International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:2, 2008

On Bounding Jayanti's Distributed Mutual Exclusion Algorithm

Awadhesh Kumar Singh

Abstract**—**Jayanti's algorithm is one of the best known abortable mutual exclusion algorithms. This work is an attempt to overcome an already known limitation of the algorithm while preserving its all important properties and elegance. The limitation is that the token number used to assign process identification number to new incoming processes is unbounded. We have used a suitably adapted alternative data structure, in order to completely eliminate the use of token number, in the algorithm.

Keywords**—**Abortable, deterministic, local spin, mutual exclusion.

I. Introduction

THE mutual exclusion is a classic problem. However, distributed mutual exclusion problem, being insidious in nature, has been a favorite area of research since many decades. In a distributed mutual exclusion algorithm, the code of each process *p* is divided into four sections: Entry Section, Critical Section (CS), Exit Section, and Remainder Section. In order to make it abortable, we introduce an Abort Section, which makes it possible for a process that waits "too long" to abort its attempt to acquire the resource. A process *p* may initiate a new attempt while in the Remainder Section. A successful attempt consists of executing the Entry Section, then the CS and finally the Exit Section. After completing the Exit Section, *p* goes back to the Remainder Section. When *p* busywaits in the Entry Section, it can (nondeterministically) choose to abort its attempt, in which case it executes the Abort Section and then goes back to the Remainder Section. The problem is to design an algorithm for Entry, Exit and Abort Sections so that the following properties hold:

1. *Safety*: mutual exclusion and deadlock freedom
2. *Liveness*: lockout freedom, bounded abort, and bounded exit
3. *Fairness*: first-come-first-served (FCFS)

In order to have better performance, preferably, the algorithm should be local-spin as well as adaptive.

We have considered Jayanti's algorithm [1] because it handles distributed mutual exclusion problem succinctly and satisfies all the properties stated in the previous section. Moreover, the algorithm has $O(n)$ space complexity and $O(min(k, \log n))$ remote reference complexity, where *n* is the total number of processes in the system and *k* is the contention. The performance of algorithm is good, in both cases, when the level of contention is low or high. At low levels of contention ($k << n$), the number of remote references made by the algorithm is proportional to *k*; and at high levels of contention ($k \approx n$), the number of remote references is bounded by log *n*. Thus, the algorithm performs well at all levels of contention.

The algorithm uses 64-bit objects supporting the LL, SC, read and write operations, which are described in Fig. 1.

The operation LL(*O*) returns *O*'s value.
The operation SC(*O*, *v*) by a process *p* "succeeds" if and only if no process performed a successful SC on *O* since *p*'s latest LL. If SC succeeds, it changes *O*'s value to *v* and returns *true*. Otherwise, *O*'s value remains unchanged and SC returns *false*.

Fig. 1 The Behavior of LL and SC Operations

Although, real machines do not support LL and SC operations, however, there are constant time implementations of 64-bit LL/SC objects from 64-bit compare&swap objects and from 64-bit "realistic" LL/SC objects [2]. As a result, the algorithm can run on almost all modern machines, as they support either compare&swap (*e.g.*, UltraSPARC [3] and Itanium [4]) or realistic LL/SC (*e.g.*, POWER4 [5], MIPS [6] and Alpha [7]).

Jayanti's algorithm [1] has a limitation that the token numbers, assigned to new incoming processes, grow unbounded, that is, its value increases indefinitely. Our algorithm is a modified form of Jayanti's algorithm, without sacrificing any of its salient properties. We have attempted to modify it in a deterministic way. The approach is based on replacing the data structure used in Jayanti's algorithm, namely *f*-array [8], with a suitably adapted alternative data structure, in order to completely eliminate the use of token numbers. Accordingly, the code of Jayanti's algorithm has also been modified, a little bit. The proofs of various properties have been included in the analysis of the modified algorithm.

