University of St Andrews

CS4204 - Concurrency and Multi-Core Architectures

Assignment: P1 - Concurrent Queue Implementations

Matriculation Number: 160001362

March 2020

Contents

1 Overview

1.1 Aim of Practical

The aim of this practical was to develop two concurrent queue implementations. One making use of locks (such as mutexes) and another, lock-free version, making use of atomic primitives to update the data structure. An analysis of the correctness of each implementation was then to be carried out, followed by the benchmarking of each data structure. Finally, the results of these benchmarks was to be analysed.

2 Execution Instructions

This practical was implemented using C++ 17.

The compilation and execution instructions for the program are available in the provided README.md file in the submission directory, but are also presented below:

2.1 Compilation

1. Open a terminal window in the 'CS4204-P1-Concurrent-Queue/src' directory
2. Run the command 'make' to run the Makefile

2.2 Execution

1. Open a terminal window in the 'CS4204-P1-Concurrent-Queue/src' directory
2. Run './locking' to run the locking implementation or './lockfree' to run the lock-free implementation

After running each of the implementations, a series of benchmarks will begin running which output the time taken for individual enqueue/dequeue tests to run. The results of these tests are also written to files in the 'CS4204-P1-Concurrent-Queue/data/' directory.

3 Design Decisions

3.1 Locking Queue

The locking implementation of the concurrent queue is located within the 'src/locking_queue/' directory. The 'lockingQueue.h' file contains the implementation of a templated class CQueue. This class contains two attributes, the first of which is a lower level linked list

data structure which stores the queue elements, and the second of which is a mutex which is used to lock the enqueue and dequeue functions of the class, providing sequential consistency through mutual exclusion.

For simplicity, and to ensure that the abstract type signature of the CQueue class closely resembled the signature in the specification, the enqueue and dequeue functions are very simple. They contain an instruction to ensure that the mutex is locked (and unlocked once the function returns), and either a call to data.insert() or data.remove() to insert or remove data from the linked-list data structure for enqueue or dequeue respectively.

The actual logic of the linked-list which defines the queue's first-in-first-out behaviour is implemented within the file 'linkedList.h'. The linked-list class logically represents a sequence of linked-list nodes (defined in 'linkedListNode.h') containing an element of type T and a pointer to the next node in the list. The linked-list contains three attributes, head, tail, and temp. Head is a pointer to the front node in the list, tail is a pointer to the rear node in the list, and temp acts as an initial 'dummy' node, to prevent head and tail pointing to NULL. This node's next pointer initially points to NULL. The standard which is used to judge whether the list is empty is if the next pointer of the node pointed to by head is NULL (i.e. the next pointer of temp will initially be NULL).

The two main points of interest in the LinkedList class are the functions insert and remove, which add a node to the back of the linked-list and remove (and return the value of) the node at the front of the linked-list.

The algorithms of the insert and remove functions are as follows:

void insert(T element):

```
create a new node
set the new node's payload to element
set the new node's next pointer to NULL

set the current next pointer of tail to the new node

set the tail to the new node
```

T remove():

```
if the next pointer of head is NULL:
    throw an exception as the queue must be empty
else:
    //i.e. retrieve value of node after the dummy node
    get the payload of the next pointer of head

    //make the next node the new dummy node
    set head to the next pointer of head

return the stored payload
```

3.2 Lock-Free Queue

The lock-free implementation of the concurrent queue is located within the 'src/lock_free_queue/' directory. Like the locking queue implementation, the lock-free implementation is implemented using an underlying linked-list data structure.

The 'lockFreeQueue.h' file contains the implementation of the lock-free CQueue class. This class is identical to the locking version, except now there is no mutex and no calls to a mutex in the enqueue and dequeue functions, as the concurrent logic is now contained within the insert and remove functions of the linked-list.

The 'linkedListNode.h' file is also identical, aside from the change to make the next pointer of a node atomic (using std::atomic).

