CHAPTER 5
LOCK-FREE CONCURRENT PRIORITY QUEUE BASED ON MSL

5.1 INTRODUCTION

Priority queues are fundamental in the design of modern multiprocessor algorithms. Priority queues are useful in scheduling, discrete event simulation, networking (e.g., routing and realtime bandwidth management), graph algorithms (e.g., Dijkstra’s algorithm), and artificial intelligence (e.g., A*search). In these and other applications, not only is it crucial for priority queues to have low latency, but they must also offer good scalability and guarantee progress. Priority queues with parallel access are an attractive data structure for applications like prioritized online scheduling, discrete event simulation, or branch-and-bound. This chapter proposes an alternative approach: to base the design of concurrent priority queues on the Modified Skip List data structure. To this end, we show that a concurrent modified Skip List structure, following a simple set of modifications, provides a concurrent priority queue with a higher level of parallelism. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system’s overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Previously known non-blocking algorithms of priority queues did not perform well in practice because of their complexity, and they are often based on non-available atomic synchronization primitives. This chapter begins to confrontation the issue of designing an efficient concurrent priority queue based on skip
list data structure of \{W.Pugh [97]\}. The other popular heap structures found throughout the literature \{ Ayani [78],sajal [89], carlo [5], sushil [91], rao [64], sanders [61]\}. This concurrent priority queue is designed with a change in the structure of modified skip list; it is presented in the simple form and produced significant performance gain.

5.2 Threaded Modified Skip List (TMSL) & Priority queue

The TMSL is a highly distributed data structures, a modified version of MSL. Surprisingly, Modified Skip Lists have received little attention in the parallel computing world, in spite of their highly decentralized nature. Chapter 2 has already detail about Modified skip list(MSL) and operation on it. It’s a search structure in which each node has one data field and three pointer fields: left, right, and down. Each level 1 chain worked solely as doubly linked list. The down field of level 1 node x points to the leftmost node in the level 1-1 chain that has key value larger than the key in x. H and T respectively, point to the head and tail of the level lcurrent chain. Underneath figure 1 shows the MSL. The previous chapter 4 introduced a lock-free implementation of MSL \{Ranjeet, Pushpa [62]\}.

![Figure:5.1 MSL upto 4 level](image-url)
In case of MSL the minimum element is the first one in one of the current chains as in above figure minimum element of first chain is 3, second one 6 & so on. If there is an additional pointer filed in each node, we can thread the elements in an MSL into a chain. The elements appear in non-decending order on this chain. The threaded chain with the addition pointer for the above MSL structure is look like(3→6→7→9→12→17→19→21→25→26). Subsequent threaded structure is referred to as TMSL (threaded modified skip list) shown in underneath figure. 5.2

![Figure:5.2 TMSL upto 4 levels](image)

### 5.3 Operations on TMSL:

The basic operation that can be perform on Threaded structure of MSL are, insertion, deletion. When a TMSL is habituated, the delete min operation can be accomplished in O (1) expected time. There is no insistence of down pointer for connecting one level to another level. There will be a pointer which works as a junction for threaded chain of MSL. The underneath section talk about these operations with their algorithm. Below is the node structure of TMSL.
5.3.1 Insertion:

The algorithm for insertion of new entry d in existing TMSL (priority queue) uses random number generator function to decide the level for new entry. We begin the insertion of a new entry (d) with value generated by random number generator function, if it is more than the current level. The new level is created with their respective head and tail. The node x is updated with their left, right, and thread_ptr. We call this new node as x in the presented algorithm. The thread_ptr of new node x is updated by comparing the key k of new node x with thread_head i.e. if x→key < thread_head, then x→thread_ptr=thread_head→right and set thread_head=x. Otherwise search for a node in threaded chain whose key is more than x→key, and update the thread_ptr accordingly. The steps of the insertion algorithm are given underneath in figure 5.4.
5.3.2 Deletion: The TMSL is represented as priority queue, there is delete_min operation is performed on It., which removes the minimum element from TMSL. The algorithm for deletion of minimum from threaded chain is given underneath.

