CHAPTER 3
MSL AND LOCK BASED APPROACH OF CONCURRENCY

3.1 INTRODUCTION

Traditional data structures give few considerations to their execution in concurrent environments. It is not sufficient to simply move a traditional data structure into a concurrent environment and expect an improvement in performance by allocating additional resources and processing power. With the emergence of multiprocessing systems, there is a steady increase in the number of processors available on commercial multiprocessors. This increase in the availability of large computing platform has not been met by a matching improvement in our ability to construct new data structure. There is a requirement to shift the way we think and construct data structures. A data structure in a concurrent environment is access by multiple computing threads (or processes) on a computer. The proliferation of commercial shared-memory multiprocessor machines has brought about significant changes in the art of concurrent programming. Given current trends towards low cost chip multithreading (CMT), such machines are bound to become ever more widespread. Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. Designing concurrent data structures and ensuring their


correctness is a difficult task, significantly more challenging than doing so for their sequential counter parts. The difficult of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on. To manage the difficulty of concurrent programming, multithreaded applications need synchronization to ensure thread-safety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures. The traditional way to implement shared Access of data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. In concurrent search structures, locks are used to prevent concurrent threads from interfering with each other. A concurrency scheme must assure the integrity of the data structure, avoid deadlock and have a serializable schedule. Within those restrictions, we would like the algorithms to be as simple, efficient and concurrent as possible. In that direction we present an efficient and practical lock based MSL. It is a modified version of basic skip list; we describe methods for performing concurrent access and update on this MSL with locking approach. Experimental result shows that MSL structure is faster than original skip list structure for representation of dictionaries This is just the beginning to see how MSL behave in concurrent environment.

3.2 CONCURRENT OPERATIONS ON MSL

We now describe a method for performing concurrent operations on MSL. In MSL elements of the list are represented by a node. The left pointer of a node points to previous node and right pointer points to the
next node in the list and the nodes are kept in sorted order according their keys. The key of a node is given by $x \rightarrow \text{key}$, the value is given by $x \rightarrow \text{value}$ and the left pointer is given by $x \rightarrow \text{left}$. The head (H) and tail (T) of a list $l$ is treated as a node and is given by $l \rightarrow \text{H}$ and $l \rightarrow \text{T}$. For purposes of reasoning about our invariants, the H and T have the sentinel value (-infinity). Due to lock based concurrent access of MSL [Herlihy, Yossi and, Luchangco [27] ] some of the assumptions are done in presented approach.

• The term thread refer to a task or process operating concurrently with other threads.

• A thread obtains a lock on a field only when updating a field that other threads might be attempting to update.

• While searching for an element, no locks are needed.

• Only a single thread may hold a lock on a field, and by convention a thread only updates a field if it already holds a lock on the field.

• If a thread attempts to lock a field that is already locked, that thread is blocked until the lock can be obtained.

3.3 ALGORITHM

We present a lock based concurrent modified skip list algorithm supporting three methods, search_node, insert_node, delete_node. Figure 3.1 shows the fields of a node. Each node has a marked flag, which is used to make remove operations appear atomic. However, we may have to link the node in MSL, and thus might not be able to insert a node with a single atomic instruction, which could serve as the linearization point of a successful inser_node operation. Thus, for the concurrent access of MSL, we augment each node with an additional
flag, fullyLinked, which is set to true after a node has been linked in it; setting this flag is the linearization point of a successful insert_node operation in our MSL implementation. Before writing the algorithmic steps of the operations carried out, there is a pictorial representation i.e flowcharts are also given.

```
struct node
{
    int key
    struct node **left
    struct node **right
    struct node **down
    bool marked
    bool fullylinked
    lock lock
}
```

Figure: 3.1 structure of lock based concurrent MSL

- **search_node (key):** search for a node with key k equal to key, and return true if key found otherwise return false.

- **Insert_node (key, d):** inserts adds d to the set and returns true iff d was not already in the set.

- **Delete_node (v):** removes v from the set and returns true iff d was in the set.

