Lock Free Linked List using Compare & Swap

Final Project
Distributed Algorithms and Systems

Lawrence Bush
Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York
April 30, 2002

Abstract

For my project, I implemented a Lock-Free Linked List. This entailed the creation of new data structures and algorithms, the simulation of the compare&swap synchronization primitive and writing a multi-threaded simulation and test program. The data structures are templated so that they can store any data type. I started with the method that John Valois explains in his paper "Lock-Free Linked Lists Using Compare&Swap" and in his Thesis "Lock-Free Data Structures" and modified it so that it could be implemented in C++.

1. Problem Description

This project addresses the issue of concurrent access to shared data. This is important to applications in parallel algorithms, distributed computing, user-level thread implementation and multiprocessor operating systems.

When multiple processes concurrently access shared data the most important issues are data integrity and performance. Data integrity can be maintained using standard mutual exclusion methods, however, this comes with a performance cost.

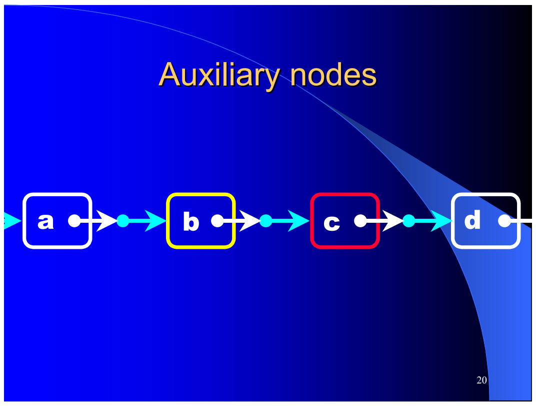
An Abstract Data Type (ADT), which allows concurrent operations by different processors without using mutual exclusion while ensuring data integrity, is presented in John Valois' paper "Lock-Free Linked Lists Using Compare-and-Swap."[1] The ADT presented in his paper differs from previous "Universal" methods because his ADT directly manipulates the data structure to improve performance.

The Lock-Free Linked List presented in the paper allows concurrent traversal, insertion and deletion operations by different processes without corrupting the data structure.

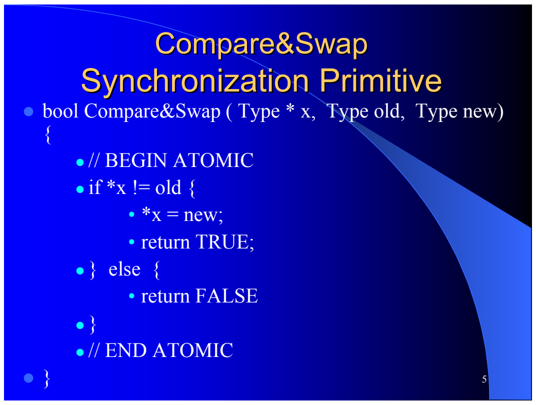
This is analogous to road construction. A system must be used to insert and delete pieces of road while cars are still going down the road.

This is accomplished by using auxiliary nodes, the Compare&Swap primitive to swing pointers and careful manipulation and checking of the data structure operations.

Auxiliary nodes are nodes that contain a next pointer but no data. These nodes are inserted between the real nodes in a linked list (shown below as the blue arrows).



Compare&Swap (shown below) is a synchronization primitive that atomically compares and updates a value.


Compare&Swap
Synchronization Primitive

```
• bool Compare&Swap ( Type * x, Type old, Type new)
  {
     • // BEGIN ATOMIC
     • if *x != old {
          • *x = new;
          • return TRUE;
     • } else {
          • return FALSE
     • }
     • // END ATOMIC
• }
```
5

The use of a synchronization primitive implies that the ADT uses mutual exclusion. Technically, it does. However, the ADT makes only limited use of hardware level atomic operations to swing pointers in the insert and delete operations.

The objective of this project is to implement these ideas.

2. Related Work

The objective of Lock-Free data structures is to avoid performance delays while objects are in the critical section. Lock-Free data structures are called wait-free. They guarantee a particular level of performance even if the concurrent objects halt. Ordinary synchronization primitives use mutual exclusion. There are basically 2 types of mutual exclusion: blocking and busy waiting. They are both marred with difficulties.

In blocking, convoying and deadlock are two potential problems. Priority inversion is a problem with busy waiting.