II. Jayanti's Algorithm

The readers may refer [1] to see Jayanti's complete algorithm, its working, complexity analysis, and detailed formal proof. However, in order to have a quick look over various pieces of the algorithm that work together to prevent

A. K. Singh is with the Department of Computer Engineering, National Institute of Technology, Kurukshetra, 136119, India (e-mail: aksinreck@rediffmail.com).

undesirable race conditions, the pseudo code is presented in the following Fig. 2.

A. The Shared Variables

The algorithm is based entirely on LL/SC variables. Although, some variables (specifically, *C* and *Q*) are not LL/SC variables, however, they can be efficiently implemented from LL/SC variables. Now, we describe below the use of shared variables of the algorithm.

***Wait*[*p*]**

Before entering the CS, any process *p* busywaits on this boolean flag. At the start of its Entry Section, *p* sets it to *true*. When it is assigned *false*, by some other process *q*, *p* is released from its busywait loop and *p* becomes the owner of CS. *Wait*[*p*] is allocated to *p*'s memory module, in order to make the algorithm local-spin.

CSowner

It holds the name of process, which is current owner of the CS. *CSowner* is assigned ⊥, if no process currently owns the CS.

Counter *C*

It is used to assign token numbers to processes requesting the CS. Any process *p*, incrementing *C* by executing inc(*C*, 1), gets the new value of *C* as its token number. Any other process *q*, incrementing *C* after *p*, would get a higher token number than that of *p*. As a result, the algorithm maintains FCFS and lockout-freedom properties.

Priority process-queue *Q*

It is a priority process-queue to hold the names and token numbers of processes waiting to enter CS. In the Entry Section, a process *p* inserts in *Q* an element [*p*, *t*], where *t* is *p*'s token number. Process *p* deletes this element when exiting or aborting. An element can be deleted only by the process that inserted it; and a process can not insert a new element before deleting the older element inserted by it. As the priority ordering of elements in *Q* is similar to Lamport's clock system [9], opearation *findmin* returns the name of longest waiting process, that is, the element with smallest token number from *Q*. If Q is empty, *findmin* returns the special value [⊥, ⊥].

B. The Pseudo Code

```
Shared variables
C is a counter, initially 0; supports inc and read operations.
Q is a priority process-queue, initially empty; supports insert, findmin and delete operations.
CSowner takes on a value from, $\{\perp\} \cup \{0,1,...,n-1\}$ , initially ⊥; supports LL, SC, read and write operations.
$\forall p \in \{0,1,...,n-1\}$, Wait[p] is a boolean, arbitrarily initialized; supports LL, SC, read and write operations.
procedure Entry(p)
    Wait[p] = true
    inc(C, 1)
    t = read(C)
    insert(Q, [p, t])
    promote()
    promote()
    wait till Wait[p] = false
procedure Exit(p)
    delete(Q, [p, t])
    CSowner = ⊥
    promote()
procedure Abort(p)
    delete(Q, [p, t])
    promote()
    if CSowner = p then
        CSowner = ⊥
        promote()
procedure promote()
    if LL(CSowner) ≠ ⊥ then return
    [q, t′ ] = findmin(Q)
    if q ≠ ⊥ then LL(Wait[q])
    if SC(CSowner, q) then
        if q ≠ ⊥ then SC(Wait[q], false)
```

Fig. 2 Abortable mutual exclusion algorithm for *n* processes. Code shown here is for process *p*

III. The Doubly Linked Concurrent List

The tokens are required to maintain first-come-first-serve (FCFS) property of the algorithm. In place of *f*-array, another data structure, namely doubly linked concurrent list, has been used for maintaining the list of processes. Each new incoming process is added to tail of the list, so that the order of incoming processes, i.e. FCFS property, is already managed by the structure of the linked list.

