Removing the Mutex from Mutable Data

Student Daniel J Gunther

Introduction

This report will cover 3 approaches to lock-free data structures, two general, and one specific to the std::vector type in C++. The first is a paper written by Greg Barnes in 1994 & is titled A Method for Implementing Lock-Free Shared Data Structures. It discusses the Herlihy methodology of copying and it's limitations, as well as proposes a second novel approach that combines thread cooperation and local thread caching. They call this the Caching Method [1]. The second was written by Damien Dechev, Peter Pirkelbauer, and the creator of C++, Bjarne Stroustrup in 2006, and is titled Lock-Free Dynamically Re-sizable Arrays. It discusses a lock-free implementation of the std::vector utilizing a global descriptor class coupled with something they dub a write descriptor to allow for concurrent access to the vector [2]. All three are quite unique, and although they may be less practical than a blocking mechanism of managing shared data across multiple threads/processes they are a mind-bending paradigm shift in how we work on applications/systems utilizing more than a single thread or process.

Background - Asynchronous Multiprocessing

Functionally, the objective of asynchronous multiprocessing is to spread work out to an arbitrary number of cores/processors using an arbitrary number of threads, thus parallelizing and speeding up a particular workflow. For thread local data, we don't need to worry as these will behave in a single-threaded fashion. For shared data, we encounter something called a race condition which is when multiple threads access that data and mutate it in unexpected, or undefined ways.

There are two primary philosophies to deal with this: (1 - locking) blocking and isolated single-threading; (2 - lock-free) avoiding the methods in (1) at all costs so that the algorithm is wait-free and non-blocking, thus finishing in a finite number of steps. In locking methods there are two simple approaches: (1) critical sections where we read or write to global data are forced to be single-threaded; (2) global data has a lock attached to it, which blocks other threads and forces them into a busy-wait cycle (spinlock) until that lock is available and it can proceed with its desired operation. This wastes valuable CPU time, and can cause stagnation and starvation over time. In addition, it is not guaranteed that a thread will complete its work in a finite number of steps. There are much more complex locking mechanisms, but at their core, they rely on blocking threads from execution while mutation occurs. The lock-free method is an alternative that relies on atomic hardware instructions to ensure that threads do not interfere with eachother.

Definitions

The following hardware instructions are important to understand lock-free algorithms:

- **Load Linked (LL)** - Reads the contents of a memory location, and tells the thread to continue monitoring that location for modifications by other threads.
- **Store Conditional (SC)** - Verifies that the value at a given memory location has not been modified since the prior load linked at that location, and if it has not been changed, then mutates the data as requested.
- **Compare & Swap (CAS)** - Functionally similar to a combined LL and SC pair, but can have problems regarding the ABA problem, thus an LL and SC pair is preferred.

Lock-Free Methodologies - Barnes [1]

Herlihy's Method

The idea of Herlihy's method is to use rigorous and extensive copying of a data structure to ensure that modification has not occurred. It has an overall time complexity O(T, C) = T + C, where T is the sequential cost of an operation, and C is the cumulative copying time of the data structure. Herlihy's method is executed as follows:

- **1.** An LL instruction is used to read a pointer to the shared data structure (e.g. - head of a linked list).
- **2.** A private copy of the entire data structure is generated locally to the thread.
- **3.** The private copy is modified by the thread.
- **4.** An SC operation is used to change the pointer to the global data to the pointer of the private copy. The arguments of the SC are (1) the load-linked pointer, (2) the current global pointer, and (3) the pointer to the private copy. 1 and 2 are compared, and if they match, the data structure has not been mutated, and the global pointer is replaced by 3.
- **5.** If the SC fails, then the thread completely restarts with a new copy.

Although there are minor optimizations to be made here such as only copying up to the desired modification point in a shared data structure, one can quickly realize how inefficient this algorithm is. The thread must copy the entire data structure at least once per mutation, and with larger data structures and more threads, the probability that the thread must copy multiple times until their global mutation is successful increases. Although this is lock-free as no blocking whatsoever occurs, I would personally choose to use a locking method over this as the time and space required to execute Herlihy's method are undesirable.

