Generic Mechanism for Transforming a Single-Producer / Single-Consumer Unbounded Linked List into a Bounded Linked List without Allocation

Frank Afriat
_Hewlett Packard Enterprise_

Follow this and additional works at: [http://www.tdcommons.org/dpubs_series](http://www.tdcommons.org/dpubs_series)

Recommended Citation
Afriat, Frank, "Generic Mechanism for Transforming a Single-Producer / Single-Consumer Unbounded Linked List into a Bounded Linked List without Allocation", Technical Disclosure Commons, (October 20, 2016)
Generic Mechanism for Transforming a Single-Producer / Single-Consumer Unbounded Linked List into a Bounded Linked List without Allocation

Abstract:

SPSC unbounded linked list are very easy to construct and may have interesting properties like Wait-freedom or Lock Freedom but generally provided implementation are unbounded and require Memory Allocation for adding nodes. The purpose of this document is to present an idea giving the possibility to keep all the interesting properties of the Linked List but to transform it into and Bounded Linked List without memory allocation.

The Idea:

The essence of a linked list is to have Nodes holding a Value and pointing to the next Node. The Linked List can be defined by the Head Node pointing to the next Node pointing again to the next till the last Node pointing to nothing, called the Tail Node.

Removing a value from the list is consisting on removing the Head Node and defining the next Node as the New Head.

Adding a new value to the list is consisting on adding a new Node holding the value attached to the Tail and to define the New Node as the new Tail.

Adding a new value is then requiring adding a new Node. To solve this problem a general idea is to create a Pool of Nodes that will be used to provide new Nodes but we have then the need of
a structure that is providing a Pool of Nodes that we can Add and Remove at the same time from the Producer Thread and the Consumer Thread:

The first idea would be to use the exact Linked List we are trying to transform. But again, the Linked List used for the Pool of Nodes will require new Nodes for holding as value the Nodes... so it is not working as a solution for removing the need of allocation.

The solution:

The solution for this problem is to not use directly the same Linked List for holding the Nodes as values but to define a new Linked List specialized only on holding values of type Nodes.

The idea is to use the value (as a Node) we want to keep in the pool but instead of requiring a new Node for holding this value, we can directly use the node itself for that!
What is crucial in the Pool of Node is the fact that the State of the Node can be changed. It is important to not forget to clean the Node before storing it in the Pool and to clean it after removing it from the Pool.

The Pool of Node itself is an Unbounded Linked List of Nodes pre-allocated with a given number of nodes.

Example of use (in Java):

Frank Afriat is the author of a Special Linked List (SPSC, unbounded) described in his blog: http://javadesignperformance.blogspot.co.il/2012/08/lock-free-wait-free-fifo-queue.html and in the following posts.

The special Linked List is a standard Linked List who’s first Node can be marked as Removed eventually when there is only one Node left on the Linked List.

Building the Pool of Nodes is easy:

The idea is to keep always one Node when the Linked List is Empty which is again only possible because the Linked List is used as a Pool of Nodes.

You can verify that the method addToTail() is not using any allocation but using the node passed as parameter.

```java
public boolean addToTail(LinkedNode<T> newNode) {
    tail.setNext(newNode);
    tail = newNode;
    return true;
}
```

```java
public LinkedNode<T> removeFromHead() {
    // Next not removed
    LinkedNode<T> nodeToRemove = head.getNext();

    LinkedNode<T> nextNode = nodeToRemove.getNext();
    // Case one last node: not removing, assuming empty
    if (nextNode == null) {
        return null;
    } else {
        // Real removing
        head.setNext(nextNode);
        nodeToRemove.setNext(null); // Need to clear the next value
        nodeToRemove.setValue(null);
        return nodeToRemove;
    }
}
```