Aside from head and tail being made atomic, the major differences in the lock-free version of the LinkedList class in 'linkedList.h' are the implementations of the insert and remove functions to make them thread safe by utilising the atomic compare-and-swap primitive 'std::atomic_compare_exchange_weak_explicit'.

The algorithms of the lock-free insert and remove functions are as follows:

void insert(T element):

```
create a new node

repeat while insertion not successful:
    store current value of the tail

    //updates tail next pointer if no other thread is in process of inserting
    atomic {
        if (tail next pointer is NULL -> set tail next pointer to newNode):
            break out of loop
    }

/* Should always succeed as all other threads
should still be in loop as previous CAS should fail */
atomic {
    set tail pointer to new node
}
```

T remove():

```
repeat while deletion has not succeeded:
    store the value of head
    store the value of tail
    store the value of the head next pointer

    if (head equals tail):
        if (head next pointer is NULL):
            throw exception as queue must be empty

        //Otherwise tail pointer must not yet
        //have been updated in insert function
        //after first item inserted
        atomic {
            //update tail
            set tail to head next pointer
        }
    else:
        atomic {
            if (head is unchanged):
                update head to head next pointer
        }
return payload of the next pointer of head
```

3.3 Testing and Benchmarking

Within the main files of the locking queue and lock-free queue (locking.cpp and lockFree.cpp respectively), code has been implemented which tests the correctness of each respective queue implementation as well as benchmarking its performance.

To assist with this process, four data structures have been declared. The first two are both CQueue data structures of type int, called 'in_queue' and 'out_queue', both of which are used by the threads during benchmarking to enqueue and dequeue threads. The 'resultSet' data structure is a std::set of type int whose members must be unique. This set is used to ensure that the correct number of elements remain after the threads have performed the process of enqueuing and dequeuing. Finally, 'threads' is a std::vector of type std::thread which is used to store all of the spawned threads as well as being used to refer to them when being joined.

The main algorithm to test the queue solutions is as follows:

```
for 5 repeat readings:
    create a file to write the data for the current repeat reading to

    //Amount of items to enqueue/dequeue at each data point
    loop from 1000 to 500,000 in increments of 1000:
        determine the amount of work to be performed by each thread

        loop until all threads have been spawned:
            //Give the thread a test function to run as a parameter
            //and the range of values it should enqueue/dequeue
            create a thread and add it to the list of threads

            calculate the range of values for the next thread

        for each thread in the thread list:
            join the thread

        calculate the total duration the threads have been running for

        record the values enqueued/dequeued in a result set for correctness

        ensure the out_queue is empty by determining if its size is 0

        write the size of the result set and the duration to file

        print the size of the result set and the duration
```

```
        clear the list of threads and the set of results
```

The algorithm of the test function run by each thread is as follows:

```
for each value to enqueue:
    enqueue the current value to the in_queue

dequeue until as many items have been removed as were enqueued:
    if dequeue from in_queue is successful:
        enqueue value to out_queue
    else if queue empty (exception):
        keep trying to dequeue current value until other thread enqueues
```

The purpose of the threads enqueuing the values to the out_queue is so the main thread can validate the dequeued values using the result set once the threads have all joined. If the number of data items added to the result set matches the workload peformed cumulatively by all of the threads, then the number of data items enqueued/dequeud should be correct.

4 Problems Encountered

Difficulties were encountered early on in the development process of the lock-free CQueue implementation. These difficulties included the solution deadlocking and segfaulting. These issues were due to the fact that the initial solution used a linked-list which when empty had the head and tail pointing to NULL (instead of a dummy node).

This meant that for every single removal where the linked-list became empty, both the head and tail would have to be updated to NULL (if the head and tail were NULL, both would also have to be updated to a new node in the case of an insertion). This was not possible to achieve atomically as the head would have to be updated using compare-and-swap, followed by the tail in a separate compare-and-swap. This meant that between the head being set to NULL and the tail being set to NULL, a value could be inserted at the tail, resulting in the head being out of date as it should then logically point to the new node.