**Algorithm: Insert_node(d)**

begin
1. Randomly generate the level k at which new node d to be inserted.
2. Create a new node x, and set x → data
3. If (k > current_level) then
4. Current_level = current_level + 1
5. Create a new chain in MSL H, with head node, node x and tail
6. Connect this chain to H
7. Update H
8. Set x → thread_ptr, accordingly the x → data
9. else
10. search the place where the new entry is to be inserted.
11. update the left and right pointer of new node x as well as preceding and succeeding of new node x.
12. update the thread_ptr of new node, preceding and succeeding node
13. Update the thread_head accordingly

**Algorithm: Delete_min()**

Begin
1. Delete the first node x from the threaded chain
2. Find out the level at which x is present say it’s k
3. Delete x from the level k list, updating left, right & thread_ptr of to-be-previous & to-be-next node of deleted node x
4. If the deleted node is the first node of threaded chain, then update the thread head and previous pointer of to-be-next node of threaded chain.

end

Figure: 5.4 insertion in TMSL

Figure: 5.5 delete_min in TMSL
Algorithm begins with removal of first element of threaded chain, than the level at which minimum element is present is searched say it k. the deletion process will be accomplished after updating the pointers of adjacent nodes, node to be delete.

5.4 Concurrent Access to TMSL

By virtue of concurrent operation on priority queue based on TMSL the similar approach of algorithm discussed in chapter 4 are applicable. Like reference counting technique for memory management, FAA & CAS synchronization primitives and, shared memory multiprocessor system, for the safe handling of memory, the memory management functions like READ_NODE, COPY_NODE, RELEASE_NODE, to set the deletion mark in to be deleted node.

5.5 Algorithm

In a concurrent environment the operations that can be performed on the structure are insertion, delete_min. For doing insertion (or delete min) of a node in TMSL, we need to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. This can be possible if we have additional information on each node about its insertion (or deletion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. Unlike the MSL structure there is no need of down field to connect the levels of MSL. There is a thread pointer in each node which makes a threaded chain. The beneath figure shows the node structure in concurrent environment.
5.5.1 Insertion

Subsequently randomly picking a level for the node, a comparison is done with existing level & generated level. If the generated level is bigger than existing one, new level is created with subsequent head & tail. I) Atomically update left and right pointer of newly inserted node II) update the next pointer of the to-be-previous node and III) atomically update the prev pointer of the to-be-next node.IV) to connect the newly inserted node with threaded chain by updating the thread pointer. For doing III step of insertion process update_insert procedure is used and for IVth step is done by update_thread function. The algorithm step for insertion are given in Figure 5.7.

If the level generated by randomlevel() function is more than k,(lines 3-27) try to create a new level and insert the new node x there, if it is not than the appropriate location of the new node is searched and

```
Union link::word
<p, d> :< pointer to node, Boolean>
Structure Node
{
  Value: pointer to word
  left: union link
  right: union link
  thread_ptr :union link
}

//local variables
Node, prev, prev2, next, next2:pointer to node
Last,link1: union link,
```

Figure: 5.6 node structure & variable used
inserted(lines 29 -39). The (line 40) update the right pointer of the previous of newly inserted node x. The thread field of node x, and next to node x is updated by sub procedure update-thread.

Flow Chat (10-1) : Insertion in lock Free TMSL
Flow Chat (10-2) : Insertion in lock Free TMSL
Flow Chat (11): Update_thread for insertion in lock Free TMSL
Function insert_node(key int, value: pointer to word)

1.  {
2.      node *p,*t,*save[max],*t_right,*up,*found_node
3.      k=randomlevel ()
4.      if(k>current_level)
5.      current_level=current_level+1
6.      temp=current_level
7.      x=create_node (key, value)
8.      COPY_NODE(x)
9.      node1=COPY_NODE(head)
10.     If(k>temp) // the generated level is more than the existing level
11.      {
12.          //create new head and tail
13.          h1=createnode(∞,∞)
14.          copy_node(h1)
15.          h1->left=null
16.          h1->right=x
17.          RELEASE_NODE(H1)
18.          t1=CreateNode(∞,∞)
19.          COPY_NODE (t1)
20.          t1->left=x
Function insert_node(key int, value: pointer to word)

