Trying to write a lock-free singly linked list, trouble with the removal

I'm trying to write a lock free singly linked list. Eventual consistency is not a problem (someone traversing a list which might contain incorrect items).

I think that I got the add item right (looping and `Interlocked.CompareExchange`).

But I can't figure out how to remove nodes (anywhere in the list) since I have to get the previous item and them set its `Next` field to the current nodes `Next` field.

```
class Node
{
    Node Next;
    object Value;
}

class SinglyLinkedList
{
    Root _root;

    public void Add(object value)
    {}

    public void Remove(object value)
    {}
}
```

i.e.

a -> b -> c

to

a -> c

Pseudo code:

```
Node prev;
Node node = _root;
while (node.Value != nodeValue)
{
  prev = node;
  node = node.Next;
}
prev.Next = node.Next;
```

how do I make this an atomic operation (i.e. make sure that `prev.Next = node.Next` is invoked without either next or prev being removed in between by another thread)?

I could use `ReaderWriterLockSlim` instead, but I found this problem interesting as I know that there exists lock free linked lists.

My concerns:

The following can happen if the current thread is suspended between the loop and the assignment:

- `prev` itself might have been removed by another thread.
- `node` might have been removed by another thread.
- `Node.Next` might have been removed

c# multithreading lock-free

edited May 31 '13 at 18:09

asked May 31 '13 at 17:52

jgauffin
55.1k 18 108 216

2 Have you looked at Interlocked.CompareExchange (msdn.microsoft.com/en-us/library/...) ? You could update prev with node.next if prev still has it's original value. – Jeff Paquette May 31 '13 at 18:00

@JeffPaquette no. both `node.Next` and `prev.Next` might have been changed either by `Add()` or another `Remove()` operation. So comparing only `prev` wont help. – jgauffin May 31 '13 at 18:02

If you need a lock free doubly linked list, I could recommend you my pure C# implementation based on a paper by Sundell and Tsigas. github.com/2i/LockFreeDoublyLinkedList – ominug Jan 27 at 16:16

1 Answer

Yep, adding is a simple two-steps loop with CAS on the previous .Next pointer; but removing is hard!

Actually you can't do it without using an additional piece of information and logic.

Harris created the solution to this problem by adding a marker bit that once set disallows anyone to modify the node. The removal becomes two-steps: first (CAS) mark the removed node to prevent anyone from changing it (especially its .Next pointer); second CAS the previous node .Next pointer, which is now safe because the other node has been marked.

The problem is now that in C Harris used the two least significant bits of the .Next pointer as marker. This is clever because with 4-bytes aligned pointers they are always unused (i.e. 00) and because they fit in the 32 bits pointer they can be CAS atomically with the pointer itself, which is key to this algorithm. Of course this is impossible in C#, at least without using unsafe code.

The solution gets a little more complicated and involves an additional reference to an immutable class containg the two fields (.Next + marker).

Instead of going into a lengthy explanation of those ideas, I have found some references on the internet that will go into all the details that you need, see:

Lock-free Linked List (Part 1) Explains the problem;

Lock-free Linked List (Part 2) Explains the C solution + the spinlock solution;

Lock-free Linked List (Part 3) Explains the immutable state reference;

If you are really curious about the topic there are academic papers with various solutions and analysis of their performance, e.g. this one: Lock-free linked lists and skip lists

answered Jun 3 '13 at 17:07

jods
3,125 3 12