Lock-Free Linked List (part 2)

published: Tue, 1-Aug-2006 | updated: Tue, 1-Aug-2006

[<< Part 1]

Last time in this series of articles on lock-free linked lists we pointed out some of the issues we needed to solve in order to have any chance of implementing such a data structure. These were to do with deleting nodes from the list: the first being that deleted nodes could not be disposed or reused since other threads may still be using them, and the other was the problem of inserting a node after a node that was just about to be deleted. I ended with the subtle hint that we would have to add some extra fields to our node class.

One of the first attempts at solving this second deletion problem was devised by T.L.Harris. He essentially split the deletion process into two steps: a logical step and a physical step. The logical deletion step marked the node as deleted, whereas the physical deletion step reformed the link that pointed to this node to point to its successor node instead.

The way he implemented this algorithm was clever. In 32-bit architectures, pointers point to absolute addresses in virtual memory. Not only that, but addresses are aligned on 32-bit boundaries (machine word boundaries). Some architectures enforce this, others, like the Intel CPUs, don't, but it's assumed anyway. Addresses that are on machine word boundaries have their least significant two bits clear. Harris used one of these two bits as a flag to indicate logical deletion.

Let's see how this works. Imagine that we have two threads, one of which wants to delete node X, the other that wants to insert a new node after node X.

The deletion thread will read the Next pointer from X. If the deletion bit is set, there's nothing to do: someone else is deleting the node. If it isn't set, the thread will CAS the Next pointer back, but with the deletion bit set. At this point the node is logically deleted. Now the deletion thread can CAS the predecessor node's Next pointer to jump over the logically deleted node and physically delete it.

Let's now look at the second thread's flow. First it creates the new node. Now it enters a standard-looking CAS loop. In the loop, it first reads X.Next. If the deletion bit is not set it sets the new node's Next pointer to X.Next, and uses CAS to set X.Next to its new node. This is repeated until the CAS succeeds, or it finds X.Next has its deletion bit set.

If the deletion bit is found set however, the insertion process knows that it no longer has anywhere to attach the new node to, so it starts over again from the beginning of the list to find the new spot to insert the new node (it's a single linked list, remember, so it can't go backwards).

There are three main points to note with this algorithm. First: once the deletion bit is set, it is never cleared. Second, code that is following the next links must remember to mask off the deletion bit before dereferencing the pointer.

The third point is, of course, it just won't work in C#.

Yes, that's right, in C# we don't have pointers unless we use unsafe mode. And even then we couldn't implement this algorithm as designed unless the whole linked list and its nodes were written in unsafe code. I don't want to go there yet -- I haven't written any unsafe C# in the past four years and I'm certainly not going to start now.

So how do we get around this problem? Our CAS operation only works with 32-bit entities and a

reference is already 32 bits. We have to CAS the new Next value no matter what, since we could find two or more threads trying to insert a new node at the same point and it would be bad to have a race condition where we could lose a node or more.

(To see this, imagine that thread X and thread Y were both about to update the predecessor's Next value to their particular node. If they didn't CAS the value one thread could update the Next value without realizing that the other thread had already updated it. The node that was inserted last would "win" and be present in the list, the earlier insertion would be lost.)

Similarly we can't just check the logical deletion flag and, if clear, quickly CAS the new value of Next to point to our new node:

```
if (!parent.Deleted)
  CAS(parent.Next, parent.Next, node);
```

This has a bad race condition too: a thread that's deleting the parent node could set the flag after we look at it, but before we manage the CAS.

One answer is to have a flag that states "this node is being updated, please wait a while":

```
while (!CAS(parent.BeingUpdated, false, true))
  /*do nothing*/ ;
if (!parent.Deleted)
  parent.Next = node;
parent.BeingUpdated = false;
```

This technique is pretty widespread: it's known as a spinlock. In essence, you can view it as the lock-free version of a standard lock. It uses a single flag to mimic a mutex or critical section, and uses CAS in a loop to set the flag true (the code "spins" doing nothing if the CAS fails). Eventually the flag will get set true and at that point this code has control of the object being protected by the flag. The code doesn't have to use CAS from now on, even when it sets the flag false again (C# and .NET guarantee that the setting of the new value is an atomic operation).

Providing that the deletion code also uses the spinlock we are home free. Except, I don't particularly like the solution. We have to use a try..finally block otherwise we could find

ourselves with a set flag and no way to turn it off. We also have got this far without using spinlocks so why should we turn to them now.

So next time, we'll solve the issue in a different way.