21. 
22. t1\rightarrow right=NULL
23. RELEASE_NODE(t1)
24. x\rightarrow left=h1
25. x\rightarrow right=t1
26. RELEASE_NODE(t1)
27. RELEASE_NODE(h1)
28. }
29. else //the generated level is in between the existing levels
30. {
31. level=head_ptr[k] // head_ptr is array of pointer storing
32. address of head for each level
33. while(level\rightarrow key<x\rightarrow key)
34. while T do
35. if prev\rightarrow right != <next,F>
36. RELEASE_NODE(next)
37. next=READ_NODE (&prev\rightarrow right)
38. continue
39. x\rightarrow left=prev
40. x\rightarrow right=next
41. if CAS(&prev\rightarrow right,<next,F>,<x,F>)
42. COPY_NODE(x)
43. break
44. back-off
45. update_insert(x,next)
46. }
47. Update_thread(thread_ptr,x)

Figure;5.7 insertion of node in Lock Free Concurrent TMSL
Procedure update_insert(x,next:pointer to node)
{
  1.  While T do
  2.  link1=next→left
  3.  if (IS_MARKED (link1) || x→right!=<next,F>)
  4.  break
  5.  if CAS(&next→left,link1, <x,F>)
  6.  COPY_NODE(x)
  7.  RELEASE_NODE(link→p)
  8.  if IS_MARKED(x→left)
  9.  prev2=COPY_NODE(x)
 10.  prev2=update_prev(prev2,next)
 11.  RELEASE_NODE(prev2)
 12.  break
 13.  back-off
 14.  RELEASE_NODE(next)
 15.  RELEASE_NODE(x)
 16.  )

Figure: 5.8 update_insert: to update the left field of to be next node in ConcurrentLock-Free FTMSL
Figure 5.9 update_thread: to update the thread field of new node x

5.5.2 Deletion

The delete_min operation starts from thread_heads node and find the first node (del_node) in TMSL that does not have deletion mark. Once the deletion mark is set by CAS in (line 24). The next step is to call the help_delete sub-procedure to write the valid pointer on the right pointer of the previous node of the to-be-deleted node in TMSL chain. The algorithm for delete_min given underneath figure 5.9
The `update_prev` function will update the left pointer of the right node of the to-be-deleted node in MSL chain. Once the node is deleted from TMSL chain the next step is to update the thread_head, which points the next of `del_node`. The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock free property (that at least one thread is doing progress) on pre-emptive systems, whenever a node with deletion mark is set is found, it calls the `help_delete` operation. The `help_delete` operation, tries to set the deletion mark of the prev pointer and then atomically update the next pointer of the previous node of the to-be-deleted node. This operation might execute concurrently with the corresponding `delete_min` operation, and therefore both operations synchronize with each other.

Node of node it is updated to be the next node. The `update_prev` sub-function, tries to update the prev pointer of a node and then return a reference to a possibly direct previous node.

Because the `help_delete` and `update_prev` are habitually used in the algorithm for helping late operations that might influence otherwise stop progress of other concurrent operations. The algorithm is seemly for pre-emptive as well as fully concurrent systems. In fully concurrent systems though, the helping approach as well as heavy assertion on atomic primitives can relegate the performance significantly. Therefore after a number of consecutive failed CAS operations in an algorithm, puts the current operations into back-off mode, the thread does nothing for a while, and in this way steer disturbing the concurrent operations that might diversely progress slower. The duration of the back-off is initialized to some value (e.g. proportional to the number of threads) at the start of an operation, and for each consecutive entering of the back-off mode during one operation invocation, the duration of the back-off
is changed using some scheme.

Flow Chat (12-1) : Delete_min operation in lock Free TMSL
Flow Chat (12-2) : Delete_min operation in lock Free TMSL
delete_min(thread_ptr **node)
    {
        prev=COPY_NODE(thread_head)
        del_node=READ_NODE(&prev->right)
        if (del_node==NULL) then
            RELEASE_NODE(del_node)
            return null
        i=1
        while T do
            While(I<=current_level)
            {
                if(head[i]->next==del_node) then
                    chain_head=head[i]
                    break
                }
                else
                i++
            }
        link1=del_node->right
        if IS_MARKED(link1) then
            help_del(del_node)
            continue
        if CAS( &del_node->right,link1<link1.p,T) then
            help_del(del_node)
        next=READ_NODE(&del_node->right)
        prev2=COPY_NODE(chain_head)
        prev2=update_prev(prev2,next)
delete_min(thread_ptr **node)