Interlude - The Cooperative Technique

The Cooperative Technique (CT) is a precursor to the Caching method as the Caching method is heavily reliant on it to function. Functionally, the CT states that if a thread t_1 wants to write to a data structure, it firsts check if another thread t_2 is working on it, and if so, it dedicates its resources to helping t_2 finish execution. There are two additional definitions:

- **opdesc** - A variable that contains a pointer to the operations wants to complete on some piece of globally shared data.
- **opptr** - A pointer that can either be NULL, or pointing to the address of an opdesc.

The CT defines something called claiming as follows. Consider a set of shared data such as alinked list, where each node is a cell, and to each cell we attach an additional opptr field as metadata. If that field is NULL, then no thread is operating on it currently. A thread claims a cell by placing a pointer to its opdesc in that cell's opptr field. A more intricate breakdown of the claiming process proceeds as the following:

- **1.** An LL instruction is used to read a pointer to some cell D_i and reads the entire record.
- **2.** If the opptr field in that record contains another thread's opdesc, the thread allocates its resources to helping that other thread complete its desired operation.
- **3.** Once the opptr field is empty, the thread creates a new record with an opptr field that points to its own opdesc.
- **4.** Use an SC instruction to set the global cell pointer to the record created by this thread, thus claiming the cell. The SC instruction insures that no other threads have claimed the cell in the meantime.
- **5.** If the SC fails, then restart from step 1 as another thread has claimed the cell.

This cooperation minimizes thread downtime, and thus speeds up the pace at which a task is completed. While the Herlihy method became limited by an increasing thread count, this method benefits from it, and is why the Caching method utilizes it.

The Caching Method

The strategy here is to first break up the data structure into cells as described by the CT. A thread first caches the particular cells it wishes to modify as opposed to the entire data structure, and then modifies the global cells with those modified cells. Already, this is a drastic algorithmic improvement over Herlihy's method. The algorithmic time complexity is O(T, C) = $Tlog_2(T)$. The Caching method is executed as follows:

- **1.** Pull each cell a thread wishes to modify into its private cache using an LL instruction to later check if it has been modified.

- **2.** Perform the operation on the threads private cache.
- **3.** For each cell the thread wishes to modify, claim it using the CT. If any cell has been modified since the original read, release claims on all cells held by the thread, and help the other thread(s) finish and restart.
- **4.** Modify the globally claimed cells, and then release all claims held by the thread.

Without copying the entire data structure for each write the Caching method provides a robust alternative to Herlihy's method that is feasible to use on large systems, with the only added complexity being metadata related to each cell.

Time Complexity Comparison

The full time complexities are as follows for each algorithm:

- **Herlihy** - $Kp(C_{max} + T_{max})$
- **Caching** - $Kp(\alpha_d * s_{max} + T_{max} log_2(s_{max}))$

K is the number of desired operations; p is the number of threads; C_{max} is the maximum number of steps used by the copying algorithm; T_{max} is the maximum amount of time needed to execute a sequential operation on a shared data structure;s_{max} is the maximum number of read write calls done by an operation; α_d is the number of first cells (head of a linked list, or tails of a deque, etc...).

Kp is identical for each, so we must compare the other parameters. Although in the Caching method, we have the additional scaling of log base 2 of the number of read write operations, the copying algorithm is juxtaposed by a multiplication of the number of first cells multiplied by the number of read writes, which is invariably going to be much smaller, especially for large data. Presumably the Caching method will scale much better for larger data and thread pools as the Herlihy method becomes untenable as these scale.

Lock-Free Vector - Dechev et al. [2]

An std::vector in C++ is functionally a generic, dynamically re-sizable array. For single-threaded use cases, it is one of the most useful and important components of the standard library, but they struggle when used and accessed across multiple threads. To combat this issue, the creator of C++ along with two other authors offer a performant implementation of the std::vector utilizing lock-free principles that is thread-safe. Although it doesn't utilize the principles discussed by Barnes [1], it is a fascinating real world implementation of a widely used standard template library data structure.

