LITERATURE SURVEY

An analysis of the literature regarding lock-free programming techniques and applications.

Ethan McNamara
ethan.s.mcnamara@gmail.com
GitHub: ethan-mcnamara

Table of Contents

1 Introduction

Lock-free programming represents a set of programming techniques for synchronizing multi-threaded programs without the use of blocking techniques such as mutual exclusions (mutex) or semaphores. An algorithm is defined as lock-free if at any given time, there always exists at least one thread making progress. Non-blocking programs are preferable to blocking programs as the potential for deadlocks is removed as there no longer exists the possibility for two threads to cause each other to stall. Lock-free algorithms are often regarded with unease as proving their correctness requires the proof of a linearization point which a technique with which not many developers have familiarity. This report will seek to prove the efficacy and benefit of lock-free programming and demonstrate how linearizability may be used to prove the correctness of lock-free algorithms.

2 Clarification on Terminology

Before the report discussion can begin, it is important to clarify the terminology used in the discussion of this topic. This report focuses on lock-free programming, which is a subset of non-blocking programming. Blocking programming refers to multi-threaded programs which rely on blocking or locking mechanisms to ensure correctness. These mechanisms include mutexes and semaphores, among others. Non-blocking programming solves the same synchronization problems, but without the use of blocking mechanisms, hence the name. The term non-blocking is a general term that includes lock-free, wait-free, and obstruction-free algorithms. Information about each of these will be described in later sections. For the purposes of this report, the terms non-blocking may be used interchangeably with lock-free, except for the case where wait-free or obstruction-free may also apply. In these cases, the term lock-free will be explicitly used. These terms may be used interchangeably due to the frequent comparison to blocking algorithms, as the more general term non-blocking and the more specific term lock-free carry any of the same characteristics when compared to blocking algorithms.

3 Problem Definition

Multi-threaded programs have existed for decades, but their popularity has increased in recent year [1] due to the widespread availability of multi-core CPUs and GPUs, whose full potential can only truly be realized with multi-threaded programs. Also, in 2004 processor clock speed stopped increasing at the exponential rate described in Moore's law, which in 1965 predicted that every two years the number of transistors on chip would double. [2] Chip manufacturers recognized that heat dissipation, power consumption, and current leakage prevented an increase in clock speed without the chip becoming too expensive to produce. [3] Many modern multi-threaded programs are not obviously multi-threaded but instead rely on libraries whose implementations are multi-threaded.

With the rise of multi-threaded programs, the need to ensure correctness has risen accordingly. Multi-threaded programs rely on data structures shared amongst several threads and as such, are liable to race conditions and/or undefined behaviour. Additionally, since there does not exist a single thread of execution a programmer can follow, debugging and developing this variety of programs can be inherently difficult. Many solutions have been developed to conquer the thread-synchronization problem. These solutions largely fall into four categories: (a) blocking, (b) wait-free, (c) lock-free, and (d) obstruction-free.

3.1 Blocking Algorithms

Blocking algorithms are the simplest solution to solving multi-threaded programs. Programs implementing blocking techniques use some form of a lock around a shared data structure. A thread must acquire a lock to modify the shared data structure and subsequently release the lock after the modification is complete. Blocking techniques, like mutexes and locks, aim to eliminate race conditions from multi-threaded applications. Race conditions can occur when two or more threads are accessing a single shared data structure simultaneously and the result of the program will depend on the essentially random nature of which thread completes its operations first.

While locks well in a simple example, problems arise when there are several locks in a program and two threads can each hold the lock the other desires, but neither is willing to release the lock they hold, leading to a situation commonly known as a deadlock. A deadlock will result in neither thread making any progress and thus the program execution will stall.

