Lock-Free Linked List (Part 3)

published: Sun, 13-Aug-2006 | updated: Sat, 25-Aug-2018

<< Part 2

So, last time, just like a cheesy Saturday morning movie serial, I left you hanging, wondering how I was going to solve the insoluble in C#. The insoluble issue in this case is the fact that we have two items of information that will vary independently of each other in different threads, but that we need to see as a unit when we write new values of them using CAS. And, of course, the size of both items is more than the size we can CAS.

The answer, for I shall reveal all straight away just like the next episode in a movie serial, is to create a class that holds both items of information, make instances of this class immutable, and then use that. Welcome to the NodeState class:

```
internal class NodeState<TKey, TValue> where TKey : IComparable<TKey> {
  private readonly bool isDeleted;
  private readonly ListNode<TKey, TValue> next;

  public NodeState(bool isDeleted, ListNode<TKey, TValue> next) {
    this.isDeleted = isDeleted;
    this.next = next;
  }

  public bool IsDeleted {
    get { return isDeleted; }
  }
  public ListNode<TKey, TValue> Next {
    get { return next; }
  }
}
```

This class maintains two pieces of information: the logical deletion state and the reference to the next node. Furthermore, once the fields are set by the constructor, they cannot be changed. The instance is immutable. If we want to change a reference to an instance of this class, we have no alternative but to replace the instance completely.

So how is this used? First of all we create a ListNode class that has a single instance of NodeState.

```
internal class ListNode<TKey, TValue> where TKey : IComparable<TKey> {
  private KeyValuePair<TKey, TValue> pair;
  private NodeState<TKey, TValue> state;

  public ListNode() : this(default(TKey), default(TValue)) {
    state = new NodeState<TKey, TValue>(false, null);
  }

  public ListNode(TKey key, TValue value) {
    pair.Key = key;
    pair.Value = value;
  }
  ...
```

I've shown the two constructors, the default one of which creates a node state instance to hold the current state. Nodes are created as not logically deleted, which makes sense. (Note also that I'm using the newer naming conventions for the type parameters; Dustin Campbell convinced me to give it a try.)

Let's now see how to logically delete a node.

```
private bool CasState(NodeState<TKey, TValue> oldState, NodeState<TKey,
  return SyncMethods.CAS<nodestate<TKey, TValue>>(ref state, oldState,
}

public void FlagAsDeleted() {
  NodeState<TKey, TValue> newState;
  NodeState<TKey, TValue> oldState;
  do {
    oldState = state;
    newState = new NodeState<TKey, TValue>(true, oldState.Next);
  } while (!CasState(oldState, newState));
}
```

The CasState() method is there merely to perform the CAS operation on the state instance of a node: code is easier to read when using it than when using the full SyncMethods.CAS() method. Anyway, let's now look at the FlagAsDeleted() method.

We start a traditional CAS loop (in other words, make a local copy of what we want to change, and then try to set the new version, and repeat until we succeed). We make a copy of the current state (an atomic operation since it's a reference). We create a new state variable that is marked as deleted and that has the same Next reference, and then try to CAS this new state variable. Because the state variable is immutable, the only way the state of a node can change is if the variable itself gets replaced.

Once we have marked the node as logically deleted, we have to ask its parent node to actually delete itself.

```
public void TryDeleteChild(ListNode<TKey, TValue> child) {
  NodeState<TKey, TValue> oldState = state;
  if (oldState.Next == child) {
    NodeState<TKey, TValue> newState = new NodeState<TKey, TValue>(oldS
    CasState(oldState, newState);
  }
}
```

Here the code is written to not actually care if the node is physically deleted, because the rest of the class will be written to cooperate in unlinking logically deleted nodes. If you remember we did the same kind of thing with the lock-free queue where the Enqueue() method would "help out" in moving the tail pointer when it determined that it could.

Why we're doing this? The problem with this operation that we're solving by this cooperative algorithm is that a new node could get linked in by another thread as the parent's child. We try once to unlink, and if we fail, oh well, some other thread will do it eventually. Here's the GetNext() method that does the physical unlink whenever it's following the node chain (a node's state is inaccessible outside the instance, so calling this method is the only way for the linked list to get at the next node along):

```
public ListNode<TKey, TValue> GetNext() {
  ListNode<TKey, TValue> node = state.Next;
  while ((node != null) && (node.state.IsDeleted)) {
    TryDeleteChild(node);
    node = state.Next;
  }
  return node;
}
```

As you can see, it's written to unlink logically deleted nodes as it comes across them when following the Next links. Again, it doesn't matter if it fails in one of these unlinking operations: it'll just try again immediately.