The use of a dummy node ('temp') in the current solution solved this issue. It allows for the above scenario to occur as it ensures that head will always have access to any newly inserted nodes via the next pointer of the current dummy node. There should never be a scenario where references to newly inserted nodes are logically 'lost' due to the head being NULL.

5 Performance Analysis

5.1 Correctness

5.1.1 Theoretical Analysis

It is obvious that the locking implementation of the CQueue data structure must be correct due to the use of a mutex before entering the insert or remove function of the underlying linked-list data structure. This ensures that only a single thread can enqueue or dequeue a value at any time, with every other thread blocking until the mutex is released.

It is less obvious why the locking implementation is correct. However it can be explained if we examine a few common scenarios of threads concurrently enqueueing and dequeuing.

Two Threads Enqueuing:

If two threads try to concurrently enqueue, one of the threads will update the next pointer of tail to a new node, and will break out of the while loop as the compare-and-swap operation will succeed (as tail next is detected to be NULL). This will cause the other thread's compare-and-swap operation to fail as the value of the next pointer of tail is no longer NULL. This will cause that thread to repeatedly try to insert a new node into the linked-list until the other thread updates the value of tail to the new node (which will cause tail next to now be NULL again, allowing the second thread to proceed and insert the second new node). This shows that only a single thread will be able to add a node to the linked-list at any time.

Two Threads Dequeuing:

If two threads try to concurrently dequeue (assuming two successful enqueue operations have already occurred) then both threads will execute the else block within the conditional statement in the while loop of the linked-list remove function. As with the insert function, one of the threads will perform a successful compare-and-swap operation, resulting in an update of head to the head next pointer. This will cause the compare-and-swap operation of the other thread to fail, requiring it to iterate again through the loop and to perform another compare-and-swap operation. At this point the successful thread can return the payload of the removed node. Finally because the successful thread has returned from the function and is not updating the head pointer, the unsuccessful thread should be able to successfully update head to head next and then return its payload.

The situation is very similar if there is only one item in the queue. The first thread will successfully remove the item at the end of the linked-list and return its payload as before, but now the unsuccessful thread will enter the if condition (as head and tail are now equal), will detect that head next is NULL, and will throw an exception as it is trying to

dequeue from an empty queue.

One Thread Enqueuing, One Dequeuing:

Due to the FIFO nature of the data structure, the majority of the time the insert function will only be concerned with the value of the tail of the linked-list, while the remove function will be concerned with the head. The edge case which requires attention is when the list is empty (and head and tail are both equal to the dummy node) and so the possibility for concurrent accesses to a single node by multiple threads from the insert and remove functions may occur.

Consider the case where one thread is in the process of enqueueing and another thread begins the process of dequeueing. If the dequeue occurs first, the thread dequeuing will simply throw an exception as it will detect the list to be empty. The enqueuing thread can then simply enqueue as normal. However if the enqueuing thread begins first and updates the value of the tail next pointer to a new node but has not yet updated the value of tail, the dequeuing thread may still detect that the head of the list equals the tail of the list and will assume that the list is empty.

This has been accounted for by a check in the if condition of the remove function that checks whether next is NULL. If next is not NULL, then the thread in the remove function knows that an insert is taking place (and that the list is not actually empty) but that the tail pointer has not yet been updated. The dequeuing thread will then update the tail pointer on the behalf of the enqueuing thread. If this compare-and-swap operation fails, it means that the enqueuing thread has now updated the tail pointer, otherwise the dequeuing thread has updated it. Either way, the dequeuing thread will then the loop again, and this time should detect that head and tail are now not equal, which indicates that it can try to set head to head next using compare-and-swap (effectively dequeuing the added node).

5.1.2 Experimental Analysis

In addition to the above theoretical explanation of the correctness of the CQueue implementations. The benchmarking code in locking.cpp and lockFree.cpp ensures the correctness of these implementations experimentally. The dequeuing of values from out_queue to the resultSet data structure ensures that the values enqueued and dequeued by the threads to in_queue are preserved. This is the case as a set can only contain unique values, and so if any values are lost, the set will contain less values than expected and will throw an exception when attempting to dequeue. A try-catch statement has also been added to the main of locking.cpp and lockFree.cpp which attempts to dequeue one more item than should be in the resultSet. If an exception is thrown when dequeuing this item (as is expected as out_queue should at this point be empty) then the size of the out_queue is set to zero and is printed during the benchmarking process.