Additionally, a traditionally common reason to avoid locks has been the fear of thread failure, i.e. a thread dies holding a lock, leading to a permanent deadlock as the now-dead thread can not release the lock. While in theory this could occur, this is not common in practise and a more valid reason against locks is OS pre-emption, which may result in the OS attempting to execute and instruction requiring the lock before the operation releasing the lock which would have negative performance consequences [4]. Also, should a low-priority thread hold the lock desired by several high-priority threads, the low-priority thread may never get an opportunity to execute the release should enough high-priority threads always be present in the queue.

Also, with traditional blocking algorithms, the entire shared data structure is locked when a thread wishes to modify its contents, regardless of how much of the data structure is actually being modified. This can cause significant performance issues as many threads each wishing to make slight edits to the shared data structure can force several other threads to stall while the edit is taking place even if the data the competing threads wish to edit is not shared, but rather simply within the same larger data structure.

The sum of these flaws is the motivation behind non-blocking solutions. Blocking the execution of a thread due to the state of another thread causes degrading performance in larger systems with larger shared data structure that may be accessed by dozens or hundreds of threads simultaneously. Therefore, non-blocking techniques have been developed to solve the same synchronization problem without the need to block threads in the same manner.

3.2 Wait-Free Algorithms

Wait-free algorithms are defined as algorithms that always guarantee that, all threads are making progress. This category of algorithms has no blocking whatsoever and specifies that each thread will complete its task in a finite number of steps, regardless of the actions of any other threads. [5]

In practise, wait-free algorithms are the least common non-blocking implementation due to the complexity that accompanies their design and implementation. The shared data structures must be designed such that no two threads may overwrite the other's modification and such that race conditions do not occur.

3.3 Lock-Free Algorithms

Lock-free algorithms are those algorithms such that at all times at least one thread is making progress. This differs from blocking algorithms in that the potential for a deadlock, which could have zero threads making progress, is removed. Wait-free algorithms are difficult to implement due to their requirement mandating that all threads must be making progress. Lock-free algorithms are, by comparison, easier to implement as they allow for some blocking but are still nonetheless better performing than blocking algorithms.

An algorithm is considered lock-free if infinitely often a thread will complete its task in a finite number of steps. This definition implies that lock-free algorithms are nearly wait-free, except for the unlikely event that two or more threads attempt to modify the same shared data structure, at which point the number of steps of one or more thread increases from the finite value.

3.4 Obstruction-Free Algorithms

Obstruction-free algorithms represent the minimum criteria an algorithm must meet to be considered non-blocking. An obstruction-free algorithm is one designed such that every thread could execute its task in a distinct number of steps if no other threads were executing. That is, if all other threads stop execution at any time, regardless of the state of the other threads, a thread will be able to complete its task in a distinct number of steps. The term obstruction-free was introduced by Herlihy, Luchangco, and Moir in their 2003 paper *Obstruction-Free Synchronization: Double-Ended Queues as an Example*. [6] This style of algorithms was introduced to increase the ease with which non-blocking algorithms can be developed as lock-free and wait-free algorithms are, in practise, difficult to develop. Obstruction-free algorithms retain the benefits of non-blocking solutions, such as the lack of deadlock potential, but allow for easier development.

3.5 Note on Non-Blocking Algorithms

It should be made clear that wait-free, lock-free, and obstruction-free algorithms are not independent sets, but rather represent a hierarchy into which each non-blocking algorithm falls. The lowest bar of entry is obstruction-free, as all non-blocking algorithms are by definition obstruction-free. A subset of obstruction-free algorithms is considered lock-free, while a subset of lock-free algorithms is also wait-free. If an algorithm is wait-free, it is also lock-free and obstruction-free, while the inverse is not true.

4 Common Lock-Free Design Patterns

Lock-free programming, true to its name, does not use locks in its implementations. Instead, atomic instructions (instructions that are guaranteed to occur sequentially without any interruption by other instructions) are used to modify the shared data structures. Atomic instructions are used as they allow threads to modify the data at a single exact instance, without the possibility for another thread to modify the shared data at the same time.