Convoying means that one slow or delayed process in the critical region affects all the other processes waiting for it.

The problem of deadlock occurs when two (or more) processes are waiting for a resource that the other is using.

If busy-waiting mutual exclusion is being used then the following situation known as priority inversion can result. Suppose that a CPU scheduler is using a priority scheduling method where the high priority processes always take precedence over the low priority processes. Then, a high priority process using the CPU can be waiting for a resource held by a low priority process. The result is that the low priority process will never get to use the CPU and the high priority process will never get to use the resource.

Lamport discovered that these problems can be avoided using Lock-Free methods. He spent twenty-seven years considering the benefits of avoiding mutual exclusion. Lamport created the first Lock-Free algorithm for the single-writer/ multiple reader shared variable. Lamport's achievements spurred much more research and, consequently, improvements in the field of Lock-Free methods.

Massalin and Pu coined the term Lock-Free. They wrote a multi processor Operating System kernel using Lock-Free data structures. Lock-Free is an alternative to mutual exclusion. It does not require exclusive access. Lock-Free data structures implement concurrent objects without the use of mutual exclusion. This method makes actions appear atomic. Conflicting operations do not corrupt the data structure. Valois' method allows simultaneous traversal, insertion and deletion.

Herlihy did research on universal synchronization primitives. Compare&Swap is a universal primitive. A universal primitive is one that solves the consensus problem.

Herlihy showed that a universal primitive is necessary and sufficient to implement Lock-Free ADTs.

An algorithm that provides Lock-Free functionality for any generic ADT is also called universal. Universal means "for any." Doing this requires a powerful synchronization primitive. In other words, a primitive that is powerful enough to solve this problem is subsequently called universal. It just so happens that this problem is analogous to the consensus problem and, therefore, if it can solve the consensus problem, it can do Lock-Free data structures.

There are currently universal (for any ADT) wait free methods but they have too much overhead to be efficient. This paper shows a direct implementation that is more efficient. Valois' uses a single word version of Compare&Swap which is commonly available on most systems.

3. Solution Description

For this project, I implemented a Lock-Free Linked List, a Test class and related algorithms. The Lock-Free Linked List is a shared abstract data type (ADT) that allows operations by different processors to occur at the same time.

For this implementation I created the following classes:

List	**Class**
Node	**Class**
Iterator	**Class**
Test	**Class**
Lock	**Class**
CriticalSection	**Class**

Together, the List, Node and Iterator Classes provide the following functionality:

Traverse
Insert
Delete

Synchronization is provided by simulating the Compare&Swap primitive using the Lock and CriticalSection classes. Each node has its own distinct lock to synchronize the swinging of its next pointer.

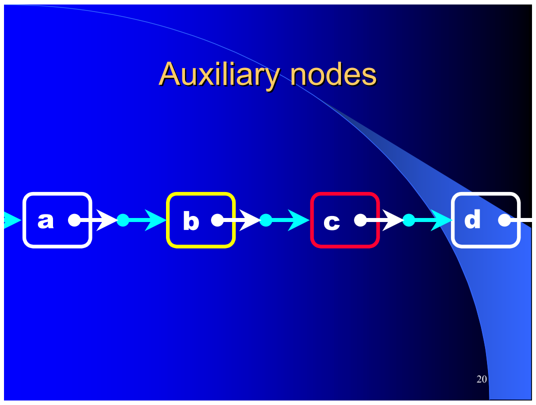
The Test class uses multiple threads to simulate distributed concurrent operations on the list.

Attachment 4 contains a complete description of each program file.

The 2 most fundamental aspects of this implementation are pointer swinging and the use of auxiliary nodes.

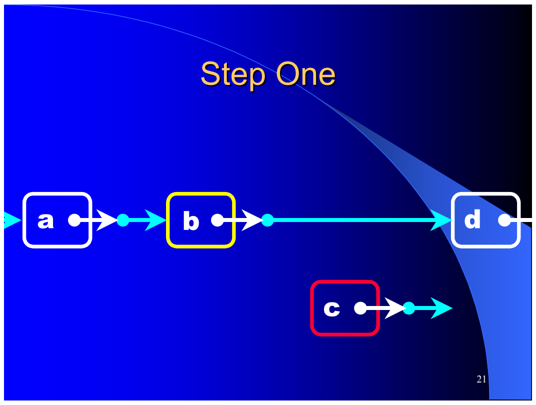
Pointer swinging entails reading a pointer and then using the compare and swap primitive to atomically recheck and change the pointer. Pointer swinging resolves contention when conflicting operations occur. The result is that one of the operations fails. An example of this is if two processes attempt to delete the same node.

