P1 - Concurrent Data Structure Implementations

180001964

July 6, 2022

1 Introduction

The aim of this practical is to create several different concurrent queue implementations, including at least one which makes use of mutex locks, and one which is non-blocking. I must then argue that all these implementations are memory safe in a concurrent setting, and test the performance of these implementations, in several scenarios.

2 Background Concepts

In this section I will use *thread* to refer to both software threads, hardware threads, and async tasks.

When referring to two threads carrying out a set of instructions at the *same* time, I mean that the order in which the instructions are executed relative to the other thread is not well defined. This refers to separate hardware threads executing simultaneously, and both software threads and async tasks which may be preempted.

2.1 Concurrency

2.1.1 Mutex Locks

A mutex lock is a simple way of preventing two threads from accessing the same memory location at the *same* time. They work by acting as a flag that only one thread can hold, and a thread must wait until it holds the flag to be able to perform operations on the shared memory.

2.1.2 Atomic Operations

Atomic operations guarantee the order in which they are executed relative to other atomics, even across multiple threads. They also guarantee that a thread is not preempted while executing the operation, and that no other thread can access an atomic value as it is being changed. This means they can be used to ensure that shared memory is accessed coherently between multiple threads. In C++, these take the shape of atomic types which support operations such as `compare_exchange`, which allows a thread to swap items at two memory locations in a way that is guaranteed that no other thread can interfere.

2.1.3 ABA Problem

The ABA problem must be considered when using the compare_exchange atomic operations. It refers to the scenario when you are watching a variable to monitor for changes, in order to perform some action when a change occurs. If the value of the variable when you first check is A, and then when you check again is still A you will continue to wait, however you may have missed that in the intervening period between checks, another thread changed the value from A to B, and then back to A. This is normally not an issue, as the action will just be delayed until the next time the variable changes state, however, in lock-free data structures, sometimes careful consideration is required to deal with this.

2.2 Cache Coherency

One way in which modern computers increase the speed at which they run processes is by making use of the cache. The cache is relatively small amounts of memory, usually a few 10s of MB on modern CPUs, stored in the CPU itself. It aims to store sections of memory which are used frequently by a process, in order to avoid having to routinely fetch memory from RAM, as cache is significantly faster for the CPU to access. Programmers aiming to make use of this, in order to speed up their programs must be careful to improve the *spatial* and *temporal* locality of the memory they are accessing. Data structures and algorithms making use of contiguously allocated arrays are often more performant than their equivalents making use of linked-lists, even when a linked-list may provide better theoretical time-complexity. This is because when traversing a linked-list, the address of the next node cannot be known until the current node is processed, known as the pointer-chasing problem, in addition adjacent nodes in such a list, are often not adjacent in physical memory, as they may be dynamically allocated at different times throughout the program. These properties of the linked-list limits the effectiveness of hardware prefetchers (Yang et al., 2004), resulting in worse performance as mentioned.

3 Design and Implementation

In order to aid with testing and benchmarking multiple implementations of the same data structure, I made use of the c++20 feature *concepts*. This allowed me to create example programs which could easily be run for all implementations, and checked for correctness at compile time, by declaring a `Queue` *concept*, which outlines the functions a `Queue` must support. Then, by adding a static assert to the default constructor of my queue implementations, the compiler checks that the implementation supports all required queue operations.

3.1 Coarse Queue

The coarse queue is the simplest concurrent queue implementation, which uses a single mutex lock to ensure that only one thread can be altering the data structure at a given time. This guarantees memory safety, but is not particularly concurrent, as although multiple threads can use the same queue, any operations on the same queue can not be executed concurrently, requiring one thread to wait for another. One potential benefit of this approach, however, is that it imposes no requirements on the data structure which backs the queue. This allows for more cache conscious data structures, such as contiguous arrays, to be used rather than the linked lists required for the other implementations. In many cases, this means that this queue may in fact be more performant, despite its lack of true concurrency. I use the cpp `std::deque` as the backing data structure. The `std::deque` behaves like a doubly linked list, but with better cache performance as it is comprised of multiple linked contiguous arrays.

3.2 Fine Queue

The fine queue uses two mutex locks, one for the head of the queue, and another for the tail. The `std::deque` does not expose enough of its internals through its api, for thread safety to be easily reasoned about when not locking the entire data structure. For this reason, I chose to implement this queue using a linked-list which is controlled entirely by the class. The downside of this approach, is that linked-lists are generally performant than similar data structures using contiguous arrays due to their cache behaviour. It may be possible to implement from scratch a data structure similar to a deque which can be made thread-safe using two locks, but this would be considerably harder, as the deque is a more complicated data structure.

3.3 Lock-Free Queue

The lock-free queue has a similar design to the finely locked queue, using a linked list with a dummy node to allow threads to enqueue and dequeue safely at the same time. In order to avoid the need for mutex locks to protect against threads trying to perform the same operation at the same time, atomic operations are used instead to control the order in which threads access and mutate the pointers to nodes of the queue. They also guarantee that writes cannot happen at the same time as other write or read operations.