5.2 Results

Figure 1 shows the time taken for each CQueue implementation to finish enqueuing and dequeuing increasing numbers of integers to and from the in_queue (and then enqueuing them onto the out_queue) using 1000 threads. The results shown in the graph for each implementation are an average of the times produced by five repeat readings. The raw data used to plot figure 1 is available in the file 'results_1000_threads.ods' in the 'CS4204-P1-Concurrent-Queue/data/' directory.

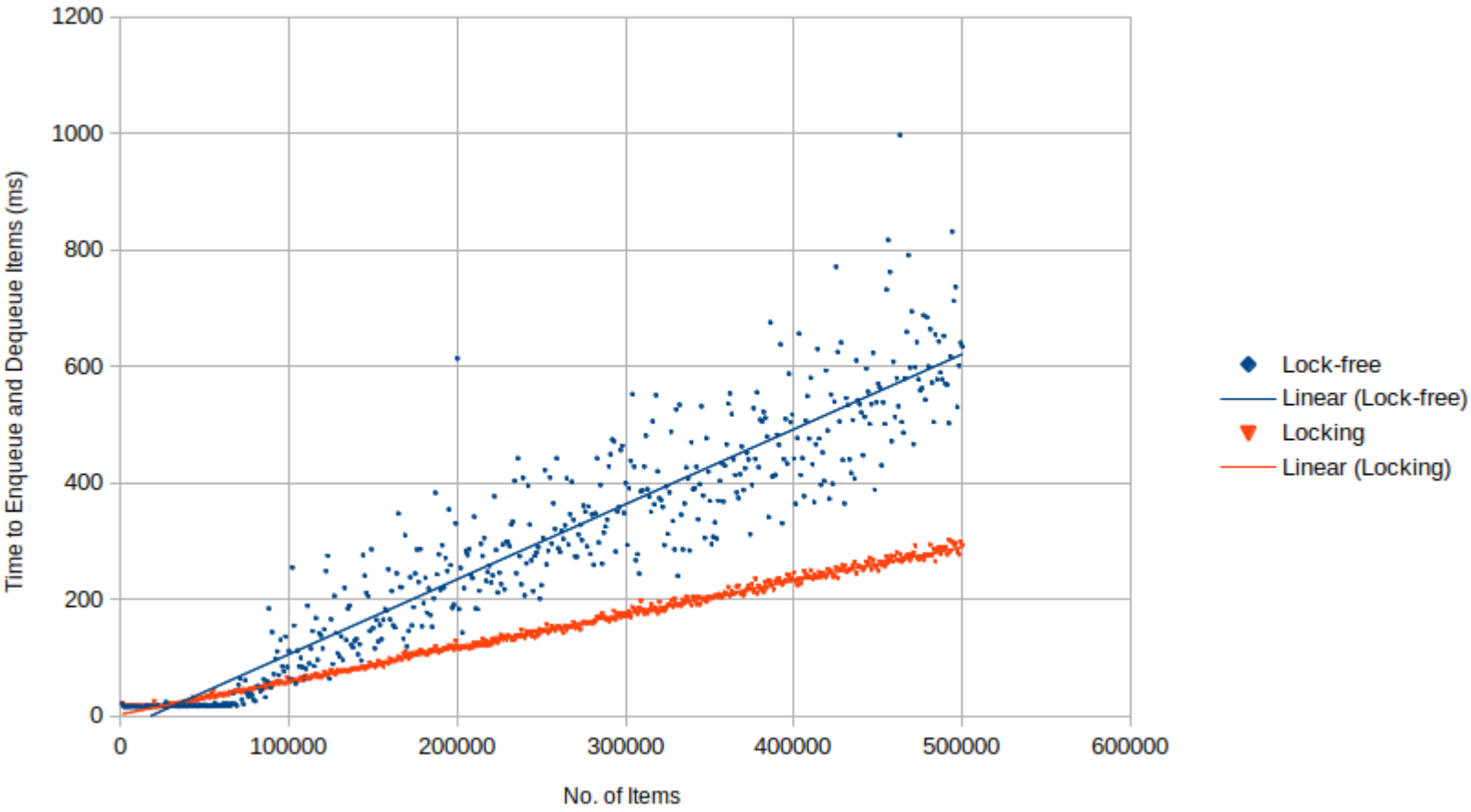

Figure 1: Time taken for 1000 threads to enqueue and dequeue increasing numbers of queue elements using each CQueue implementation

Figure 2 shows the time taken for each CQueue implementation to finish enqueuing and dequeuing increasing numbers of integers to and from the in_queue (and then enqueuing them onto the out_queue) using 4 threads. The raw data used to plot figure 2 is available in the file 'results_4_threads.ods' in the 'CS4204-P1-Concurrent-Queue/data/' directory.