Desired Operations of the std::vector

The vector needs 6 fundamental operations to be effective. They are as follows:

- **resize** - Resize the vector when the underlying array reaches capacity, and we desire to insert new elements. This operation is internal to the vector itself.
- **push back** - Push an element of type T to the end of the vector.
- **pop back** - Pop the item off of the back of the vector, remove it, and return it to the user.
- **write at** - Write an element to a specific location in the vector.
- **read at** - Read an element at a specific location in the vector.
- **size** - Retrieve the current size of the vector, not the size of the underlying array.
- **at** - Grab the value at a particular location in the vector.

Although the vector has more operations than this, these are the most important ones.

Needed Components

There are a number of needed components in addition to the vector to facilitate it functioning in a thread-safe fashion with a lock-free methodology.

- **Vector** - The vector itself.
- **Descriptor** - An object attached to each vector that contains the size. Functionally a global modification coordinator.
- **Memory Array** - The actual underlying array in memory.
- **Write Descriptor** - A descriptor that is created during a write operation specific to a thread that is contained within the descriptor. It holds information such as the location, as well as the old and new values. This hearkens back to the idea of a CAS.

The following sections will discuss the algorithms described in depth.

Push Back Operation

The algorithm is as follows:

- **1.** Repeat the following instructions until a CAS on the global descriptor is successful (Vector size is unchanged).
- **2.** Grab the global descriptor, and store it local to the thread.
- **3.** If needed, allocate a new bucket of memory in a thread-safe fashion using the memory bucket technique, calling the resize operation if the vector will now exceed the underlying array size.
- **4.** Create a new write descriptor that contains a pointer to the desired memory location, the element to be inserted, and the current size of the vector.
- **5.** Generate a new descriptor that increments the vector size and pass the write descriptor to validate the write.
- **6.** Do the initial CAS. If it succeeds, proceed, otherwise restart at 2.
- **7.** Complete the pending write, and update the descriptor.

Pop Back Operation

The algorithm is as follows:

- **1.** Repeat the following instructions until a CAS on the global descriptor is successful (Vector size is unchanged).
- **2.** Grab the global descriptor, and store it local to the thread.
- **3.** Decrement the local descriptor.
- **4.** Create another descriptor that passes along a NULL write descriptor as we are not writing.
- **5.** Do a CAS on the initial descriptor and global descriptor to ensure no global changes. If this fails, restart, otherwise proceed.
- **6.** Return the element at the end of the vector.

Read & Write Operations

The algorithm is as follows:

- **1.** Read or write to the global object as this operation is atomic and will not be interfered with by other threads.

At Operation

The algorithm is as follows:

- **1.** Grab the position in memory based on the current memory bucket location, plus the desired index.
- **2.** Do some memory shenanigans to actually grab the proper location from the prior value.
- **3.** Return the value at the now calculated memory location.

Size Operation

The algorithm is as follows:

- **1.** Grab the global vector descriptor.
- **2.** Pull the size from the now thread local descriptor.
- **3.** Check if the descriptor has a pending write operation and if so, return the current size decremented by 1.
- **4.** Otherwise just return the current size of the vector.

The ABA Problem

As this lock-free implementation relies primarily on the CAS operation, we encounter something called the ABA problem. The ABA problem is functionally when the underlying object has been modified, but then modified again to look the same as before the initial modification so that other threads don't think it has been modified when it fact it has been. This is dealt with by some very advanced modern C++ 26 features like the hazard pointer but is one of the major obstacles to feasibly implementing lock-free data structures, and is why an LL and SC pair is preferable to CAS.

Conclusions

Lock-free data structures provide an alternative to commonly used blocking techniques such as mutexes, semaphores, read-write locks, etc. Although they were at one time a primarily academic exercise, hardware and software support for them is becoming more prevalent, and a future where modern multi-threaded systems rely on these implementations is now a reality. The three approaches to this presented, two theoretical, and one practical were incredibly interesting to research as they really expanded my perspective on how modern computing systems become complex and untenable and require intentional effort to make sure that they function properly. I hope that the reader of this learned something useful and at the very least enjoyed a mind-bending foray into the world of asynchronous multi-processing.

References

[1] G. Barnes A Method for Implementing Lock-Free Shared Data Structures. *Max Planck Institut Fur Informatik* (1993) 261-270.

[2] D. Dechev, P. Pirkelbauer, B. Stroustrup Lock-free Dynamically Resizable Arrays. *Principles of Distributed Systems: 10th International Conference* (2006) 142-156.