A queue can be implemented by a singly linked list, however, we have used doubly linked list, as it supports the traversal from both ends. The detailed discussion on parallel linked list data structure is available in the paper by Tang et. al. [10]. The algorithms for various operations over a parallel linked list were originally designed for processor self-scheduling on parallel computers. However, we have modified them to accommodate our requirement. Before we present the algorithms for append, delete, and search operations, we describe the structure of a node for the doubly linked concurrent list. We consider that our list goes from left to right. In addition to the data field that depends on the application of the linked list each node (representing a process), as shown in Fig. 3, has four fields. Originally, a node in Tang et. al.'s list [10] had only three fields; however, we have added one more field, namely *flag*, in order to make it suitable for error-free operation of the linked list in our algorithm. The fields are described below:

1. *left* : It is a pointer that contains the address of the left node in the list.
2. *right*: It is a pointer that contains the address of the right node in the list.
3. *lock*: This field can assume any value from the set {0, 1, 2}. Value 0 indicates that the node is being deleted; value 2 indicates that the node is presently locked by another process, and value 1 represents the unlocked state of the node.
4. *flag*: A node can set its *flag* when the node has to be deleted by the process either exiting from CS or aborting. When a node's *flag* is set the node cannot be locked by any other process. Also, if a node is already locked by some other

process and subsequently the node sets its *flag*, then no process can lock it further.

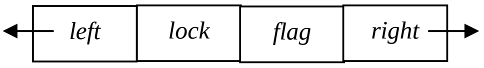

Fig. 3 Structure of the node

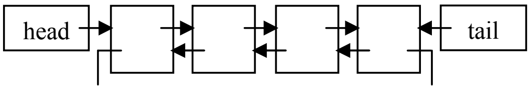

Fig. 4 The Doubly Linked Concurrent List

Fig. 4 shows a doubly linked concurrent list. Head is the pointer to first node of the list and tail is the pointer to last node of the list.

The *lock* is a synchronization variable used for coordinating the deletion of adjacent nodes. Each *lock* has three states: unlocked (*lock* = l), locked (*lock* = 2), and closed (*lock* = 0). The initial state is unlock (*lock* = l). The *flag* represents the state of the node whether it is being deleted by the process. It can be set only by the process that owns the node and when set, it cannot be locked by any other node. The operations that are supported by the list (append, delete, search, and check) are explained below.

A. The Append Procedure

The Append procedure appends a new node after the last node of the linked list and splices the pointers accordingly. Multiple processors can append their new nodes concurrently. The procedure ensures that these concurrent operations always yield a well connected linked list. Also, append can execute in parallel with delete operation. However, deletion of the last node and appending of a new node are mutually exclusive operations. Each new node is initialized (*right* = *null*, *lock* = 1 (unlocked), and *flag* = 0) before it is appended to the list. When a processor appends a new node to the list, it needs the following steps:

Lock the last node of the linked list by changing its *lock* from 1 to 2. If its *lock* is not 1, repeat this step again.

Set the *left* of the new node. Accordingly, change the *right* of the last node. Also, change the tail pointer of the linked list.

Unlock the last node (original) by changing its *lock* from 2 to 1.

```
procedure append(p)
    begin
    do
    x = tail;                          //fetch address of last node
    if (x ≠ null) then                 //check if there is any node in the list
    if(x = tail & x → flag = 0)        //if the tail has not changed since
fetching its address
    {x → lock = l; increment};         //lock the last node
          endif
       else                            //if no node is present in the list
    if (x = tail) then                 //if the tail has not changed since
fetching its add.
              {HL = l; increment};     //lock the head node
          endif
       endif
    while (failure);           //repeat until the last node or head node is locked
    tail = p;                  //tail pointer now points to the new appending
node
    p → left = x;              //left pointer of new node points to original last
node
    if (x ≠ null) then         //if the original last node is present and not null
    x → right = p;             //right pointer of original last node points to the
new node
    else                       //if no node is there in the list
    head = p;                  //the head points to the new appending node
    endif
    if (x ≠ null) then         //if the original last node is present and not null
    {x → lock ; decrement}; //unlock the original last node
    else                       //if no node is there in the list
       {HL; decrement};        //unlock the head node
    endif
    end
```

In this paper, we use the syntax $p \rightarrow$ lock, similar to the language C, to denote the *lock* field of the node pointed by *p*. Other fields of the node are denoted similarly.