An auxiliary node is a node with only a next pointer (no data). We insert an auxiliary node in between each cell in the list (shown below as the blue arrows). This allows adjacent operations to take place without interfering with each other.

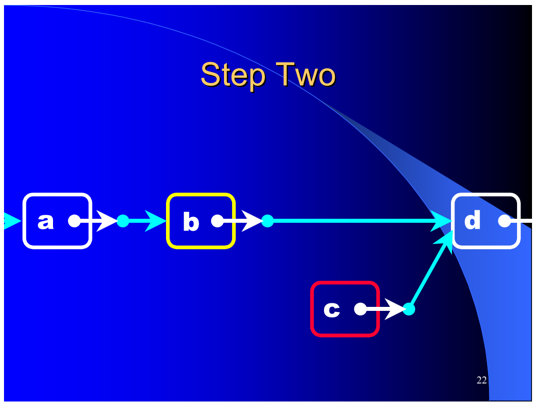


Consider the following example where an insert and delete simultaneously occur on adjacent nodes. We are going to delete Node b while inserting Node c.

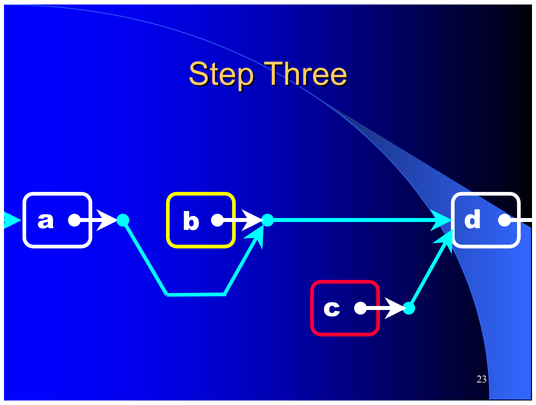
Step One: Create and connect Node c and an accompanying auxiliary node.



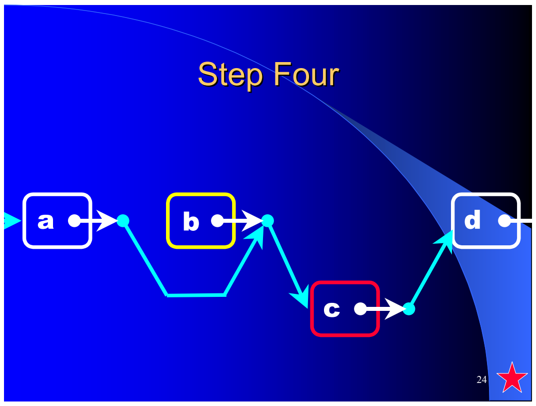
Step Two: c->next->next = d



Step Three:
a->next->compare_and_swap_next(b, b->next)



Step Four: auxiliary node next = node c
b->next->compare_and_swap_next(d, c)



We can see from above that when an insert is performed a forward path remains for any traversing process. If a deletion is performed the deleted node maintains a forward path for any traversing process. In a garbage collection system, that deleted node will continue to exist until there are no more pointers to it. However, in my implementation, I use the assumption regarding Memory Management stated in Valois' paper on page 7:

> *"We have thus far assumed that new cells could be allocated whenever necessary, and that deleted cells could be left intact for cursors to continue traversing them."*

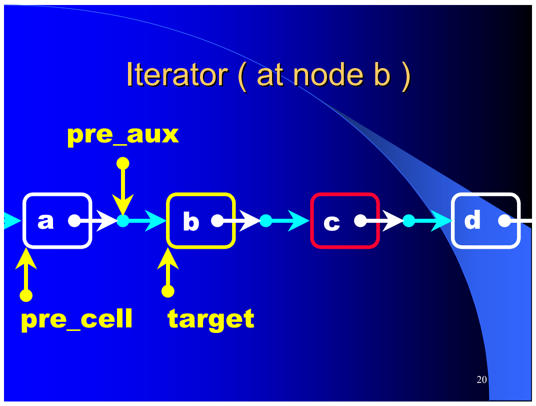
Note that the above concurrency depiction only establishes the basic logical construct for the operations and not the entire implementation.

