Concurrent Linked Lists Using Multi-Resource Locks

Danny Cosentino, Peter Dorsaneo, Pauline Johnson

Computer Science, University of Central Florida
Orlando, FL 32816, USA

{danny.cosentino12,dorsaneo,pauline.johnson,}@knights.ucf.edu

1 An Introduction to Linked Lists

A linked list is a linear data structure made up of node elements, linked together by pointers. Each node contains data and a reference to the next node in line. Its key operations are INSERT, FIND, and DELETE. INSERT inserts a node with given data into the list. FIND returns whether a given data value exists in the list. DELETE deletes the node with the given data.

2 Incorporating MRLOCK

Our multithreaded implementation of a linked list uses a multi-resource lock or MRLOCK[1]. Our linked list only supports the use of a head pointer, so average case runtimes are O(n) for INSERT, FIND, and DELETE methods. Each node in the list has a key, which is used to turn on its bit in the thread assigned `bitset` (used by the MRLOCK to lock it). Since our code does not allow for duplicate values to be in the list at a given time nor does it allow for negative values to be in the list, we are able to utilize the value stored in the node as the nodes key as well. Each node in the list also maintains a boolean mark variable for logical deletion of the node from a list. The key algorithms of our implementation are INSERT, DELETE, and FIND.

Algorithm 1. Linked List with MRLOCK

```
1  struct Node
2  {
3    uint32_t val;
4    uint32_t key;
5    bool marked;
6    Node* next;
7
8    Node(const uint32_t& val) :
9      val(val), key(val), marked(false), next(nullptr)
10     {}
11 };
12
13
14 class MRLazyLockList
15 {
16   private:
17     const static int kNodeBankSize = 1e7;
18     const static int kNumThreads = 4;
19     const static int kInitListSize = 1e4;
20
21     mrlock l;
22
23     Node* head;
24
25     vector<Node*> node_bank[kNumThreads];
```

```
26      array<bitset<NUM_BITS>, kNumThreads> bits;
27
28      bool Validate(Node* pred, Node* curr);
29      void PreBuildList(void);
30      void BuildNodeBank(void);
31
32    public:
33      MRLazyLockList(void);
34
35      bool Insert(uint32_t value, uint8_t thread_id);
36      bool Find(uint32_t value);
37      bool Delete(uint32_t value, uint8_t thread_id);
38
39      void Print(void);
40  };
```

2.1 Insertion

Algorithm 2 shows our implementation of the insert function using an MRLOCK. Insertion is done such that the nodes are sortedly placed in the list. We utilize the Lazy Synchronization locking technique[2] to traverse the list while ignoring locks and only locking the `pred` and `curr` nodes of the list once we have reached them. We utilize a thread local class `bitset` object and a global MRLock class instance to attempt locking, marking the bit in the `bitset` that corresponds to a nodes key. Our VALIDATION method does not require us to traverse the entire list, rather it only ensures that the `pred` node's `next` field still points to the `curr` node and neither of the nodes have been marked as *logically* deleted before we attempt to lock them. From here, if we are able to insert the value into the list, we pull a node from our local thread's `node_bank` and assign the `value`, `key`, and `mark` to the node as appropriate. Utilizing a `node_bank` for each thread allows us to avoid the overhead cost of memory allocation for inserting a new node into the list when running the threads. We then proceed to insert the node between the `pred` and `curr` nodes of the list. Otherwise, if the insertion fails at the validation step, we will release any locks on the nodes and the loop will iterate through the list again to attempt an insertion.

Algorithm 2. Insert

```
1  bool MRLazyLockList::Insert(uint32_t value, uint8_t thread_id)
2  {
3    uint32_t key = value;
4
5    while (true)
6    {
7      Node* pred = head;
```

```
8       Node* curr = head->next;
9
10      while (curr && curr->key < key)
11      {
12        pred = curr;
13        curr = curr->next;
14      }
15
16      bits[thread_id].set(pred->key);
17
18      if (curr)
19      {
20        bits[thread_id].set(curr->key);
21      }
22
23      uint32_t handle = lock(ref(l), bits[thread_id]);
24      if (Validate(pred, curr))
25      {
26          // Handling when pred is the last element of the list and curr
27          // is nullptr.
28        if (!curr)
29        {
30          pred->next = node_bank[thread_id].back();
31          node_bank[thread_id].pop_back();
32
33          pred->next->val = value;
34
35          bits[thread_id].set(pred->key, 0);
36          unlock(ref(l), handle);
37          return true;
38        }
39        else if (curr->key == key)
40        {
41          bits[thread_id].set(pred->key, 0);
42          bits[thread_id].set(curr->key, 0);
43          unlock(ref(l), handle);
44          return false;
45        }
46        else
47        {
48          Node* n = node_bank[thread_id].back();
49
50          n->val = value;
51          n->key = value;
52          n->marked = false;
53
54          if (n->key == pred->key || n->key == curr->key)
55          {
56            bits[thread_id].set(pred->key, 0);
57            bits[thread_id].set(curr->key, 0);
58            unlock(ref(l), handle);
59            return false;
60          }
61
```

```
62          n->next = curr;
63          pred->next = n;
64
65          node_bank[thread_id].pop_back();
66          bits[thread_id].set(pred->key, 0);
67          bits[thread_id].set(curr->key, 0);
68          unlock(ref(l), handle);
69
70          return true;
71        }
72      }
73
74      bits[thread_id].set(pred->key, 0);
75      bits[thread_id].set(curr->key, 0);
76      unlock(ref(l), handle);
77    }
78  }
```

2.2 Validation

Our VALIDATION method does not require us to traverse the entire list, rather it only ensures that the `pred` node's `next` field still points to the `curr` node and neither of the nodes have been marked as logically deleted before we attempt to lock them.