B. The Delete Procedure

The delete procedure allows multiple processors to delete different, non-adjacent, nodes simultaneously. In order to have proper splicing of pointers, the deletions of adjacent nodes should be mutually exclusive. When a processor deletes a node, it executes the following steps:

Set the *flag* and close the node to be deleted (i.e. the node pointed by *p*) by changing its *lock* from 1 to 0. If the *lock* is not 1, busy wait at this step until it becomes 1.

Lock the left node by changing its *lock* from 1 to 2. If its *lock* is not 1 or its *flag* is set, re-fetch the address of the left node and repeat this step.

Proceed to delete the node by changing the *right* of the left node and the *left* of the right node.

Unlock the left node by changing its *lock* from 2 to 1.

```
procedure delete(p)
    begin
    p → flag = 1;                   //set the flag to indicate that the node is
going to be deleted
    do
    {p → lock = 1; decrement};      //change the lock of the node to zero
    while (failure);                //repeat until the lock is set to 0
    y = p → right;                  //y points to the right node
    do
       x = p → left;                //x points to the left node
    if (x ≠ null) then              //if the current node is not the first node in
the list
          if (x → flag = 0 & x = p → right) then    //if left node is not going to
delete and left node has not changed yet
    {x → lock = 1; increment};                       //lock the left node
          endif
          if (!failure) then     //if left node has been locked
    x → right = y;               //right pointer of left node points to the right
node
          endif
       else                      //if the current node is the first node in the list
          head = y;              //head node points to the right node
       endif
    while (failure);             //repeat until right node’s right pointer or head
points to right node
    if (y ≠ null) then           //if current node is not the last node in the list
    y → left = x;                /right node’s left pointer points to the left node
    else                          //if current node is the last node in the list
        tail = x;                 //tail node points to the left node
    endif
    if (x ≠ null) then            //if left node is present and deleting node is not the first node
        {x → lock; decrement};    //unlock the left node
    endif
    end
```

First of all, the *flag* of the currently deleting node is set such that no other node can lock it later. Then the process tries to lock its own node. Some other node might have already locked it, so it loops until it succeeds in locking its node. After this, it fetches the address of its left node and tries to lock it. If the deleting node is not the first node in the list then the left node belongs to some other process. Before locking the left node its *flag* is checked, if it is unset then only proceed to lock it. The address of the left node is again checked to verify if it has not changed in the meantime. After locking the left node change its *right* to point to the right node of the currently deleting node. However, if the currently deleting node is the first node in the list, then change only the head pointer to point to the right node of the currently deleting node. The process loops until it succeeds in changing the *right* of the left node or the head pointer.

Now, the *left* of the right node is changed to point to the left node if the right node is not null, *i.e.*, currently deleting node is not the last node in the list. If it is the last node in the list, then change the tail pointer to point to the left node of currently deleting node. In the end, release the *lock* of the left node that was locked earlier by the currently deleting node.

If the left node's flag is set or it is locked then the deleting node has to wait until the left node is deleted. If all the left nodes are willing to delete them, i.e., their flags are all set, then deletion starts from the left most node and continues towards right until it reaches the currently deleting node.

C. *The Search Procedure*

The job of search procedure is as follows:

1. Starting from the first node, traverse the linked list; search for the first node that is eligible to enter the critical section, *i.e.*, the first node which is not closed and whose *flag* is unset.
2. If no such node is found return null.

```
procedure search()
    beign
    q = head;             //q points to the first node of the list
    while (!check(q))     //if check procedure returns false
        q = q → right;    //q points to right node
    endwhile              //repeat until check returns true
    return q;             //return value of q
    end
```

Initially, a temporary variable q is pointed to the head node of the list. This pointer is used to traverse through the list. Until the check procedure returns true, q moves rightwards. The check procedure is explained below.