28. RELEASE_NODE(prev2)
29. RELEASE_NODE(next)
30. link2=READ_NODE(del_node->thread_ptr)
31. thread_head=COPY_NODE(link2)
32. continue
33. break
34. RELEASE_NODE(del_node)
35. RELEASE_NODE(link2)
36. back-off
37. return
38. }

Figure 5.10: Deletion of minimum node value from TMSL

Procedure mark_prev(pointer to node node)
{
    while T do
        link1=node->left
        if IS_MARKED(link1) OR CAS(&node->left,link1,<link1.p,T>
            break
    }

Figure 5.10: Deletion of minimum node value from TMSL
Procedure Help_Del(node: pointer to Node)
{
  1. Mark_Prev(node)
  2. last=NULL
  3. prev= READ_NODE(&node right)
  4. next= READ_NODE (&node right)
  5. while T do
  6.   if prev = = next then
  7.     break
  8.   if IS_MARKED(next right) then
  9.     mark_prev(next)
 10.   Next2= READ_NODE (&next right)
 11.   RELEASE_NODE(next)
 12.   next=next2
 13.   continue
 14.   prev2= READ_NODE (&prev right)
 15.   if prev2 = NULL then
 16.   if last != NULL then
 17.     Mark_Prev(prev)
 18.   next2= READ_NODE (&prev right)
 19.   if CAS(&last right,<prev,F>),<next2,F>)
 20.   RELEASE_NODE(prev)
 21. else
 22.   RELEASE_NODE(next2)
 23.   RELEASE_NODE(prev)
 24.   prev=last
Procedure Help_Del(node: pointer to Node)

25. last=NULL
26. else
27. prev=prev2
28. continue
29. prev2=READ_NODE(&prev\rightarrow left)
30. RELEASE_NODE(prev)
31. prev=prev2
32. continue
33. if prev2 != node then
   {
34. if last !=NULL then
35. RELEASE_NODE(last)
36. last=prev
37. prev=prev2
38. continue
39. RELEASE_NODE(prev2)
40. if CAS(&lprev\rightarrow right, <node,F>,<next,F>)
41. COPY_NODE(next)
42. RELEASE_NODE(node)
43. break
44. back-Off
45. if last != NULL then RELEASE_NODE(last)
46. RELEASE_NODE(left)
47. RELEASE_NODE (next)
48. }
Function update_prev(prev,nodex: pointer to Node): pointer to Node

{
    1.   last=NULL
    2.   while T do
    3.   prev2:=READ_NODE(&prev[right])
    4.   if prev2 = NULL
    5.   if last != NULL
    6.   mark_prev(prev)
    7.   next2:=READ_NODE(&prev[right])
    8.   if CAS(&last[right], prev,F),<next2,F>)
    9.   RELEASE_NODE (prev)
   10.  else
   11.  RELEASE_NODE (next2)
   12.  RELEASE_NODE (prev)
   13.  prev=last
   14.  last=NULL
   15.  else
   16.  prev2=READ_NODE(&prev[left])
   17.  RELEASE_NODE (prev)
   18.  prev=prev2
   19.  continue
   20.  link1=node->left
   21.  link1=node->left
   22.  if IS_MARKED(link1)
   23.  RELEASE_NODE (prev2)
   24.  Break
   25.  if prev2!= node
26. if last!= NULL
27. RELEASE_NODE(last)
28. last=prev
29. prev:=prev2
30. continue
31. RELEASE_NODE (prev2)
32. if link1→p = prev
33. break
34. if (prev→right = node) & & CAS(&node→left,link1,<prev,F>)
35. COPY_NODE(prev)
36. RELEASE_NODE (link1→p)
37. if IS_MARKED(prev→left)
38. break
39. back-Off
40. if last != NULL
41. RELEASE_NODE (last)
42. return prev

Figure:5.12: update_prev function update the left pointer of right nod

5.6 CORRECTNESS

This section describes the correctness of presented algorithm. Here we outline a proof of linearizability (Herlihy [26]) and then we prove that algorithm is lock-free. Few definitions are required before giving proof of correctness.

