* in the paper. As an example, some code pieces are said to be *idempotent* if they are really executed for multiple times and it appears to be that they are only executed once. A single CAS call is *idempotent*.

Transactional Memory is yet another approach to write *lock-free* programs. TM maintains a series of *transactions*, consisting of read or write operations. Those operations are all atomic. When one processor see the result of the operation, it is guaranteed that all processors can see the same result.

Before the end of this section, I demonstrated a (buggy) implementation of *lock-free* stack.

Listing 1: Lock-free stack.

```
struct Stack { Node* top; };

void push(Stack* s, Node* n) {
  while (true) {
    Node* old_top = s->top;
    n->next = old_top;
    if (compare_and_swap(&s->top, old_top, n) == old_top)
      return;
  }
}

Node* pop(Stack* s) {
  while (true) {
    Node* old_top = s->top;
    if (old_top == nullptr)
      return nullptr;
    Node* new_top = old_top->next;
    if (compare_and_swap(&s->top, old_top, new_top) == old_top)
      return old_top;
  }
}
```

The implementation listing 1 comes from CMU's slide. Although ABA problem still exists (it could be mitigated using TaggedPtr), I'll neglect it here for a clean demonstration. Notice that both of the functions are wrapped with `while (true)`, this guarantees correctness when the CAS failed for times. This is common in *lock-free* programming.

A *thunk* is a function without parameter. Any functions can be wrapped to *thunk*. In C++, for example, you can wrap a function by a lambda that captures its parameters.

3 Back to Paper

3.1 Building Blocks

Let's repeat the definition of *idempotence* by a formal definition presented in the paper. Intuitively, we want the thunks (the lambdas) to be synchronized. For n thunks executing by m threads simultaneously, for each step in the original thunk T, there will only be one thread that *actually* execute this step. Other threads' execution will not be effectual. This idea keeps all threads synchronized.

Definition: A thunk T is *idempotent* if any valid execution E (a series of steps) consisting of runs (steps in a thunk) of T interleaved with arbitrary other steps manipulating shared memory, there exists a subsequence E' of $E \mid T$ (removing steps that do not belongs to T in E), such that

1. if there is a finished run of T, i.e., the last step of T appears, then the last step must be the end of E',

2. removing all of T's steps from E other than E' result in a valid history consistent with a single run of T.

Note that a step consist of an instruction's parameter and result.

E' is the valid steps that is coherent to T, which are the steps that we would like to pick out of the execution process. This definition guarantees that this subsequence exists.

Further more, we want to provide a library, that using the primitives in library to build the thunk will result in a *idempotent* thunk. This target indicates that the library should be powerful and easy-to-use, since we just need to perform transformation. In high-level, it provides these interfaces in listing 2, where function names clearly indicate their usages.

Listing 2: High-level Interfaces

```
1  struct mutable<V> {
2    shared<V> v;
3    V load();                  // (1)
4    void store(V);             // (2)
5    void CAM(V oldV, V newV); // (3)
6  };
7
8  V* allocate<V>(args);        // (4)
9  void retire<V>(V*);          // (5)
```

Although I'll neglect the proof here, the theorem is strong,

Theorem: Replacing each mutable shared variable accessed by a thunk T with a mutable and allocating and retiring all objects in T with the provided allocate and retire operations yields an idempotent version of T.

This paper achieves this by using a shared Log sequence. When any of these operations is executed, it emits a log indexed by a position. To be specific, as demonstrated in listing 3, where `Log` is shared by all threads executing the thunk T, and each thread owns a program counter, indicated and stored in `position`. In any step, `log[...].read()` will give any thread the same result, and `log[...]` will only record the correct value along the thunk's execution because once the value is written, both the original thread and other threads will get the same result, the transition order of the state machine is maintained for all threads, in a *lock-free* manner.

Listing 3: High-level Interfaces

```
1  log := shared<entry>[...];
2  <V, bool> commitValue(V val) {
3    bool isFirst := log[log->position].CAS(empty, val);
4    V result := log[log->position].read();
5    position++;
6    return <result, isFirst>;
7  }
```

3.2 Lockless Locks

Now that we have powerful building blocks, the *idempotent* thunks. Under this technique, building the *lockless lock* is rather easy, following this procedure,

1. The tryLock process itself should be *idempotent*.
2. Whether or not it is locked, we *help other* by using *idempotence* to help it, i.e., execute it. Because of *itempotence*, this extra run will not affect the result.
3. When a tryLock succeed, the lock will be released. It behaves as if the whole thunk T is not executed.

> Note that although tryLock is *idempotent*, it do not strictly follows the theorem above, **tryLock uses local allocation**. But *idempotent* is pertained, since the two branches are exactly the same in the state machine, except local variable will be modified. tryLock is an exception to implement *idempotence*.

There are some difficult problems left here, for example, why can tryLock be cascaded. And we also neglect the proof of correctness in tryLock. But there is actually a simpler view that can give us a better intuition.

3.3 State Machines

The whole paper can be revised in a finite state machine (FSM) view. Where states are encapsulated by all `mutable_` and local variables, i.e., the shared variables among all threads and thread locals. While there can have local variables, local variables should not lead to discrepancy in state machine transitions. What we are going to do, ultimately, is to maintain states so that all FSMs in threads are correctly synchronized.

Every function call to the previously stated five functions are a transition, local variables **must** have the same initial values, so that they can be maintained by the FSM. Every function call contains a tuple, function name, function parameter, return value. This paper finds a good way to give all FSMs the same transition, so that they are synchronized (FSMs are determined by their transition sequences).

However, synchronization is a stronger condition than *idempotent*. tryLock function is a good example. What we really care in synchronization is the shared variables. Ensuring the operation sequence to all **shared variables** are exactly the same, and can implies *idempotence*.

And, its approach is to maintain a shared log sequence, recording each operation so that state sequences are the same. It is a simple but effective idea.

4 Implementation and Results

The paper present an C++ library at cmuparlay/flock. Using this library, we can have a convenient run-time option, *use lock-free algorithm or not*. This enable us to mitigate the difficulty in designing and writing *lock-free* algorithms, instead we can write it in fine-grained lock form, with the following basic interfaces.

1. lock, lockless lock
2. try_lock(lock, thunk), succeed if the thunk appears to successfully executed, just like traditional try_lock
3. mutable_<T>, shared variables maintaining states

Although using this interface will result in a slight performance reduction, from the author's experiments, the overhead is generally less than 10% with fine-grained lock, which is much more acceptable than previous work, saving all the context in each step.

As for extreme cases, when the number of threads is much more than the number of cores, *lock-free* algorithms tend to outperform *blocking* algorithm. But, this transform can be applied without any extra cost, just writing this whole codebase in Flock, you can get a theoretically *lock-free* data structure.

The library also provide classes to mitigate ABA-problem, together with an efficient epoch-based memory manager, enabling real industrial use.

5 Epilogue

Although previously I'm not familiar with *lock-free* programming and its relevant research problems such as ABA-problem, multiword-CAS and lockless lock, I believe this report shows my understanding of this area only after hours of reading. I committed one section to explain the terms as an introduction, then I step into the paper, introducing its building block, *idempotent thunk*. Built upon *idempotent thunk*, lockless lock is readily achieved. Later, I presented my understanding on FSM and the essence of building lockless programs using techniques in this paper. Finally, I gave a brief description of its performance.

References

[1] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. "Lock-Free Locks Revisited". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 278–293. ISBN: 9781450392044. DOI: 10.1145/3503221.3508433. URL: https://doi.org/10.1145/3503221.3508433.