There are sets of atomic instructions used commonly in lock-free algorithms. The availability of each instruction set depends on the manufacturer of the CPU on which the program is run. The compare-and-swap (CAS) instruction is the more popular of the two, with some machines instead implementing the load-linked (LL) and store-conditional (SC) instructions. The two instruction sets offer different functionality but can be used to accomplish similar tasks, with only slight modifications needed to adapt a program from one instruction set to the other.

4.1 CAS Instruction

The CAS instruction compares a provided value with the value stored in a provided memory address. If these two values match, the third parameter is assigned to the memory address. If the two values do not match, the value in the memory address is assigned to the local variable, hence the swap implied in the instruction's name.

The common implementation will involve a do-while loop where inside the do statement the value is read and stored in a temporary variable before writing to another temporary variable the desired new value of the variable. Inside the while statement a CAS function is called that checks whether the value read and stored is still equal to the variable in memory in which case the new desired value is written and breaks from the loop and if not, the loop reiterates.

4.1.1 Issues with CAS

The CAS instruction, in its traditional implementation, can only perform the comparison between a single-word variable and a corresponding single-word memory address. While double-word CAS instructions exist, they are not as common and carry the same issues.

Should manipulation of multiple variables be required in an atomic nature, the CAS function is not the ideal tool since it is only considering the value of a single variable at a time and cannot support the examination of multiple variables at once. [7] Should the atomic manipulation of multiple variables be required, there exists two main strategies which can be used to maintain the use of the CAS function without sacrificing the program's validity. Firstly, one can wrap the variables within a struct or class and manipulate the abstracted data structure rather than the primitive data structures. [3] While this approach does work well for objects of smaller size, when these objects are sufficiently large, locks are used by the CPU when accessing this data as they may require multiple cache reads to fully bring into main memory which may cause unintended performance degradation. Alternatively, one can pack smaller variables into a single 64-bit word which can easily be manipulated atomically. [8] This approach only applies when multiple variables smaller than 64-bits are being manipulated and will require custom logic for manipulating the variables but may be the ideal approach for some programs.

4.2 LL & SC Instructions

The LL instruction atomically loads a word into the L1 cache and tags the cache line on which the data was loaded. This tag represents that this word has not been updated by this thread or any other thread. Should another thread load the same word and modify it after the first thread tagged the word in their cache line, the cache coherency protocol of the CPU would remove the cache line tag. The SC writes a modified version of the word back to the cache line only if the line is still tagged. If the cache line is no longer tagged, the SC operation returns false. By calling the LL operation, modifying the shared variable, and storing the result in a temporary variable and then calling the SC operation to write the desired value only if the cache line is still tagged, non-blocking writing of a variable can still occur.

4.2.1 Issues with LL & SC

An issue experienced only by the LL & SC approach is that cache lines may be untagged for reasons other than the expected data not being present. It is possible that through the background OS processes the cache line becomes untagged, causing an additional iteration of the loop used with this approach.

4.3 Atomic Fetches

While less popular in lock-free programming than the CAS or LL & SC operations, the atomic fetch_add and fetch_sub functions in the C++ standard library can also be used in some lock-free applications. True to their names, these operations fetch a variable, increment or decrement the fetched variable by the provided value, and store the new value to the existing memory location. These operations are all done atomically, so no other instructions may be inserted in between. These functions can be useful if the shared data structure is an integer or a floating point value and whose primary operations are incrementation and decrementation, but with different data types and/or more complex modifications, the techniques mentioned above may be more applicable.

5 Downsides of Lock-Free Design

Lock-free synchronization can be useful for many problems, but there do exist some drawbacks of the programming style. These drawbacks include the issues previously mentioned in Section 4.1.1 and Section 4.2.1 but also include a set of problems to which lock-free programs are susceptible, including the ABA problem.