Algorithm 3. Validate

```
1  bool MRLazyLockList::Validate(Node* pred, Node* curr)
2  {
3    return !pred->marked && !curr->marked && pred->next == curr;
4  }
```

2.3 Deletion

Our DELETE method is also lazy, as it only takes two steps to do so: first, mark the target node as being *logically* removed, and second, redirect the `pred` nodes `next` field to *physically* remove it. While our implementation traverses the list ignoring the locks only until it must lock the `pred` and `curr` nodes, but VALIDATION does not require retraversing the entire list to determine whether the nodes are still in the set prior to locking. Once the validation is complete and the node we are looking for is found to exist in the list, we utilize the local threads `bitset` to mark the corresponding `pred` and `curr` nodes in the `bitset`, then passing that into the MRLock for locking. We then remove the desired node, placing its reference into the local threads `node_bank` for reusability and avoiding the overhead of memory deletion in C++ while running the threads. Once completed we unmark the bits in the `bitset` and release the locks on the nodes.

Algorithm 3. Delete

```
 1   bool MRLazyLockList::Delete(uint32_t value, uint8_t thread_id)
 2   {
 3     uint32_t key = value;
 4     Node* pred, *curr;
 5
 6     while (true)
 7     {
 8       pred = head;
 9       curr = head->next;
10
11       while (curr && curr->key < key)
12       {
13         pred = curr;
14         curr = curr->next;
15       }
16
17       bits[thread_id].set(pred->key);
18
19       if (curr)
20       {
21         bits[thread_id].set(curr->key);
22       }
23
24       uint32_t handle = lock(ref(l), bits[thread_id]);
25       if (Validate(pred, curr))
26       {
27         // Handling for when we are at the tail of the list.
28         if (!curr)
29         {
30           bits[thread_id].set(pred->key, 0);
31           unlock(ref(l), handle);
```

```
32            return false;
33          }
34
35
36          else if (curr->key != key)
37          {
38            bits[thread_id].set(pred->key, 0);
39            bits[thread_id].set(curr->key, 0);
40
41            unlock(ref(l), handle);
42            return false;
43           }
44           else
45           {
46             curr->marked = true;
47             pred->next = curr->next;
48
49             node_bank[thread_id].push_back(curr);
50
51             bits[thread_id].set(pred->key, 0);
52             bits[thread_id].set(curr->key, 0);
53
54             unlock(ref(l), handle);
55             return true;
56           }
57         }
58
59        bits[thread_id].set(pred->key, 0);
60        bits[thread_id].set(curr->key, 0);
61        unlock(ref(l), handle);
62      }
63    }
```

2.4 Find

Our FIND method traverses the list once while ignoring locks. It returns true if the node it was searching for is in the list and is unmarked, and false otherwise. This satisfies the property of the method being wait-free.

Algorithm 4. Find

```
1  bool MRLazyLockList::Find(uint32_t value)
2  {
3    uint32_t key = value;
4    Node* curr = head;
5
6    while (curr && curr->key < key)
7    {
8      curr = curr->next;
9    }
10
11   return curr && curr->key == key && !curr->marked;
12 }
```

3 Experimental Results

To test our MRLazyLockList, we spawned 4 threads and randomly chose operations (Insert, Delete, Find) for them to complete. Below are our average execution times for the various trials.

We see a satisfactory performance utilizing the Lazy Synchronization locking technique with the MRLock.

Thread	Milliseconds
(1 Insert, 1 Delete, 2 Find)	60
(2 Insert, 2 Delete)	120
(2 Insert, 1 Delete, 1 Find)	77
(1 Insert, 2 Delete, 1 Find)	81
(4 Insert)	99
(3 Insert, 1 Delete)	110
(3 Insert, 1 Find)	78

4 References

1. MRLOCK: Deli Zhang, Brendan Lynch, Damian Dechev, Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors, In Proceedings of 17th International Conference on Principles of Distributed Systems (OPODIS 2013), Nice, France, December 2013.
2. Herlihy, M., & Shavit, N. (2008). The art of multiprocessor programming. Amsterdam: Elsevier/Morgan Kaufmann.
3. Deli Zhang and Damian Dechev. 2016. Lock-free Transactions without Rollbacks for Linked Data Structures. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16). Association for Computing Machinery, New York, NY, USA, 325–336.