The use of atomic operations makes it difficult to reclaim the memory being used by the nodes when they are dequeued. This is because even if the head has been dequeued by one node, another thread also trying to dequeue might still be holding it, and will not have advanced to the next node until it manages to carry out a `compare_exchange` operation. This can be remedied in a number of ways, such as through the use of hazard pointers (Michael, 2004), reference counting, or garbage collection. This increases the complexity of the queue implementation, and slows down the execution of the program. I have chosen not to worry about memory reclamation during the life-time of the queue, instead freeing each node when the queue is destroyed. This is achieved through keeping a pointer to the original head, and, since the head is simply discarded when items are dequeued, the pointers to the next nodes remain valid, allowing the entire linked-list to be iterated over, freeing each node in turn. This is not a concern for infrequently used, or shortly lived queues storing only small elements, but for other use cases, this is a potential memory leak which could potentially lead to an out of memory error. Another upside to not freeing nodes during the lifetime of the queue, is the ABA problem does not need to be considered. If the memory of old nodes is reused it is possible for a node to have the same pointer when `compare_exchange` is called, but refer to a different place in the queue. For example, in `LFQueue::enqueue`, after the tail has been swapped with the new node, we link the old_tail to the new_node.

4 Theoretical Evaluation

4.1 Coarse Queue

This queue is trivially thread-safe, as both functions which access the underlying data are locked by a single mutex. This means that only one thread may be accessing the data of the queue at any one time. Though this is thread-safe, it lacks concurrency. I would expect this queue to perform the worst in highly concurrent scenarios, with multiple producers and consumers. However, since the underlying data structure is more cache-conscious, and the method by which thread-safety is achieved is simpler, this queue will probably perform the best in instances with only a few producers and consumers.

4.2 Fine Queue

The fine queue is an iterative improvement to concurrency over the coarse queue, allowing threads to enqueue, and dequeue concurrently, though multiple occurrences of the same operation are not concurrent. If two threads are performing the same operation, the mutex lock ensures that the second thread must wait for the first one to finish before touching the shared memory. This means we only need to consider the case when two threads are enqueuing and dequeuing concurrently to check for correctness. The `enqueue` operation only accesses the tail node, where as `dequeue` accesses both the head and the next node. In the case of three or more nodes, it is thread-safe as `enqueue` and `dequeue` cannot be accessing the same node. For fewer than three nodes, more careful analysis is required. `Enqueue` changes the value of the tail pointer, but this is safe as it is only read and changed in `enqueue`, where it is protected by the tail mutex. `Enqueue` also writes the next pointer in the node pointed to by the tail. Though `Dequeue` also interacts with this node, it never reads or writes to the next pointer. The node at the head is only freed when dequeued, which can only happen if there is more than one node, so `enqueue` can also not be writing to the next pointer as the node is being freed. This means the queue is thread safe.

4.3 Lock-Free Queue

A major downside to lock-based concurrent data structures, is that if a thread is suspended while holding a lock, it prevents other threads from accessing the data structure and completing their own operations. This can happen if the scheduler preempts a thread in the middle of an enqueue or dequeue operation, while it is still holding a mutex. The effects of this may be heightened if the scheduler preempts a thread holding a lock in favour of another thread trying to hold the same lock, preventing either from doing any work at all, while still incurring the cost of context switching. This can be overcome by lock-free data structures which rely on atomic operations rather than locks. Although these are often slower to process one operation, scheduled threads can always make progress, even if other threads are not running.

4.4 Wait-Free Queue

It is important to note that, although the lock-free queue means that threads are not blocked from making progress by other threads holding a lock. It is still possible for threads to be blocked from making progress. If every time a thread is scheduled, a second thread has already succeeded in executing a compare and swap operation on the same node in the intervening period, the first node must then spend its time changing the node it is trying to swap. This means that, even if the scheduler deciding which threads get to run is fair, this does not guarantee that the threads actually can make progress. It is possible to design a wait-free queue which does not have this problem by creating some mechanism by which threads can register that they are attempting to do some work, and if another thread notices that this work has not been done, take over and complete it (Kogan and Petrank, 2011).

5 Testing and Benchmarking

I chose to use the cpp catchorg (2022) framework for testing, as it provides a number of macros which make writing test cases and assertions simple across multiple similar types. I originally also attempted to use its benchmarking functionality, but found it problematic as it does not recreate the data structure before each test iteration. Instead, I opted to write my own benchmarks, using the cpp standard library `high_resolution_clock`, and templated functions to reuse code, along with *concepts*, for type-safety.

5.1 Testing

I wrote both single-threaded and multi-threaded tests for all my queue types. The catch2 framework provides type parameterized tests which allows me to run tests across all my different queues with no code duplication. The single-threaded tests ensure that basic queue properties are respected for all my implementations, and the multi-threaded tests attempt to create conditions conducive of data races and detect when one occurs. It was not particularly easy to trigger violations of thread-safety, especially when the queue is used with small types such as `int`.

