Lock-free linked list (part 1)

published: Thu, 27-Jul-2006 | updated: Thu, 27-Jul-2006

Every now and then I just need to do some programming. Not only that, but something I haven't done before, preferably translating some weird code from something like an academic paper or from a book by Knuth into .NET and C#. So something algorithmic or data structure-y: I want to continue learning and being stretched.

And then I read a comment by CornedBee in The Daily WTF, replying to a series of posts about what to ask people in interviews:

> OK, perhaps I'm stupid, but what's the point of implementing a sorted linked list? How often do you need a container that has O(N) insertion complexity, O(N) lookup complexity for an arbitrary element, and happens to deliver the elements in sorted order on iteration? That's one of the most pointless data structures I've ever heard of.

Not so, I'm afraid. A good answer came immediately to me: a lock-free sorted linked list. Well, OK, so it was on my mind, having written a series of articles on simple lock-free data structures, but it's a valid use of a sorted linked list, especially as implementing a lock-free sorted dictionary with any normal data structure (a red-black tree, say) has not been fully or reasonably solved. In fact, once we have a lock-free linked list, we can fairly easily implement a lock-free skip list which is a pretty good basis for a sorted dictionary, as I discovered after researching lock-free linked lists.

So far I've described and inplemented a lock-free stack, a lock-free queue, and a lock-free simple priority queue, and it's been pretty easy, mainly because we didn't have to give the user access to our internal linked lists. But with a standard linked list, one of the operations is going to be iterating over the nodes in the list. Not only that, but at the same time that nodes are being inserted and deleted by other threads. And without a lock in sight.

What I ultimately want to do is to define and implement a lock-free sorted dictionary, where the implementation uses a linked list internally. There should be four main operations: insert an item/key pair, delete an item given its key, find an item given its key, and iterate over the items in sorted order by key. In fact, to make things simple (and, as you will see in a moment, that "simple" deserves an unqualified *Ha!*), we'll just talk about a single linked list.

First, I'll define some ground rules. A node will look like this in C# using generics, at least for now:

```
class Node<K,T> {
  Node<K,T> Next;
  public K Key;
  public T Item;
}
```

That is, there's a Next field pointing to the next node in the linked list and there are generic fields to hold the item of type T and the key of type K. Furthermore, for simplicity, the linked list will have a Head node and a Tail node that are allocated when the list is created and never change (although Head.Next obviously will).

Now let's think about the problems that will arise in a multi-threaded environment and that we'll have to solve.

We'll first imagine that we have two threads running at the same time, say on a single CPU. If both threads are just walking the linked list, following Next links, there's no problem: we have a simple readonly list. So, let's be more perverse: the first thread is scanning through the linked list, node by node, following the Next links. It's arrived at the node before X (and so we assume that it has read the pointer to X that's stored in the Next field), and is about to dereference the Next link and get node X when it's preempted in favor of the second thread. The second is about to delete node X (cue an evil laugh), and does so with impunity. Sometime later, many microseconds later, aeons later, the first thread resumes. What happens now?

It depends. In a non-garbage-collected environment, if the second thread had deallocated the node after it deleted it from the list (as we would normally do), we will be in big trouble. Essentially we'll be dereferencing a Next pointer that points to deallocated memory in the heap. Or worse that points inside some structure that's already been allocated off the heap. A big no-no. In a garbage-collected environment, we're fine providing that we don't have code that tries to reuse the node (much as I implemented in the lock-free stack and queue).

So, the first point to make is that we can't reuse nodes when we delete them: there may be many threads walking the list that have already made copies of nodes that have been deleted and that are just about to dereference that Next link.

(We can't dispose of nodes either in a non-garbage-collected environment, since many threads may be holding onto them and processing them. Yuk. So, from now on in this article, I'm going to ignore all that and assume that we have a garbage collector. One day, I'll talk about the other case.)

Let's posit another scenario. We're inserting a node after node X in one thread and, you guessed it, deleting node X in another. Let's call our new node, N. What we'd like the insert to look like is this:

```
N.Next = X.Next;
X.Next = N;
```

The first line of code there is pretty uneventful. We've presumably walked the linked list to X and its successor and determined that we need to insert our new node in between them. We've already said that we can't mess around with the node X on a delete operation, and so all that line of code does is to copy X.Next and to assign it to the Next value for our local, unshared node N. At this point, no other thread is aware of what we're doing.

It's the next line of code that's going to mess us up. If X hasn't been deleted yet, there's still a link from X's parent to it and the linked list will still be complete after the change and the new node will be present and visible to all threads.

If X has been successfully deleted from the list though, we are in a world of hurt. There will be no link from any node in the linked list to X (that's what it means to be deleted), and our new node will not be visible, except from X. Those threads that know about X will see our new node, at least temporarily, until they've walked on by, but no other threads will ever see it.

Next time, we'll talk about how to solve this issue. Certainly it'll be no surprise that we'll be using CAS (Compare and Swap), but unlike our implementation of a lock-free stack and queue, we're just going to have to add some more fields to the Node<K,T> class.