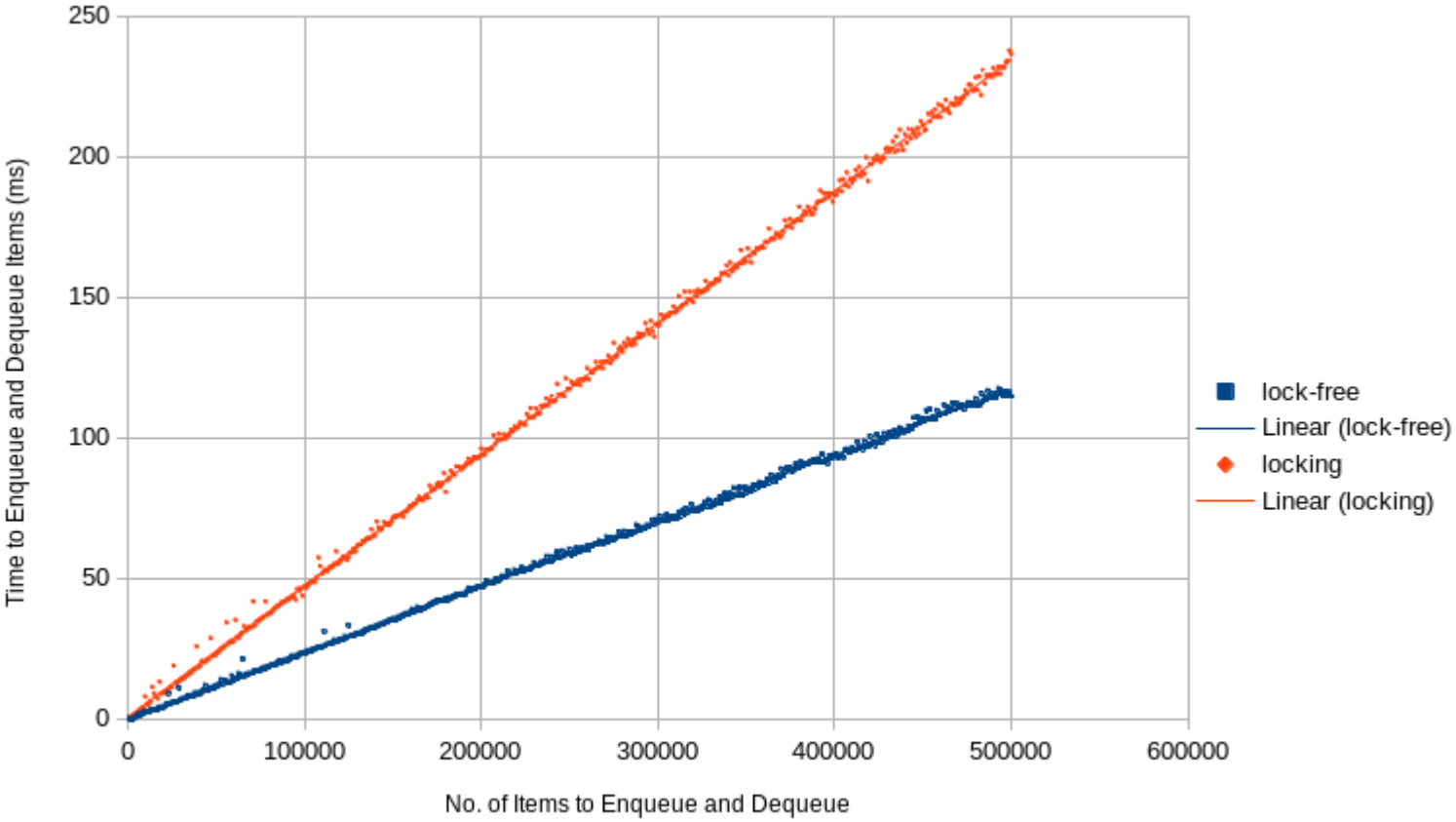

Figure 2: Time taken for 4 threads to enqueue and dequeue increasing numbers of queue elements using each CQueue implementation

5.3 Comparison of Solutions

5.3.1 Performance Characteristics

From figure 1, we can see that both implementations suffer from a linear increase in run-time when presented with additional items to enqueue and dequeue. This is to be expected for a fixed number of threads due to the increased workload for each thread. Perhaps surprisingly, the lock-free implementation suffers from a greater increase in run-time compared to the locking approach for the same increase in the number of items to enqueue/dequeue. Intuitively, one might hypothesise that the lock-free approach would match if not exceed the performance of the locking implementation due to threads not having to block while waiting for a mutex to be freed.

It is likely that this phenomenon has occurred due to the use of 1000 threads running on a quad-core lab machine which would therefore have to context-switch in the lock-free

implementation, but not the locking implementation (due to all threads but one being in a blocking state waiting for the mutex to be released). As such, an additional set of tests were conducted, this time running with 4 threads in an attempt to minimise the overhead caused by context-switches (by attempting to run one thread per core). These results are presented in figure 2. They confirm what was initially hypothesised - that the rate of increase of the run-time of the lock-free solution is less than that of the locking solution when the number of items to enqueue and dequeue is increased.

It makes sense that the lock-free implementation would run faster than the locking implementation if, for example, we consider two sequential enqueue operations. If one thread has created a new node and is in the process of performing a compare-and-swap operation, another thread can concurrently be creating its own new node which it will enqueue after the first node is finished. The creation of this node by the second thread would have to be performed completely after the first thread is finished in the locking solution and has unlocked the mutex.