Finally, the last node operation is to add a new node after a given node:

```
public bool InsertChild(ListNode<TKey, TValue> newNode, ListNode<TKey,
  NodeState<TKey, TValue> oldState = state;

  if ((!oldState.IsDeleted) && (oldState.Next == successor)){
    NodeState<TKey, TValue> newState = new NodeState<TKey, TValue>(fals
```

```
    newNode.state = new NodeState<TKey, TValue>(false, oldState.Next);
    return CasState(oldState, newState);
  }
  return false;
}
```

This method tries once to link the new node as the next. We take a copy of the current node's state. If it is not logically deleted and the Next node is equal to the successor node passed in (in other words, no one has managed to insert a new node as we were trying to), we create a new state instance for this node, set the new node's state to our next node, and try to CAS the new state. If we succeed, all fine and dandy, the method returns true, otherwise it returns false.

Having seen the node implementation, we can now take a peek at the linked list's implementation. First, the constructor:

```
public class LockFreeLinkedList<TKey, TValue> where TKey : IComparable<
  private ListNode<TKey, TValue> head;

  public LockFreeLinkedList() {
    head = new ListNode<TKey, TValue>();
  }
  ...
```

Nothing too interesting here: in essence we create a head node from which we can hang the rest of the linked list. There is no tail node since a Next reference of null indicates the end of the list.

Now let's see how to find a given node (that is, the node with the given key) in the list. I've written a set of interlocking methods here to make the Add() and Delete() methods easier.

```
private bool NodeHasSameKey(ListNode<TKey, TValue> node, TKey key) {
  return (node != null) && node.IsEqualToKey(key);
}

private ListNode<TKey, TValue> FindNode(TKey key, out ListNode<TKey, TV
  ListNode<TKey, TValue> dad = head;
  ListNode<TKey, TValue> node = dad.GetNext();

  while ((node != null) && node.IsLessThanKey(key)) {
    dad = node;
    node = dad.GetNext();
  }
  parent = dad;
  return node;
}

public bool Find(TKey key, out TValue value) {
  ListNode<TKey, TValue> parent;
  ListNode<TKey, TValue> node = FindNode(key, out parent);
  if (NodeHasSameKey(node, key)) {
    value = node.Value;
    return true;
  }
  value = default(TValue);
  return false;
}

public TValue Find(TKey key) {
  TValue value;
  Find(key, out value);
  return value;
}
```

The second method listed here is the workhorse: it tries to find the key by walking the nodes in the list. If the key was found, it will return the node with the equal key as the function result, as well as its parent as an out parameter. If the key was not found, it will return the successor node as function result, together with the node that would act as the parent if the key that was not found were to be added. The premise here is that if the key were to be added, it would be inserted between these two nodes.

Then there are two public Find methods. The first returns a boolean indicating whether the key was

found, and if it was, the value associated with the key. The other one, either returns the default value if the key were not found, or the value associated with the key if it were.

To find and delete the key from the linked list:

```
public void Delete(TKey key) {
  ListNode<TKey, TValue> parent;
  ListNode<TKey, TValue> node = FindNode(key, out parent);
  if (NodeHasSameKey(node, key)) {
    node.FlagAsDeleted();
    parent.TryDeleteChild(node);
  }
}
```

Here we make use of the two node methods we've already seen.

To add a new key and value to the linked list:

```
public void Add(TKey key, TValue value) {
  ListNode<TKey, TValue> parent;
  ListNode<TKey, TValue> node;
  ListNode<TKey, TValue> newNode = new ListNode<TKey, TValue>(key, valu

  do {
    node = FindNode(key, out parent);
    if (NodeHasSameKey(node, key))
      throw new InvalidOperationException("Key already exists in linked
  } while (!parent.InsertChild(newNode, node));
}
```

This is slightly more complicated. We try and find the key in the list. If we find it, we signal an error by throwing an exception. If we don't find it, the FindNode() method will return the node that will precede our new node, as well as the node that will follow it. We call the new parent's InsertChild() method to try and insert the new node. This will fail if the parent becomes logically deleted in the meantime, or if the successor node we found is no longer the real successor.

If it does fail, we go round the loop again and try and find the place to insert the new node anew. In other words, if we fail to insert our new node, we start our search over from the beginning of the linked list.

This last caveat can be a real problem. Next time we'll take a look at how to minimize the amount of searching we have to do again when we fail to insert a new node.