The algorithms are written using the syntax and semantics of programming language C

**3.3.1 SEARCH_NODE**
Searching in concurrent access to MSL is accomplished by search_node procedure see figure 3.1, which takes a key v and search exactly like a searching in sequential linked list, starting at the highest level and proceeding to the next down level, if it encounters a node whose key is greater than or equal to v. The search process also save the predecessor and successor of a searched node v for further reference. search_node does not acquire any locks, nor does it retry in case of conflicting access with some other thread. We now consider each of the operation in turn.
Flow Chat (1) : Searching in lock Base MSL
3.3.2. INSERT_NODE OPERATION

The algorithm calls `search_node` to determine whether a node with the key is already in the list. If so, and the node is not marked, then the add operation returns false, indicating that the key is already in the set. However, if that node is not yet fully linked, then the thread waits until it is (because the key is not in the abstract set until the node is fully linked). If the node is marked, then some other thread is in the process of deleting that node, so the thread doing the add operation simply retries (lines 29-30).

If no node was found with the appropriate key, then the nodes returned by `search_node`, whose pointers are associated with the new node. The

```c
1. procedure search_node (int key):bool
2. {
3.     p=h
4.     while(p!=NULL)
5.         {
6.             While(p→value→key<key)
7.                 {
8.                     p=p→right
9.                 }
10.             if(p→value→key==key)
11.                 {
12.                     return true and break
13.                 }
14.             else
15.                 p=p→left→down
16.         }
17.     return false
18. }
```

Figure: 3.2 Searching in concurrent MSL
randomLevel function is used in the beginning of insert_node operation to determine the level at which new node to be. If the number generated is more than current level then a new level is created and insertion is done in (lines 44-74). The (lines 75-95) try to insert the new node at the level 1 and update the down pointer. If the generated level is less than current level, the new node is inserted according to the stored value of save[k]. A new node x is said to be inserted, if it is fully linked. If validation fails, the thread encountered a conflicting operation, so it releases the locks and retries. The algorithm for lock based insertion in MSL is given underneath in figure 3.2
Flow Chat (2-1) : Insertion in lock Base MSL
Flow Chat (2-2) : Insertion in lock Base MSL

1. Wait until it is fully linked.
2. Create a new node
   X set temp = current_level
3. If K > temp
   NO
   YES
4. Increment the no of level by 1
5. Save the necessary pointers values.
6. Locks the nodes that need to update at current level during insertion.
7. Valid = To check the validity of locked node.
8. Result
Flow Chat (2-3) : Insertion in lock Base MSL
Flow Chat (2-4) : Insertion in lock Base MSL

1.

Lock the nodes that needs to be updated during insertion.

Valid=To check the validity of locked nodes.

If (1 valid)

YES

Update the previous, Next of inserted nodes.

Update the down pointer of new node, k+1 and k-1 level node.

Fully link the new node x.

Unlock all the locked variables.