My four multi-threaded tests do pass for my thread-safe queues, and fail when the single-threaded queue is used, this provides me with some confidence that my implementations are correct. They are certainly more thread-safe than the standard deque.

Due to non-thread-safe data structures generating undefined behaviour when modified concurrently, the multi-threaded tests tend to fail in a variety of unclean ways when using a single-threaded queue. For this reason, the single-threaded queue is not tested on these tests by default, instead to verify that it fails, it must be added to the `THREAD_SAFE_QUEUES` tuple.

In addition, to the multi-threaded tests I also used the compiler option `-fsanitize=thread`, which helps detect data races. This helped me detect a potential, but extremely unlikely data race in `FQueue::deqeue` when the next pointer in the head node is checked. To fix this `FQueue::dequeue` must also hold the tail mutex when this check occurs. As this only checks for a null pointer, this will only impact the program if `enqueue` is not finished writing the value of the pointer when the check occurs, meaning it will be non-null, and `dequeue` advances, but is not finished writing this value when `dequeue` reads it on the next operation. Despite two mutexes being held, there is no possibility for a deadlock, as any thread holding the tail mutex will always be able to make progress. I am unsure if this data race can actually create undefined behaviour on a modern cpu, as it is a single 8 byte read/write, and if it is being read at the same time as it is being written to in the check in `dequeue`, this does not matter as we only care if it is non-null, so no undefined behaviour can occur. For undefined behaviour to occur, the `enqueue` thread must have not finished writing after the check in `dequeue` has occurred, which seems extremely unlikely, though if this value is cached then problems might occur. Despite not observing any problems in my tests, I chose to make the change for correctness.

There was also a similar problem in my lock-free queue implementation, converting the next node pointer in the `Node` struct to also be atomic and carefully choosing the memory ordering for the atomic operations fixed this issue.

5.2 Benchmarking

In addition to the previous queues, I also introduced a *linked coarse queue*, which uses a thread-safety mechanism equivalent to the coarse queue, while having an underlying data structure of a linked-list, similar to the fine queue. This allows me to demonstrate both the performance improvements of the concurrent fine queue over a coarse queue, while also demonstrating that in many cases the performance of the underlying data structure is a grater factor than the concurrency.

In an effort to increase the accuracy of the results, the benchmarks are carried out with simultaneous multithreading disabled, and with a minimum number of background processes, including no active desktop environment or windowing server.

The benchmarks were carried out on a desktop pc with a Ryzen 5 2600x CPU with 6 cores clocked at 3.6Ghz, and 16GiB of DDR4 with a frequency of 2133Mhz.

Queues	Enqueue		Dequeue	
	mean	std dev	mean	std dev
Single Threaded	2ns	0.5	1ns	0.0
Linked Coarse	32ns	0.5	12ns	0.0
Coarse Grain	12ns	0.5	6ns	0.0
Fine Grain	33ns	2.8	15ns	0.0
Lock-Free	10ns	0.0	17ns	2.1

Table 1: Single Threaded Benchmarks

Queues	One-One		Five-One		One-Five		Two-Two		Three-Three	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std de
Coarse Grain	2080ms	930	967ms	84	1010ms	42	1370ms	78	1610ms	100
Linked Coarse	1700ms	160	1760ms	100	1770ms	90	2840ms	100	3870ms	230
Fine Grain	4290ms	780	1620ms	38	1620ms	40	2100ms	130	2700ms	100
Lock-Free	1660ms	190	414ms	8.9	403ms	12	943ms	29	778ms	23

Table 2: Multi Threaded Benchmarks

Table 1 shows that all methods of ensuring thread-safety incur a cost to performance, with a singly locked queue being the fastest on average over enqueue and dequeue operations. Much of this performance is down to the superior data-structure used.

Table 2 shows that the finely locked queue offers no performance gains over the coarsely locked queue backed by the `std::deque`, across many multithreaded scenarios. The lock-free queue performs the best, especially with unbalanced enqueue and dequeue threads, though I suspect much of this performance lead would disappear if the lock-free queue implementation freed its memory rather than leaking it.

6 Evaluation and Conclusion

From the results it is clear that there is no one best thread-safe queue implementation for every possible scenario. Instead, when choosing an implementation, one must evaluate the needs of the system in which they are using it. If the environment is not highly concurrent, then it is likely that the best queue is one which uses just a single mutex lock.

It is also not enough to rely on tests to verify the thread-safety of your data structures, as it is difficult to reliably trigger and detect unsafe behaviour. At best, it is possible to verify that some data structures are more thread-safe than others, but to be certain of thread-safety requires careful analysis of the data structure, and the conditions under which it is used.

References

catchorg, H. (2022). Catch2 cpp testing framework. `https://github.com/catchorg/Catch2`.

Kogan, A. and Petrank, E. (2011). Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46(8):223–234.

Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504.

Yang, C.-L., Lebeck, A. R., Tseng, H.-W., and Lee, C.-H. (2004). Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(4):445–475.