5.1 ABA Problem

The ABA problem refers to the reading of an object by thread A at one point, the deletion of the same object by thread B immediately after, and then the creation of a new object by thread B which is assigned the same memory location as the just-deleted object by the OS, all before thread A writes to the object in memory. From the perspective of thread A, the object read initially and the object read at the time of writing in the CAS function are identical as they are both the same object type in the same memory address, despite the fact their contents might be very different. The LL & SC approach will not experience this issue as once thread B modifies or deleted the object, the cache line of thread A will no longer be tagged, at which point the SC operation is guaranteed to fail, causing another LL operation to be required.

It is possible to use CAS and avoid the ABA issue, but additional data is required in the pointers to ensure they are not inappropriately reused. One approach involves the use of counted pointers. [4] Counted pointers add a serial number to each pointer class. This serial number is of sufficient size so it is very unlikely that rollover will occur such that two pointers are declared to the same memory address, with the same serial number, all before the first thread has read the address again after the initial read. Since the address and serial number must now match the initial read, it is sufficiently probable that the ABA issue will not occur. A downside of this approach is the additional comparison that must now be done to atomically compare both the pointer and the serial number. Unless the serial number can be especially small, it is likely a double-word CAS operation must be used, which limits the machines on which this program could run as not all machines support a double-word CAS.

6 Proving Correctness of a Lock-Free Algorithm

For an algorithm to be correct, it must have safety and liveness. [4] Safety is the guarantee that "bad things" will not happen and defines liveness is the guarantee that "good things" will eventually happen. Individually, these requirements can be trivially satisfied but together prove an algorithm to be correct. When considering lock-free algorithms, the property that must be proven to prove correctness is the property of linearizability.

6.1 Linearizability

Linearizability is a property which if proven, describes an algorithm whose list of operations can be described in a linear or sequential manner. The execution history of an algorithm (the list of every operation an algorithm executes which for a multi-threaded algorithm may not be a sequential list) is linearizable if it is equivalent (containing the same invocation and return operations with the same argument values) to a sequential execution that respects: (a) object semantics, and (b) "real-time" order. [4] Object semantics refers to the fact that the requirements of the call are met, i.e. a pop operation will always return the top of a stack. "Real-time" order is the property that the return values of operations will reflect the true order in which they were called, i.e. two subsequent pop operations will properly return the initial head value to the first call and the next head value to the second call.

An equivalent explanation of the linearizability property is that every operation must appear to happen instantaneously at a single moment in time. For simpler algorithms, this may be a line of code in which an atomic operation was invoked. For more complex operations, this may not be so trivial. This point in the execution is known as the linearization point. The linearization point is frequently used when proving an algorithm's linearizability.

The principle of linearizability also respects the composability property. This property implies that if structure A is linearizable and structure B is linearizable, then all histories using both structure A and structure B are linearizable.

A common mistake in proving an algorithm's linearizability is believing an operation's linearization point is after it completes its required task. Often, the linearization point is located after an operation has completed a portion of its task but still has additional work to do. This additional "clean-up work" could be done by any thread, not necessarily the thread who performed the work preceding and including the linearization point. Since the linearization point is the point at which the task has instantaneously occurred, the algorithm could be designed that between that point and the true end of the task, other threads may recognize that the task is not fully complete and complete the task. Therefore, when proving the linearizability of an algorithm, it is important to recognize where the dividing line exists between the work including and preceding the linearization point and the "clean-up work". Incorrectly recognizing this point could render an incorrect or invalid proof of linearizability.

6.1.1 Visual Representation Example

To demonstrate the usage of the linearizability property, consider the classic implementation of a stack. A stack is a FIFO (first in, first out) data structure which supports the pushing (adding) of new data, the popping (removing) of the data element at the top of the stack, and the clearing of the stack’s contents. Consider two threads, thread A and thread B, concurrently accessing this stack. Figure 1 below demonstrates the timeline of each thread’s individual interactions with the shared data structure. The dotted line represents the continuous timeline while the black bars represent the time during which the thread is executing the labelled function.