STOP
Procedure insert_node(key,d)
1. {
2. int max=50,temp,current_level
3. k=randomlevel ()
4. node * found_node=null,*x,*h1,*t1
5. node *save[max] ,*t ,*s ,*d ,*u
6. while(true) {
7. //search the msl for a key k of d and save the necessary pointers.
8. p=h
9. i=current_level
10. while (p#null) do
11. {
12. while(p→data→key<key) do
13. {
14. save[i]=p
15. p=p→right
16. }
17. If(p→data→key==key)
18. {
19. found_node=p
20. Write “searched node is found at p” and break
21. }
22. else
23. {
24. p=p→left→down
25. i=i-1
26. }
27. }
28. If (found_node != NULL)
29. {
30. if( ! found_node→marked)
31. {
32. }
Procedure insert_node(key,d)
32.  while (found_node→fullylinked)
33.   { }
34.  }
35.  continue
36.  }
37.  else
38.  {
39.   //create a new node x and set its value
40.   //connect the new node at level returned by random function
41.   temp=current_level
42.   current_level=current_level+1
43.   if(k>temp)
44.    { 
45.      try {
46.       s=save[temp]
47.       t=save[temp]→right
48.       if (s!=null)
49.         s→lock.lock()
50.       valid = ! s→marked&&!t→marked&&s→right==t
51.      }
52.     if (! Valid) continue
53.    else         // create new Head(H1) and tail(T1)
54.      h1→data=∞
55.      h1→right=x
56.      h1→left=null
57.      h1→down=h
58.      h=h1
59.      t1→data=∞
60.      t1→right=null
Procedure insert_node(key,d)
63. t1→left=x
64. t1→down=t
65. t=t1
66. /*update the new node pointers*/
67. x→left=h1
68. x→right=t1
69. if((save[temp]→right→down)==NULL &&
    (save[temp]→right→data>x→d))
70. {
71. x→down= save[temp]→right
72. }
73. }
74. elseif (k==1) /* k is from existing levels*/
75. try
76. {
77. s=save[k]
78. t=save[k]→right
79. d=save[k+1]→right
80. if (s!=null&& d !=Null)
81. s→lock.lock()
82. d→lock.lock()
83. valid = ! s→marked&& !t→marked&& !d→marked&& s→right==t
84. }
85. if (! valid) continue
86. else
87. {
88. x→left=save[k]
89. x→right=save[k]→right
90. save[k]→right→left=x
Procedure insert_node(key,d)
91. if((save[k-1]->right->down)==NULL & & (save[k-1]->right->data>x->d)) then
92. x->down= save[k-1]->right
93. else
94. x->down=NULL
95. }
96. else //k is in between the current level ,the else of if-then-elseif-
    else
97. {
98. try
99. {
100. s=save[k]
101. t=save[k]->right
102. d=save[k+1]->right
103. u=save[k-1]->right
104. if (s!=null & & d !=Null & & u!=null)
105. {
106. s->lock.lock()
107. d->lock.lock()
108. u->lock.lock()
109. }
110. valid = ! s->marked&& !t->marked&&
      !d->marked&&u->marked&&s->right==t
111. }
112. if (! valid) continue
113. else
114. {
115. node *x
116. x->left=save[k]
117. x->right=save[k]->right
Procedure insert_node(key,d)
118. save[k]→right→left=x
119. if((save[k-1]→right→down)==NULL OR (save[k-1]→right→data>x→d)) then
120. x→down= save [k-1]→right
121. if ((save[k+1]→right→down)==NULL OR
122. (save[k+1]→right→data<x→d)) then
123. x→down= save [k+1]→right
124. else
125. x→down==null
126. ]
127. }
128. x→fullylinked==TRUE
129. return true
130. finally
131. unlock(s,d,u)
132. }
133. }