Iterator

In the insert/delete example above, it is not clear how the algorithm knows where the various pointers are. This is accomplished through the iterator object. The iterator in my implementation is analogous to the cursor in Valois' paper. It is like a pointer on steroids. It is used to indicate where to insert and delete. An iterator contains the following 3 node pointers:

target
pre_cell
pre_aux

These are shown in the picture below.



It also contains its own functions. Its most fundamental operation is the forward traversal (++). This is implemented through operator overloading as shown in the following code:

```
iterator operator++(int n) {
          iterator temp;
          temp = *this;
          go_next();
          return temp;
          }

bool go_next() {

   if(target->is_last_node()) { return false; }
         pre_cell = saferead(target);
         pre_aux = saferead(target->next);
         update_iterator();
         return true;
   }
```

```
void update_iterator() {

        if ( pre_aux->next == target ) { return; }
        list_node * p = pre_aux;
        list_node * n = p->next;

        while (  ( n->is_not_lastnode() )
                 && n->is_aux_cell()  )
        {
        pre_cell->
        compare_and_swap_next( p, n );

         p = n;
         n = saferead(p->next);

        } // end while

         pre_aux = p;
         target = n;

} // end iterator update
```

The first function (update_iterator) is overloaded. It is a postfix iterator, therefore it prepares to return the previous value of the iterator. It then calls go_next. The go_next function moves the pre_cell pointer forward to the target pointer and moves the pre_aux forward to the next cell. It then calls update_iterator which is where the target finally gets set. The final destination target is dependent on the location of pre_aux. If pre_aux already points to target, then the iterator does not need updating and the function returns. However, while iterating forward, it will need updating. Therefore the function moves two temporary pointers (p and q) progressively forward until the n pointer is pointing to a normal cell (with data) rather than an aux_cell. This will be the next normal cell after pre_cell. Then it sets the pre_aux and target and returns.

Note that if the update function encounters a string of aux_nodes along the way, it will remove the extra ones using the compare and swap function. Under high contention, extra aux_nodes can occur. This is by design. However, as page 3 of Valois' paper indicates, chains of auxiliary nodes are permitted in his algorithm, although they are undesirable for performance reasons. Therefore, any passing iterator removes them.

The above C++ code does not show the full detail of every function called during the iteration procedure; however, it does show the basic logic behind the operation.

4. Solution Analysis

To test this ADT I created a separate class which runs a program test sequence. The test sequence was written to be exhaustive (many inserts, deletes and traversals) and to create a lot of contention. The test program runs 40 concurrent threads. Each thread makes about 1,000 insertions and 400 deletions for a total of about 40,000 insertions and about 16,000 deletions. The function "TestFunctionG" (in the attached file test.cpp) performs this thread specific testing sequence. The test sequence is as follows:

Each thread makes 500 insertions. It does this 10 nodes at a time, and then moves the iterator back to the beginning of the list.

Each thread then performs the following sequence 100 times.

1. Move the iterator to the begining of the list.
2. Iterate to the 25th cell (note iterating skips auxiliary cells).
3. Delete 2 nodes.
4. Iterate 2 nodes forward.
5. Insert 3 cells.
6. Move the iterator to the beginning of the list.
7. Iterate to the 25th real node.
8. Insert 3 nodes.
9. Iterate forward 2 nodes.
10. Delete 2 nodes.

Each of the 40 threads operate concurrently. The general idea is that the deletions would cause the inserting iterators to "fall" back to its position. This would then make them perform operations on the same cell or on directly adjacent cells, creating contention.

The test program tracks various statistics to verify the results.

The contention created by the numerous insertions and deletions causes some of the insertions and deletions to fail. This is the intended behavior of the ADT.

When an insertion or deletion fails, it returns the value of false. When an insertion or deletion is successful, it returns the value of true.

Each thread keeps track of how many insertions and deletions it makes. It also keeps track of how many of these fail. Each thread then calculates a net number of additions to the list (i.e. successful insertions – successful deletions). Each thread then adds these figures to the (net) total number added to the test data structure (using a synchronization object to prevent race conditions on the value). That number is shown below as the "Sum of threads net additions to the list."