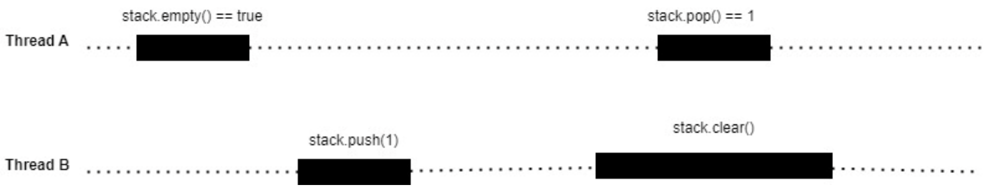

Figure 1: *Thread-independent timelines of interactions with a shared data structure.*

As the above figure demonstrates, undefined behaviour can occur when a shared data structure is implemented without properly considering linearizability. Thread B calls the clear function before Thread A calls the pop function and as such, it should be expected that the clear function is executed before the pop function. However, due to the extended duration the clear function took to execute, Thread A’s pop function was executed and returned a value on a stack that should be empty. This demonstrates that this naïve stack data structure is not linearizable in its current implementation.

If the data structure had been designed with multithreading in mind, it would have incorporated synchronization methods that allowed for an execution history as described in Figure 2, which orders the function calls by the order in which they first called. While this figure is an abstraction of a true data structure, the important principle of linearizability is still evident as every call in the algorithm can be said to execute at a single instantaneous moment in time, rather than over a duration of unknown time.

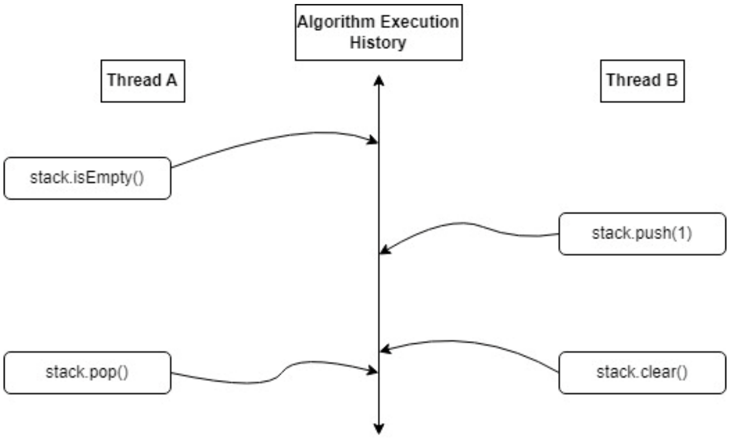

Figure 2: *An execution history of a linearizable stack.*

6.2 Alternative to Linearizability

While linearizability is the main property proved to prove the correctness of lock-free algorithms, there does exist a second property used to prove the correctness of multithreaded programs. Used primarily in database systems, the property of serializability is similar to linearizability, albeit with some notable differences. Firstly, linearizability is the property that every transaction with the shared data structure occurs in a serial manner, i.e. each transaction occurs in between the preceding and proceeding transaction and no transactions are said to occur concurrently. In practise, this is not necessarily true, but it must be proven that this reorganization of transactions would be possible for the system to be proven serializable. This differs from linearizability as there is not a single point at which the transaction occurs, which is a more effective model for a blocking algorithm than a lock-free algorithm. Additionally, as mentioned above linearizability processes the property of composability, which serializability does not. All in all, serializability can be used to prove the correctness of any multithreaded program, including lock-free implementation, but linearizability is better suited and more commonly used to prove the correctness of lock-free algorithms.

7 Lock-Free Data Structures

While any data structure can technically be used in a lock-free algorithm, there exist a set of data structures adapted specifically for lock-free programming. This genre of data structures optimizes their data modification mechanisms to use CAS or LL & SC instructions such that optimum performance is realized. It can also be useful to reuse lock-free data structure in new programs as their proof of correctness can be relied upon to prove the correctness of the new program.

