Concurrent Data structures

Hello, and welcome to this lesson on locks and concurrent data structures.

This lesson will be about how to use locks, or maybe better, how to avoid locks whenever possible.

We will study in detail the implementation of a concurrent list with ordered elements.

We will start with a naïve, coarse-grained locking based implementation. We will see that this implementation does not scale to many threads or processors.

We explore a sequence of improvements to this basic coarse-grained locked list, starting with a fine-grained implementation, and a so-called optimistic implementation, which will follow the motto that "it is easier to ask for forgiveness than for permission". This approach will significantly reduce the amount of locking.

This can be further improved by tweaks in the data structure.

Finally, we will briefly look at a lock-free implementation of the list. Lock free implementations can become quite complicated, so we limit ourselves to lists that only support addition of elements and that don't have a remove function.

At the end of this lesson you will have seen a number of sophisticated ways to make data structures scalable for use across many threads and many processors.

Let's begin.

Example: Concurrent Lists

Let's start with a simple linked list, which we call **CoarseList** because we will use coarse-grained locking.

In this pseudo-C++ notation, the list supports, in addition to the constructor, two functions to add items and remove items.

Internally, the list is a list of nodes, with a head pointer, and a lock that ensures mutual exclusive access to the list.

The constructor makes evident the structure of the implementation.

We initialize the lock, and we initialize the list, with a head, and a tail node.

Each node is associated with a key, and the nodes in the list are ordered in increasing key value. The first node has key MIN_VALUE, and the tail has key MAX_VALUE.

CourseList: add

Let's see how we add to this data structure.

First, we use a lock to turn the entire code into a critical section. Now we can freely operate on the list without having to worry about interference.

We initialize two pointers, called "predecessor" and "current" to the head and the first node in the list.

We then traverse the list by shifting the two pointers one node at time until the key of the current node is larger than the key of the new item. We notice that we cannot fall off the end of the list because the tail's key is MAX_VALUE, and therefore cannot be exceeded.

We assume that we fail, and check whether an item with the new key exists already.

If it does not: - we create a new node to contain the new item, - we insert the node into the list by fixing the pointers accordingly, - and we declare success. - at the end we return the whether we succeeded or not.

We notice that this implementation is basically identical to a sequential one, except that we add a critical section, using the lock associated with the list.

CoarseList: remove

The function to remove items is implemented similarly.

- We turn the code of the function into a critical section by using the list lock.
- Again we initialize the predecessor and current pointers.
- and we traverse the list until we find a matching key.
- Again, we assume failure,
- and compare the current key with the key of the item to be removed.
- If the two match we delete the item from the list by short-cutting the next pointer of the predecessor to point to the node after the node to be removed.

In code: we set predecessor -> next to current -> next.

- we declare success.
- After releasing the lock we return.

Note that, in order to keep the code simple, we rely on a garbage collector to clean up memory.

The problem, of course with this solution is that both the functions add and remove lock the list for an extended period of time. As the number of concurrent threads using a list increases, the amount of blocking shoots up, and the performance suffers.

A way needs to be found to reduce the amount of blocking due to locks.

Conc. Lists with Fine-Grained Synchronization

One way to do that is to replace the course-grained locking, which locks the entire list a one time, buy fine grained locking, which locks small portions of the list at a time.

This give rise to **FineList**, which is similar to **CoarseList**, except that it does not have a list-wide lock.

Instead, we have a single lock per node in the queue.

FineList: add

How would we add items to the queue when we have per-node locks? In this approach, rather than locking the entire queue, we lock two nodes at a time. As we traverse the queue, we lock the next node before us and release the locks of nodes that we just traversed.

We initialize the predecessor and the current pointers.

We then lock only the predecessor and the current nodes.

As we traverse the list searching for the place to insert the item, we keep releasing the lock of the node that we leave behind and locking the next node. In this way we have only two nodes of the entire list locked at any time.

When we find a place to insert the item, we don't need to worry about additional synchronization, because the we are inserting the new node between "predecessor" and "current", which are both locked. No other thread can interfere.

Once we are done, we unlock and return.

FineList: remove

The remove function works very similarly. . We initialize the predecessor and current pointers, and lock the first two nodes in the list.

We then traverse the list, looking for a matching key. Just as in the add function, we unlock the node that we leave behind and lock the next node as we traverse the list.

If we find a node with a matching key, we remove it from the list by pointing around it, just as in the case of the course-grained list.

We unlock the two locked nodes, and return.

FineList: remove - Why 2 Locks?

One question that may come up is why we need to lock two nodes in this data structure. After all, we are modifying only the link of the predecessor when we add or remove an item. Why can't we just lock the predecessor? The reason for this becomes apparent when we look at what can happen when multiple threads remove an item.

For this, let's assume that we have a list and two threads, A and B.

- The first thread, traverses the list,
- and just reaches the place where it wants to delete Node A.
- It locks the predecessor node.
- Now the thread gets preempted and thread B takes over.