The number of successful and unsuccessful insertions and deletions for each individual thread is also shown in the **"Sample Output" Attachment 3**. One interesting point that the data reveals is that the thread that finishes first typically has fewer failed insertions and deletions than the other threads. This is because it had more time in the insertion area by itself (or without as many other concurrently operating threads). In other words, it experiences less contention. The same is true for the threads that finish last. Note that the threads are run in order (1 – 40) but they do not necessarily finish in that order. This really depends on how much CPU time each thread is given.

The list data structure also tracks these additions and deletions. A synchronization object also protects the changes to this value. That number is shown below as the "List internal add/delete counter: ListSize."

After all of the threads have quit, the program runs an integrity test on the list. This is run in non-concurrent mode. It adds up all the normal and auxiliary nodes in the list and reports the figures ("List internal add/delete counter: ListSize" and "total_aux_cells").

You can see from the data below that all 3 measurements indicate the same number of normal cells in the list. This shows that the list functions are correctly executing the insertion and deletion requests. It also means that the ADT correctly reports to the threads when these operations fail. In all the tests I performed, these numbers always matched.

You can see from the data in **Attachment 3** that there were numerous insertion failures. This is normally the case. It is a small percentage of the total attempts but with 40,000 inserts, the small percentage is a significant number (about 600 in total).

There are only 4 failed deletions in this run. There are usually 0 –5 for this particular test. The test attempts to create as much contention as possible by having 40 concurrent threads all inserting and deleting in the same area of the list (approximately from node 20 – 40). However, causing deletions to fail requires more contention than causing insertions to fail.

The data in the **Attachment 3** shows a list of the first 1,000 nodes in the final list. The data below also shows the total number of real nodes and auxiliary nodes. There is 1 more auxiliary node than normal cell in the final list. This is perfect because we need at least one more auxiliary node so that there is one before and after each cell.

The algorithm does not guarantee that there will be just 1 more auxiliary node than real nodes. However, this is usually the case. Sometimes there are a few more than needed. The algorithm attempts to remove them all, but depending on the type and amount of contention, it can leave some extras. They will be cleaned up later, but at any given moment, there may be some extra auxiliary nodes in the list. This is consistent with the intended operation of the ADT.

Attachment 3 includes a report on the contents and type of the first 1,000 cells so that you can see that this is so. The list also contains 1 basenode and 1 lastnode (the last node is not shown in the report).

DATA

```
Report from the treads:
Sum of threads net additions to the list  =  25485

Integrity test :

          total_normal_cells.         =      25485
          total_aux_cells             =      25486

List internal add/delete counter:

          ListSize                    =     25485
```

5. Conclusion

In conclusion, the ADT performed exactly as expected.

Note: The test discussed above was run on a 600 MHz AMD Compaq Presario running Windows 98. When the same test is run on a Windows 2000 XP machine with a faster processor, the results are different. The difference is that on the XP machine there are no insert or delete failures. This is most likely due to the faster processor. It could also be due to the way user space threads are allocated CPU time, particularly when another thread is blocking.

Running additional threads in combination with a different and more aggressive insertion, traversal, deletion pattern, would most likely create, additional contention and subsequent insertion/deletion failures.

Please refer to Attachment 5, files test.cpp and test.h for the complete test sequence code.

References

1. J.D. Valois. "Lock-Free Linked Lists Using Compare-and Swap." In Proceedings of the Fouteenth Symposium on Principles of Distributed Comuting (1995)

2. J.H. Anderson. "Lamport on Mutual Exclusion: 27 Years of Planting Seeds" (2001)

3. H. Massalin and C. Pu. "A Lock-Free Multiprocessor OS Kernel." Technical report CUCS-005-91.

4. M. Herlihy. "Impossibility and Universality Results for Wait-Free synchronization." In Proceedings of the Seventh Symposium on Principles of Distributed Computing.

Attachments:

1. Instructions

2. Test Sequence

3. Sample Output

4. Program File Descriptions

5. Code Printouts:

    - main.cpp
    - test.h
    - test.cpp
    - lockfreelist.h
    - lock.h
    - lock.cpp
    - criticalsection.h
    - criticalsection.cpp

6. Main Reference Paper: J. Valois. "Lock-Free Linked Lists Using Compare and Swap."

7. Project Proposal