7.1 Treiber Stack

The Treiber stack was first presented by R. Kent Treiber in his 1986 article "Systems Programming: Coping with Parallelism". The Treiber stack is a scalable lock-free stack which uses CAS operations to perform its push and pop operations. The pseudocode below is taken from the slides of Michael Scott's video lecture [4] to demonstrate how the Treiber stack is implemented.

```
class stack
  <node*, int> top

void stack.push(node* n):
  repeat
    <o, c> := top
    n->next := o
  until CAS(&top, <o, n>, <n, c>)

node* stack.pop():
  repeat
    <o, c> := top
    if o = null return null
    n := o->next
  until CAS(&top, <o, c>, <n, c+1>)
  return o
```

While it is known that this data structure is lock-free, it is not necessarily true that this algorithm is correct. To prove the correctness of this algorithm, the linearization points of each of the functions must be determined. For the push operation, the linearization point is the line on which the CAS function is called. Due to the atomic nature of the CAS function, the element to be added to the stack is added exactly at the successful calling of the CAS function. Should that function fail, the loop will repeat at which point the previous calling of the CAS function is no longer the operation's linearization point. For the pop operation, the linearization point will depend on whether there exist elements in the stack or not. If the stack is empty when the pop operation is called, the linearization point is the if o = null return null line. This is because at this point the pop operation has completed its work and can be said to have executed at that exact moment. If the stack is not empty, the linearization point is the first successful calling of the CAS function, like the push function as this is the moment at which the shared data structure is modified and the operation has completed its task. Unlike the push function, the pop function does have additional instructions after its linearization point, namely the returning of the element popped. According to Scott, this work is known as "clean-up work" and could theoretically be done by any thread who encounters this unfinished work, although with an example this trivial it is unlikely that would occur.

Because the linearization point of every operation utilized in this algorithm has been identified, it can be said that this algorithm is correct.

8 Conclusion

Lock-free programming is a technique that can provide a significant benefit to an application's performance, provided it is implemented properly, and the appropriate considerations are made to ensure that lock-free programming is indeed the ideal optimization technique for the problem at hand. While it does avoid many of the pitfalls incurred by other multi-threaded programming styles, it does come with its own that must be handled accordingly to guarantee the application under development is running efficiently and properly.

9 References

[1] F. Feinbube, P. Tröger and A. Polze, "Joint Forces: From Multithreaded Programming to GPU Computing," in IEEE Software, vol. 28, no. 1, pp. 51-57, Jan.-Feb. 2011, doi: 10.1109/MS.2010.134.

[2] G. E. Moore, “The Future of Integrated Electronics,” Electronics Magazine, Apr. 1965.

[3] H. Sutter, “The Free Lunch is over: A fundamental turn toward concurrency in software,” Dec-2004. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm. [Accessed: 23-Oct-2022].

[4] Michael Scott. "Michael Scott — Nonblocking data structures. Part 1.," YouTube, Oct 14, 2019. [Video file]. Available: https://youtu.be/9XAx279s7gs. [Accessed: Oct 23, 2022].

[5] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. Burlington, MA: Morgan Kaufmann Publishers, 2008.

[6] M. Moir, M. Herlihy and V. Luchangco, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in 2013 IEEE 33rd International Conference on Distributed Computing Systems, Providence, Rhode Island, 2003 pp. 522.

[7] Tony van Eerd. "Introduction to Lock-free Programming - Tony van Eerd,” YouTube, Sep 24, 2018. [Video file]. Available: https://youtu.be/RWCadBJ6wTk. [Accessed: Oct 23, 2022].

[8] Fedor Pikus. "CppCon 2015: Fedor Pikus PART 1 “Live Lock-Free or Deadlock (Practical Lock-free Programming),” YouTube, Oct 9, 2015. [Video file]. Available: https://youtu.be/lVBvHbJsg5Y. [Accessed: Oct 23, 2022].