D. *The Check Procedure*

The job of check procedure is to return *true* if the node *p* is not null and its *flag* is unset, *i.e.*, node *p* is eligible to enter the critical section, else *false* is returned.

```
procedure check(p)
    begin
    if (p ≠ null || p → flag = 0) then    //if node p is not null and its flag is unset
        return true;                      //return true to the calling procedure
    else                                  //if node p is null or flag is set or both
        return false;                     //return false to the calling procedure
    endif
    end
```

IV. The Modified Algorithm

Now, we present the modified mutual exclusion algorithm. Out of four shared variables, used by Jayanti, counter *C* and priority process-queue *Q* have been taken off, as they are no more required. However, *Wait*[*p*] and *CSowner* have been used for the same purpose as they were used, originally, in Jayanti's algorithm. The modified mutual exclusion algorithm, using doubly linked concurrent list, is as follows:

Shared variables

CSowner takes on a value from, $\{\perp\} \cup \{0,1,...,n-1\}$, initially ⊥; supports *LL, SC, read* and *write* operations.

$\forall p \in \{0,1,...,n-1\}$, *Wait*[*p*] is a boolean, arbitrarily initialized; supports *LL, SC, read* and *write* operations.

```
procedure Entry(p)
    1. Wait[p] = true    //Wait[p] is set true so that p busywaits on it
    2. Append(p)    //append the node of the process to the doubly linked concurrent list
    3. promote()    //call promote() to put the next process in CS
    4. promote()    //again promote() to prevent livelock
    5. wait till Wait[p] = false    //busywait on Wait[p] until it gets into CS or aborts
procedure Exit(p)
    6. delete(p)         //remove node from the linked list
    7. CSowner = ⊥       //set CSowner to null indicating no process in CS now
    8. promote()         //call promote() to put the next process in CS
procedure Abort(p)
    9. delete(p)         //remove node from the linked list
    10. promote()        //call promote() to put the next process in CS
    11. if CSowner = p then    //if CSowner is set to p by some other process
    12. CSowner = ⊥      //set CSowner to null
    13. promote()    //again call promote() to put next process in CS
procedure promote()
    14. if LL(CSowner) ≠ ⊥ then return    //if some process is in CS then return
    15. q = Search()    //Search() returns the first node in list eligible for CS
    16. if check(q) then LL(Wait[q])    //perform LL on Wait[q] if q is still eligible for CS
    17. if SC(CSowner, q) then    //if SC operation on CSowner is successful
    18. if check(q) then SC(Wait[q],false)    //if q is still eligible for CS set Wait[q] to false
```

V. The Proof of Correctness

We also need some definitions, similar as used by Jayanti [1], to carry out the description of various proofs. The system state (or configuration), is determined by the values of the shared variables, local variables, and program counters of all *n* processes. The configuration *C* changes when a process executes some step *s*. In the initial configuration, we assume that all *n* processes are in the Remainder Section and the shared variables are initialized. A run *R* is a (finite or infinite) sequence C_0, s_1, C_1, s_2, C_2, ... of alternating configurations and steps such that C_0 is the initial configuration and, for all $i > 0$, the step s_i is enabled in C_{i-1} and causes the configuration to change from C_{i-1} to C_i. In addition, we also follow the following notations used in Jayanti's paper [1]:

- An attempt by process *p* refers to each execution by *p* of the Entry Section followed by either the Exit or the Abort Section.
- *Line*(*p*, *k*, *m*) denotes the step in which process *p* executes Line *m* of the algorithm in its *k*th attempt.
- *Line*(*p*, *k*, 5[*last*]) is the step corresponding to the final iteration of process *p*'s busywait loop on Line 5, during the *k*th attempt by *p*, in which *p* reads *false* in *Wait*[*p*] and moves into the Critical Section.
- If π is an execution of promote() by some process, π(*m*) denotes the step of π corresponding to the execution of Line *m*. (For example, π(14) is the execution of Line 14 by π.)
- We call an execution of promote() successful if it executes Line 17 and its SC operation on Line 17 succeeds.