Definition 1: We denote with Mt the abstract internal state of a threaded modified skip list as priority queue at the time t. Mt is viewed as a set of values (p,w) consisting of a unique priority p and a corresponding value w. The operations that can be performed on the structure are Insert (I) and Delete_min(DM). The time t1 is defined as the time just before the atomic execution of the operation that we are
looking at, and the time \( t_2 \) is defined as the time just after the atomic execution of the same operation. The return value of true2 is returned by an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is \( M_1 : O_1; M_2 \), where \( M_1 \) is the conditional state before the operation \( O_1 \), and \( M_2 \) is the resulting state after performing the corresponding operation:

\[
[p_1,\_] \in M_{t_1} : I_1([p_1, w_1]) = \text{TRUE},
\]

\[
M_{t_2} = M_{t_1} \cup \{ [p_1, w_1] \},
\]

\[
[p_1, w_{11}] \in M_{t_1} : I_1([p_1, w_{12}]) = \text{TRUE}_2
\]

\[
M_{t_2} = M_{t_1} \setminus \{ [p_1, w_{11}] \} \cup \{ [p_1, w_{12}] \}
\]

\[
[p_1, w_1] = \min \{ \min [p, w] \mid [p, w] \in M_{t_1} \}
\]

\[
DM_1() = [p_1, w_1], M_{t_2} = M_{t_1} \setminus \{ [p_1, w_1] \}
\]

\[
M_{t_1} = : DM_1() = \text{NULL}
\]

**Definition 2:** In order for an implementation of a shared concurrent data object to be linearizable (Herlihy [26]), for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.

**Definition 3:** The pair \([p, w]\) is present \(([p, w] \in M)\) in the abstract internal state \(M\) of implementation, when there is a connected chain of next pointers (i.e. \(\text{prev} \rightarrow \text{link} \rightarrow \text{right}\)) from a present node in the doubly linked list that connects to a node that contains the value \(w\), and this
node is not marked as deleted (i.e. is_marked (node) =false).

**Definition 4:** The decision point of an operation is outline as the atomic statement where the consequences of the operation is finitely decided, i.e. independent of the result of any sub operations after the decision point, the operation will have the same result. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.

We will now practice these definitions to show the execution history of point where the concurrent operation occurred atomically.

**LEMMA 1:** An insert_node operation which flourish (I [p, w]) = true), takes effect atomically at one statement.

**PROOF:** The decision point for an insert operation which succeeds (I [p, w] = true ) when the CAS sub-operation CAS(& prev right, <next, F>,<x, F>) of insert operation succeeds, and the insert operation will finally true. The state of the list (Mt1) directly before passing of the decision point must have been [p, _] ∈ Mt1, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be [p, w] ∈ Lt2.

**LEMMA 2:** A Delete_Min operation which get ahead (D() = [p, w]), takes effect atomically at one statement.

**PROOF:** The verdict point for an delete_min operation which succeeds (DM () = [p, w] ) is when the CAS sub operation CAS (&del_node right,link1 < link1.p,T) flourish. The state of the list (Mt) directly before passing of the decision point must have been [p,w] ∈ Mt, otherwise the case would have failed. The state of the list
after passing the decision point will be \([ p, \_] \in L_t\).

**Lemma 3:** A delete_node operation which fails \((DM() =\text{NULL})\), takes effect atomically at one statement.

**Proof** The decision point for a delete operation which fails \((DM() =\text{NULL})\) is the check in line if \((\text{del_node} == \text{NULL})\). State of the list \((M_t)\) directly before the passing of the state-read point must have been \(M_t = \emptyset\).

**Lemma 4:** With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.

**Proof:** Here we examine the possible execution paths of our implementation of TMSL. There are numerous conceivably unbounded loops that can stalling the termination of the operations. We call these loops retry-loops. The retry-loops take place when sub-operations search-out that a shared variable has changed value. This is observed either by a subsequent read sub-operation or a failed CAS. These shared variables are only adapted concurrently by other CAS sub-operations. According to the explanation of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding Insert or delete_min operation will progress. Thus, the one operation will always progress independent of any number of concurrent operations.
5.7 CONCLUSIONS

This chapter has introduced a lock free implementation of priority queue on the concept of TMSL. The TMSL is constructed with minor modification in MSL. The presented algorithm is first step to lock free algorithmic implementation of priority queue with MSL. It uses a fully described lock free memory management scheme, the atomic primitives used in this chapter are available in modern computer system.