УДК 004.043

Krivuliak V. V., instructor

# OVERVIEW OF LOCK-FREE DATA STRUCTURES

*Zaporizhzhya State Engineering Academy, department SoAS*

In classic lock-based programming, whenever you need to share some data, you need to serialize access to it. The operations that change data must appear as atomic, such that no other thread intervenes to spoil your data's invariant. Even a simple operation such as `++count_` (where `count_` is an integral type) must be locked. Incrementing is really a three-step (read, modify, write) operation that isn't necessarily atomic. In short, with lock-based multithreaded programming, you need to make sure that any operation on shared data that is susceptible to race conditions is made atomic by locking and unlocking a mutex. On the bright side, as long as the mutex is locked, you can perform just about any operation, in confidence that no other thread will trample on your shared state. In lock-free programming, you can't do just about anything atomically. There is only a precious small set of things that you can do atomically, a limitation that makes lock-free programming way harder [1]. In fact, what would be the minimal set of atomic primitives that would allow implementing any lock-free algorithm—if there's such a set? For example, Herlihy's [2] paper gave impossibility results, showing that atomic operations such as test-and-set, swap, fetch-and-add, or even atomic queues (!) are insufficient for properly synchronizing more than two threads. (That's surprising because queues with atomic push and pop operations would seem to provide quite a powerful abstraction.) On the bright side, Herlihy also gave universality results, proving that some simple constructs are enough for implementing any lock-free algorithm for any number of threads.

Atomic operations are ones which manipulate memory in a way that appears indivisible: No thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic.

Read-modify-write (**RMW**) operations go a step further, allowing you to perform more complex transactions atomically. They’re especially useful when a lock-free algorithm must support multiple writers, because when multiple threads attempt an RMW on the same address, they’ll effectively line up in a row and execute those operations one-at-a-time. Examples of RMW operations include `_InterlockedIncrement` on Win32 platform, `OSAtomicAdd32` on iOS, and in C++11 `std::atomic<int>::fetch_add`. Be aware that the C++11 atomic standard does not guarantee that the implementation will be lock-free on every platform, so it’s best to know the capabilities of your platform and toolchain. You can call `std::atomic<>::is_lock_free` to make sure [3]. The simplest and most popular universal primitive, and the one that I use throughout, is the compare-and-swap (**CAS**) operation:

```
template <class T>
bool CAS(T* addr, T expected, T value) {
   if (*addr == expected) {
      *addr = value;
      return true;
   }
   return false;
}
```

CAS compares the content of a memory address with an expected value, and if the comparison succeeds, replaces the content with a new value. The entire procedure is atomic. Many modern processors implement CAS or equivalent primitives for different bit lengths (the reason for which we've made it a template, assuming an implementation uses metaprogramming to restrict possible Ts). As a rule of thumb, the more bits a CAS can compare-and-swap atomically, the easier it is to implement lock-free data structures with it. Most of today's 32-bit processors implement 64-bit CAS; for example, Intel's assembler calls it **CMPXCHG8** [2].

**Sequential consistency** means that all threads agree on the order in which memory operations occurred, and that order is consistent with the order of operations in the program source code. A simple (but obviously impractical) way to achieve sequential consistency is to

disable compiler optimizations and force all your threads to run on a single processor. A processor never sees its own memory effects out of order, even when threads are pre-empted and scheduled at arbitrary times. Some programming languages offer sequentially consistency even for optimized code running in a multiprocessor environment. In C++11, you can declare all shared variables as C++11 atomic types with default memory ordering constraints. In Java, you can mark all shared variables as `volatile`.

```
std::atomic<int> X(0), Y(0);
int r1, r2;
void thread1()
{
    X.store(1);
    r1 = Y.load();
}
void thread2()
{
    Y.store(1);
    r2 = X.load();
}
```

Because the C++11 atomic types guarantee sequential consistency, the outcome `r1 = r2 = 0` is impossible. To achieve this, the compiler outputs additional instructions behind the scenes – typically memory fences and/or RMW operations. Those additional instructions may make the implementation less efficient compared to one where the programmer has dealt with memory ordering directly. Different CPU families have different habits when it comes to memory reordering. The rules are documented by each CPU vendor and followed strictly by the hardware. For instance, PowerPC and ARM processors can change the order of memory stores relative to the instructions themselves, but normally, the x86/64 family of processors from Intel and AMD do not. We say the former processors have a more relaxed memory model.

A "wait-free" procedure can complete in a finite number of steps, regardless of the relative speeds of other threads. A "lock-free" procedure guarantees progress of at least one of the threads executing the procedure. That means some threads can be delayed arbitrarily, but it is guaranteed that at least one thread makes progress at each step [3].

**Conclusions:**

Lock-free data structures are promising. They exhibit good properties with regards to thread killing, priority inversion, and signal safety. They never deadlock or livelock. In tests, recent lock-free data structures surpass their locked counterparts by a large margin. However, lock-free programming is tricky, especially with regards to memory deallocation. A garbage-collected environment is a plus because it has the means to stop and inspect all threads, but if you want deterministic destruction, you need special support from the hardware or the memory allocator.

**Sources**


1. Non-blocking algorithm [Web – resource], Non-blocking algorithm – access http://en.wikipedia.org/wiki/Non-blocking_algorithm *open access.*
2. Andrei Alexandrescu, Lock-Free Data Structures, DrDobbs, – access http://www.drdobbs.com/lock-free-data-structures/184401865 open access.
3. Jeff Preshing, [Web – resource] An Introduction to Lock-Free Programming, preshing.com – access http://preshing.com/20120612/an-introduction-to-lock-free-programming/ open access.