In the algorithm, a step is a call to *promote*, that is, the execution of Lines 3, 4, 8 and 10 by a process. The execution of iteration, of Line 5 that consists of reading *Wait*[*p*] and comparing it with *false*, is also a step. A process *p*, while busywaiting at Line 5, nondeterministically, chooses either to execute an iteration of Line 5 or to jump at Line 9 in order to execute the Abort Section.

The critical section (CS) has been modeled as Line $5'$ (although it is not shown in the pseudo code of the algorithm). When *p* executes an iteration of Line 5, if *p* reads *false* in *Wait*[*p*], then *p* enters the CS, *i.e.*, *p*'s program counter becomes $5'$. If *p* takes a step from CS, its program counter changes from $5'$ to 6.

The Remainder Section has been modeled as Line 0 (this is also not shown in the pseudo code of the algorithm). If *p* takes a step from the Remainder Section, its program counter becomes 1. When *p* completes the Exit Section or the Abort Section, it enters the Remainder Section.

A. Mutual Exclusion

Lemma 1 is that successful executions of promote() do not overlap, an observation that follows immediately from the semantics of LL and SC operations.

Lemma 1. If π and π' are distinct successful executions of promote(), then either π (17) < π'(14) or π'(17) < π(14).

Lemma 2. If process *p* enters the CS during its *k*th attempt, then there is an execution π of promote() such that π(17) writes *p* into *CSowner*, and *Line*(*p*, *k*, 1) < π(17) < *Line*(*p*, *k*, 5[*last*]).

Lemma 3. We state this lemma in two parts:

1. Consider any step *s* in which some process executes a successful SC operation on Line 17. *CSowner* has the value ⊥ in the configuration immediately preceding step *s*.

2. Consider any step *s* in which a process *p* executes either Line 7 or Line 12. *CSowner* has the value *p* in the configuration immediately preceding step *s*.

Lemma 4. If a process *p* is in the CS in a configuration *C*, then the value of *CSowner* in *C* is *p*.

Lemma 5. (Mutual Exclusion): At most one process is in the CS in any configuration.

Jayanti has verified the mutual exclusion property by proving above five lemmas. However, the proof of mutual exclusion, in our case, is simple and given as follows:

Proof. For any process *p* to be in CS, there must be some execution of π(15), where search() returns *p* and some execution of π(17), that successfully writes *p* in *CSowner*. The search() procedure returns the first node, eligible for CS, in the linked list. For two or more processes to be in CS at the same time, search() must return more than one different processes which is not possible. Hence, there can be only one process in CS at a time.

B. Deadlock Freedom

Following four conditions [11, 12] are necessary for a deadlock to occur:

1. *Mutual Exclusion:* each process has exclusive use of its resources.
2. *Nonpreemption:* a process never releases the resources it holds, until it is through using them.
3. *Resource waiting:* each process holds resources while waiting for other processes to release theirs.
4. *Cycle of waiting processes:* each process in the cycle waits for resources that the next process owns and will not relinquish.

If any of the above mentioned four conditions does not hold then deadlock cannot occur. Consider the fourth condition. In our algorithm, a resource refers to the lock of a node. Consider a situation where all the nodes have acquired their own lock and are waiting to lock their respective left nodes. Thus, each process is waiting for the resource that is locked by its left node. However, deadlock will not occur because this is not the case for the first node in the linked list. As, the first node has to acquire the head lock and the head cannot be locked by any node other than the first node in the list; hence, there is no cycle of waiting processes. Therefore, the fourth condition does not hold and deadlock cannot occur.

C. Lockout Freedom

Lockout freedom property states that if a process initiates an attempt and does not abort that attempt, then it eventually enters the CS in that attempt. Let *p* be a process, which initiated an attempt and would not abort that attempt. There are two ways in which *p* can be deleted from the list: either by aborting or by exiting. As it is interested in CS, abort cannot take place; hence the only way for a node to delete itself is through exit. Since deadlock is not possible, therefore, a process cannot be in the list forever. Now, for *p* to delete itself through the exit section, it has to enter the CS first. Hence, p eventually enters the CS.