Then the original Linked List must be modified to use the pool:
When adding a Node, the node is taken from the Pool of Nodes and when removing a Node, the Node is returned to the Pool of Nodes.

```java
public boolean addToTail(T value) {
    ListNode<T> newNode = this.pool.removeFromHead();
    if (newNode == null) {
        return false;
    }
    newNode.setValue(value);
    tail.setNext(newNode);
    tail = newNode;
    return true;
}
```

```java
public T removeFromHead() {
    //Next not removed
    ListNode<T> nodeToRemove = head.getNext();
    if (firstNodeMarkedAsRemoved) {
        ListNode<T> nextNode = nodeToRemove.getNext();
        //Case empty
        if (nextNode == null) {
            return null;
        }
        //At least one node
        //Removing effectively the first node is possible
        //real remove of previous node
        head.setNext(nextNode);
        firstNodeMarkedAsRemoved = false;
        nodeToRemove.setNext(null); //reset next value
        pool.addToTail(nodeToRemove);
        nodeToRemove = nextNode;
    }

    T value = nodeToRemove.getValue();
    nodeToRemove.setValue(null);

    //Case one last node
    if (nodeToRemove.getNext() == null) {
        firstNodeMarkedAsRemoved = true;
    } else {
        //real removing
        head.setNext(nodeToRemove.getNext());
        nodeToRemove.setNext(null); //reset next value
        pool.addToTail(nodeToRemove);
    }
    return value;
}
```

I used the same Performance Test used inside the post
http://javadesignperformance.blogspot.co.il/2016/09/testing-any-spsc-queue-with-

http://www.tdcommons.org/dpubs_series/304
to compare the performance of the bounded Linked Lists against the unbounded versions:

<table>
<thead>
<tr>
<th>Queue</th>
<th>SPSC: 2 Threads</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ops/sec</td>
<td>latency (ns)</td>
</tr>
<tr>
<td>SpecialLinkedListSupplier:SingleLinkedListNode</td>
<td>6,851,065</td>
<td>145</td>
</tr>
<tr>
<td>SpecialLinkedListSupplier:SingleLinkedListNodeVolatile</td>
<td>4,348,465</td>
<td>229</td>
</tr>
<tr>
<td>SpecialLinkedListSupplierSynchronized:SingleLinkedListNode</td>
<td>5,333,343</td>
<td>188</td>
</tr>
<tr>
<td>SpecialLinkedListSupplier:SingleLinkedListNodeAtomicLazy</td>
<td>3,965,476</td>
<td>252</td>
</tr>
<tr>
<td>ConcurrentLinkedQueue</td>
<td>7,728,539</td>
<td>129</td>
</tr>
<tr>
<td>ArrayBlockingQueue</td>
<td>6,485,962</td>
<td>154</td>
</tr>
<tr>
<td>SpecialLinkedListSupplierBounded:SingleLinkedListNode</td>
<td>15,823,631</td>
<td>62</td>
</tr>
<tr>
<td>SpecialLinkedListSupplierBounded:SingleLinkedListNodeVolatile</td>
<td>10,311,041</td>
<td>98</td>
</tr>
<tr>
<td>SpecialLinkedListSupplierSynchronizedBounded:SingleLinkedListNode</td>
<td>7,014,601</td>
<td>144</td>
</tr>
</tbody>
</table>

The Bounded Linked List SpecialLinkedListSupplierBounded:SingleLinkedListNodeVolatile is offering a throughput much better than the Unbounded version but also much better than the ArrayBlockingQueue provided inside the JDK.

One of the reason is the fact that the Bounded Version is no more putting any pressure on the Garbage Collector because there is no more allocation.

One other possible reason is the natural balance created between the add() and the remove() because of the use of the Pool of Node using the same Linked List but with the opposite operation, that may improve the global throughput.

**Conclusion:**

This document has presented an idea that can be used to transform an unbounded Linked List to a bounded version. An example has also been provided in Java that demonstrates that transforming the unbounded Linked List to a bounded one may improve a lot the Throughput of the initial Linked List by reducing the pressure on the Memory.

Disclosed by Frank Afriat, Hewlett Packard Enterprise