Perhaps the results would differ if multiple mutexes were used, one for the head of the linked-list and one for the tail. This would in theory allow the locking solution to perform better than it does currently due to the potential for threads to then enqueue and dequeue concurrently. Given more time the locking solution would be modified to investigate this approach.

5.3.2 Tradeoffs of Each Approach

Aside from the performance increase suggested by the results of figure 2, the lock-free CQueue implementation also has the benefit that because it is not blocking, deadlock is prevented in the case where a thread holding the mutex goes to sleep or dies, in other words, the delay of one thread does not delay others.

However, speaking from personal experience having implemented this practical, one could argue that the performance increase of the lock-free implementation is outweighed by the far greater time investment to ensure that the use of the atomic primitives is suited to the task and that the resulting implementation is correct. In addition, most programming languages provide well tested and supported mutex implementations, where as fewer will provide atomic primitives that are supported universally on all architectures.

6 Conclusion

In conclusion two concurrent queue implementations have successfully been developed. The first being a locking implementation, making use of a mutex to control access to the critical sections of the underlying queue implementation, and the second being a lock-free implementation, using a compare-and-swap like C++ primitive to ensure that

updates to the queue data structure are performed atomically. Benchmarking was also performed, showing the lock-free implementation of the concurrent queue having a slight performance edge over the locking implementation. Given more time, other variants of the locking queue would be implemented, such as a version making use of separate head and tail mutexes. It could be insightful to implement this and compare its performance to both the original locking solution as well as the lock-free solution.