It also traverses the list

- and reaches the place where it wants to delete node B. (5)
- It locks the predecessor, which is free.
- Thread B gets preempted and Thread A resumes again.

Thread A now delete the node from the list.

- After Thread B later deletes node b from the list,
- Node A has been deleted, but Node B is still in the list.

This cannot happen when we lock both the predecessor and the current node.

Let's look at a slightly different scenario, which would cause difficulties with a single node locked.

- when thread B reaches Node B, which it wants to delete,
- it locks the predecessor.

If it gets interrupted and Thread A reaches Node A as it traverses the list, Thread A locks the predecessor as well at some point. In the last iteration of the while loop where it looks for a matching key it also updates predecessor and current, and frees the previous predecessor node, which we don't worry about here. (5)

It attempts to lock the new current node. It fails because the current node is A, which is the predecessor of B and has been locked by the other thread.

- Now thread B continues, and acquires the lock for Node B as well.
- It deletes node B from the list,
- and gives up the lock for the predecessor node A.
- Thread A now continues and acquires the lock for the current node A. (10)
- and deletes node A from the list.
- It releases the lock on the predecessor node and returns.

FineList: Hand-over-Hand Locking

This form of pair-wise locking of elements is called "hand-over-hand" locking.

It works because, in order to initiate an operation on the list, the thread has to acquire two neighboring locks.

A second thread has to acquire two neighboring locks as well. If the two threads want to modify portions of the list that are too close, they will compete for at least one lock. In this way, we can ensure the correctness of the operations.

The problem with the fine-grained locked list is that we still have to lock all the nodes individually as we traverse the list trying to find where to add or remove a node. When the list is long, this makes for a lot of locking activity. We remember from our previous discussion on lock implementations that lock operations are expensive even when there is no contention.

Optimistic Lists: Forgiveness vs. Permission

Let's see if we can reduce the number of lock operations by using an optimistic approach.

The fundamental premise of optimistic approaches to concurrency is that it is often easier to ask for forgiveness than for permission.

Let's see how this work. We call our new list **OptimisticList**. The structure is identical to the fine-grained list, except that we have an additional function, called **validate**. We get back to this function in a minute.

OptimisticList: add

Let's look at the **add** function.

- We first assume that we fail, and that we will be working on this for a while.
- We will retry adding the item until we succeed.
- We traverse the list until we have found the spot to add the item. We notice that we are not locking any nodes. This traversal is therefore very fast.

Once we have located the spot, we lock the predecessor and the current node. We unlock them later.

Once these two nodes are locked, and cannot be touched by other threads, we check whether they are valid. We will see later that the function **validate** checks whether the predecessor still points to the successor, and if the predecessor is still part of the list.

If this is the case, we are basically done.

If the key does not exist in the list already we add the new item and declare success.

If predecessor and current are not valid, we do not set "done" to true, and repeat the while loop until we find a valid pair.

Once we are done looping we return.

Let's look a bit more closely at the **validate** function.

As we said earlier, the validate function checks whether the predecessor next pointer still points to current, and whether the predecessor is still reachable from head, in other words, whether the predecessor is still in the list.

The function looks as follows: - We start with the head. - and we iterate through all the nodes with keys smaller or equal to the predecessor's key. - If the the node is equal to predecessor, that is, we have found the predecessor in the list, we check whether the predecessor next pointer still points to current. If it does, we return true, and false otherwise. - If we reach the end of the list without finding the predecessor we return false.

With this very simple function we cut down tremendously on the overhead to traverse the list.

OptimisticList: remove

The remove function for the optimistic list is again similar in structure to the add function.

Similarly to the add function, we assume failure, get ready for a lot of work, and start looping until we eventually succeed.

We look for a node with the matching key without locking any nodes.

Now we lock the predecessor and the current node and release the locks later.

We validate whether the locked predecessor and current node are still valid, or if some other thread interfered between the link traversal and the locking.

If the nodes are valid, we are basically done,(5) and we delete the node if we indeed did find the item in the list.

If the nodes did not validate, we retry until we are done.

After we are done we return.

OptimisticList: Why Validation

Let's look at an example that illustrates why are are validating the predecessor and the current node.

Let's start with this list, Thread A wants to delete the node with key "a". Thread A therefore traverses the list looking for "a". As it does so, it moves the pair of predecessor and current pointer increasingly to the right in this figure. At some point, maybe just when Thread A reached Node "a", some other thread, either on the same processor or on another one, removes one or more nodes from the list, including Node "a". The other thread is free to do so, as Thread A holds no locks on these nodes.

We recall that Thread A validates the predecessor and current nodes before proceeding with deleting Node "a". As part of that it detects that, while the predecessor node is still part of the list, its next pointer does not point to current. Thread A therefore know that some other thread interfered, and so restarts the search for and removal of Node "a".

While this validation saves us a lot of locking, it has the cost that we need to traverse the list twice, once when we search for the node and a second time when we validate. For very long lists, this is a lot of traversing that has to happen. There are tricks that allows us to cut down on the cost of validation.