Figure:3.3 Lock based insertion in MSL

### 3.3.3 Delete_Node

The delete_node operation shown in Figure 3.4, likewise calls find_Node to determine whether a node with the appropriate key is in the list in (lines 8-23). If so, the thread checks whether the node is “okay to delete” means it is fully linked, not marked (line 28). If the node meets these requirements, the thread locks the node and verifies that it is still not marked. If so, the thread marks the node, which logically deletes it; that is, the marking of the node is the linearization point of the remove operation(lines 30-34). The remaining part of the
procedure accomplishes the “physical” deletion, removing the node from the list by first locking its predecessors, upward, and downward node (lines 53-55). As in the insert_node operation, before changing any of the deleted node’s predecessors, the thread validates that those nodes are indeed still the deleted node’s predecessors. This is done using the weak validate function, which is the same as validate except that it does not fail if the successor is marked, since the successor in this case should be the node to be removed that was just marked (lines 57-60). If the validation fails, then the thread releases the locks on the old predecessors (but not the deleted node) and tries to find the new predecessors of the deleted node by calling find_Node again. However, at this point it has already set the local isMarked flag so that it will not try to mark another node. After successfully removing the deleted node from the list, the thread releases all its locks (line 65) and return true.
Flow Chat (3-1) : Deletion in lock Base MSL
Flow Chat (3-2) : Deletion in lock Base MSL
Procedure delete_node (int v)
1. node *delete_node = null;
2. bool ismarked = false
3. int found_node = -1
4. node *save,*t,*prev,*succ,*up
5. //search the msl for a key k of d and save the necessary pointers.
6. p=h
7. i=current_level
8. while (p#null) do
9. { 
10. while (p->data->key<key) do
11. { 
12. p=p->right
13. }
14. if (p->data->key==key&& found_node== -1)
15. { 
16. found_node=i
17. Write “searched node is found at i” and stop
18. }
19. else
20. { 
21. save[i] =p
22. p=p->left->down
23. i=i-1
24. } 
25. }
Procedure delete_node (int v)
26. while (true)
27. {
28. if (ismarked || (found_node! = -1 && right_to_delete (p, found_node))
29. {
30. if (!ismarked)
31. {
32. delete_node = p
33. delete_node.lock.lock()
34. if (delete_node.marked)
35. {
36. delete_node.lock.unlock()
37. return false
38. }
39. delete_node.marked = true
40. ismarked = true
41. }
42. try
43. {
44. prev = delete_node.left
45. succ = delete_node.right
46. upp = save [i+1].left.dup
47. if (prev = null && succ = null && upp = null)
48. {
49. if (save [i+1].left.dup = p)
50. save [i+1].left.dup = p.right
51. } // end of try
52. delete_node.lock.unlock()
Procedure delete_node (int v)
53  prev.lock.lock()
54  succ.lock.lock()
55  upp.lock.lock()
56  }
57  valid= !prev.marked&&!succ.marked&&!upp.marked
58  if (! valid) continue
59  delete_node(left)->right=delete_node(right)
60  delete_node(right)->left=delete_node(left)
62  return true
63  }
64  finally {
65  unlock(prev, succ, upp)
66  }
67  else
68  return false
69  }
70  }

Figure: 3.4 lock based deletion of node form MSL

Boolright_to_delete(node *c,int f)
{
  return(c->fullylinked&&c->marked)
}

Figure: 3.5 to check the linked node
3.4 CORRECTNESS

This section, sketch a proof for lock based concurrent modified skip-list algorithm. There are two properties to prove algorithm correctness: that the algorithm implements a linearizable set, it is deadlock-free, which we define more precisely below

3.4.1 LINEARIZABILITY

Linearizability [Herlihy [26]] is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

For the sake of linearizable proof, few assumptions are made like:

i) Nodes are initialized with their keys

ii) Next pointers of nodes are initialized with null

iii) The fullylinked and marked fields of nodes are initialized with false value

With these assumptions in mind we can drive the following lemma:

Lemma: for a node $n$ and $0 \leq j \leq n \rightarrow \text{toplayer}$:

$$n \rightarrow \text{right}[j] \neq \text{null} \text{ and } n \rightarrow \text{key} < n \rightarrow \text{right}[j] \rightarrow \text{key}$$

we can define a relation $\rightarrow_i$ so that $x \rightarrow_i y$ (read “$x$ leads to $y$ at layer $i$”) if $x \rightarrow \text{right}[i] = n$ or there exists $x'$ such that $x \rightarrow_i x'$ and $x' \rightarrow \text{right}[i] = n$; that is $\rightarrow_i$ is the transitive closure of the relation that relates nodes to their immediate successors at level $i$

using these observations, we can show that if $x \rightarrow_i y$ in any reachable
state of the algorithm, then \( x \rightarrow_i y \) in any subsequent state unless there is an action that remove \( n \) out of the level-\( i \) list, claim is already proved by [Herlihy [26]], and that can also be applicable on our algorithm. Because \( n \) must already be marked before being removed out of the MSL, and the fullylinked flag is never set to false value, this claim implies that a key can be removed from the abstract set only by marking its node.

### 3.4.2 DEADLOCK FREEDOM

The algorithm is deadlock free because a thread always acquires locks on nodes with larger keys first, if a thread holds a lock on a node with \( k \) then it will not attempt to acquire a lock on a node with key greater than or equal to \( k \).

### 3.5 CONCLUSION

We introduced a concurrent modified skiplist using a remarkably simple algorithm. The wait free traversal in concurrent MSL leads to simpler and possibly more efficient algorithms for related data structures such as dictionaries.