Lazy Lists: Mark Nodes

As is often the case, there are ways to trade off memory against work.

This is the case with the lazy list, where we mark nodes that are being removed. The structure of the list remains the same as before, except that each node now has a mark.

LazyList: remove

This time, let's look at the remove first. The overall structure remains the same as for the Optimistic List.

We set up for retries.

We traverse the list to find a matching key.

We lock and validate.

If validation succeeds and if we indeed found the item we delete the node from the list.

Let's note however that we also mark the node as deleted.

Once we are done we return.

Again, the function remove is identical to the one in OptimisticList, except that we mark the node as deleted.

The function **validate** now becomes totally simple.

We check whether the predecessor is marked as deleted, whether the current node is marked as deleted, and whether predecessor next points to the current node. Maintaining the field "marked" to mark deleted nodes saves us from traversing the entire list to validate the two nodes.

LazyList: add

The implementation of the add function is no different than the one for Optimistic List except that we can use the much more efficient new validate function.

LazyList: contains

One of the other nice benefits of marking deleted nodes is that it allows for a very efficient function to query whether a particular item is contained in the list.

All function has to do is to traverse the list until it finds a matching node, and then return false if the matching node is marked as deleted.

LazyList: add

One problem that remains, and here we are looking at the "add" function again, are those pesky lock operations.

Would it be possible to make do without any locks at all?

This is what lock-free data structures are about.

Lock-Free Lists: Vanilla Attempt

Let's try to illustrate the general approach of lock-free data structures by implementing a vanilla version of a lock-free list. In order to keep the implementation somewhat simple, our LockFreeList does not support the removal of item. We can only add items to the list.

We don't need to mark deleted nodes in this version of the list.

LockFreeList: add

The structure of the add function is somewhat similar to that of the optimistic list.

We start re-trying, so-to-say.

As many times before, we initialize predecessor and current, and traverse the list until we find a spot to insert the new node.

If the key exists already we return false.

Otherwise we create a new node and insert it into the list. We don't do this using elementary data structure operations however, but do it a bit differently. . First we the the new node next to current. This is not new.

In order to set the predecessor next to new, we use the Compare-and-Swap instructions, however.

We recall that compare-and-swap takes three arguments, namely old, new, and a memory location. It then compares old with the value at the address.

If the two match, it assigns new to the memory location and returns true. In our case this would be when current is the same as predecessor next; if this is the case, it assigns new to predecessor next and returns true. We are done.

If predecessor next is not current, some other thread must have interfered, and compare-and-swap returns false. If so, we retry by repeating the while loop.

LockFreeList: add in Action

Let's illustrate with a diagram how this works.

Thread A wants to add node "new" to the list.

It starts traversing the list to find a spot for the node.

Let's assume that it found such a spot.

It starts inserting the node by setting its next pointer to current.

Now it calls the compare-and-swap instruction to compare whether predecessor next still points to current, and if so, it makes it point to new.

If the result is true, the new node has been successfully added.

If before the compare-and-swap instruction, another thread has inserted a node before current, predecessor next does not point to current anymore, but rather to the newly inserted node.

Compare-and-swap therefore fails, and we restart with the insertion of the new node.

As we start again, we traverse the list to find the spot to add the new node.

Now compare-and-swap returns true, telling us that the node has been inserted.

Conclusion: Locking and Data Structures

With this brief excursion into lock-free data structures we have come to the conclusion of this lesson on Locking and Data Structures.

This lesson was about how to use locks, or maybe better, how to avoid the over-use of locks, with the intention to improve the performance of the system.

We have looked in detail at the implementation of a concurrent list with ordered elements.

We started with a naïve, coarse-grained locking based implementation. This implementation does certainly not scale well for large numbers of threads and processors.

We improved the implementation by replacing the coarse, list-wide lock with node-based fine-granularity locks. This implementation reduced contention, but the amount of lock/unlock activity was very high, and thus likely to affect performance.

The optimistic list, applied an "asking for forgiveness rather than permission" approach, by first traversing the list without consideration of other threads, then locking the predecessor and current nodes, and only then checking if the nodes actually were even part of the list anymore and connected anymore. This significantly

reduced the amount of locking, cut required that the list be traversed twice, first for finding a spot and then again to validate that the nodes still make sense.

We looked at the Lazy list as a way to trade off memory space against work by marking the deleted nodes. The validation now does not have to traverse the list anymore to check whether nodes are still in the list. Rather it simply checks whether the node is marked as deleted.

Finally, we had a brief glimpse at so-called lock-free data structures. These use atomic instructions to avoid locks. We have implemented a simple lock-free list that only allows additions. Allowing items to be removed causes all kinds of complications, most prominently the so-called A – B – A problem.

We truly hope that you enjoyed this exploration into concurrent data structures, and how to build data structure such that they scale to large numbers of threads and across many processors.

Thank you for watching.