D. Local Spin

Line 5 of our algorithm corresponds to busy-wait of each process *p* on a Boolean variable *Wait*[*p*] in order to get into the critical section. As, *Wait*[*p*] is mapped into the processor's local memory module or cache, while waiting, the process accesses only its local variable. Thus, the algorithm is local spin.

E. FCFS

Lemma 6. Let R be any finite run such that in the configuration C at the end of R, *CSowner* has a non-$\perp$ value p. Then, there exists $k \geq 1$ such that the following statement is true: Let π be the latest execution of promote() such that $\pi(17)$ is in R and $\pi(17)$ writes p into *CSowner* (by a successful SC operation). Such a π exists and satisfies *Line*(p, k, 2) < $\pi(15)$. (Note that this implies that R includes the step *Line*(p, k, 2).)

Proof. Let π be the latest execution of promote() such that $\pi(17)$ is in R and $\pi(17)$ writes p into *CSowner* by a successful SC operation (such a π exists because *CSowner* has the value $p \neq \perp$ in C). This implies that search() of $\pi(15)$ returns p. It is possible only if p is already present in the list, *i.e.*, p inserts its node into the list on Line 2 (in some attempt) and does not remove its node from the list in that attempt (by Line 6 or 9), before execution of $\pi(15)$. Thus, there exists an attempt $k \geq 1$ (of p) such that the following statement is true: *Line*(p, k, 2) < $\pi(15)$. This fact establishes the lemma.

Lemma 7. In any run R, if *Line*(p, k, 2) < *Line*(q, m, 1) and p does not abort its kth attempt, then q does not enter the CS in its mth attempt before p enters the CS in its kth attempt.

Proof. If *Line*(p, k, 2) and *Line*(q, m, 2) append p and q, respectively, in the list, then p's node is ahead of q's node in the linked list. Let us assume that the Lemma is false and consider that q gets into CS in its mth attempt before p. Thus, by Lemma 4, *CSowner* has value q in configuration C. Lemma 6 implies that there exists an execution π of promote() such that $\pi(15)$ executes search() after *Line*(q, m, 2), and $\pi(17)$ writes q in *CSowner*. This is possible if $\pi(15)$ receives q from search(). But search() returns the first, eligible for CS, process in the linked list. As, p is ahead of q in the linked list and also eligible for CS (since its not going to abort in the kth attempt), hence, search() would return q if and only if p is trying to abort or is in the process of deletion, which is not the case. Therefore, our assumption is false and the Lemma holds.

F. Bounded Exit

Lemma 8. If a process wants to acquire its own node's lock and some process has locked its node, then it would take a constant amount of time to acquire the lock.

Proof. Let p be the process that wants to acquire its own node's lock and some other process q has locked it. There are two possibilities to consider: (i) q is deleting its node from the linked list or (ii) q is appending its node to the linked list. If q is deleting its node then q must be the right node of p. In the delete operation, the only step that takes $O(k)$ time (where k is the number of contending processes in the linked list) is the locking of the left node and remaining steps require constant amount of time. As, q has already locked its left node, *i.e.*, p's node; hence, q would release the lock of p's node after constant amount of time. Also, p has already set its *flag* to 1 before trying to acquire its node's lock, thus, no other process can further lock p's node. Consequently, p would acquire its node's lock next, after q releases it. Hence the lemma holds.

Lemma 9. If two adjacent nodes are attempting to delete simultaneously, then the node, which is ahead in the list, is deleted first.

Proof. Let two neighboring nodes p and q, where p is ahead of q in the linked list, try to delete simultaneously. While attempting to delete simultaneously, say, both nodes have set their respective *flag* 1, simultaneously. Assume that the Lemma is false and q is deleted before p. In order to delete itself, a node has to lock its left node. This implies that q had been able to lock node p even after p's *flag* was set to 1. This results in a contradiction as a node cannot be locked while its *flag* is set to 1. Hence, our assumption is false and the lemma is true.

Lemma 10. The delete(p) operation takes at most $O(k)$ remote steps.

Proof. Let, the node performing the delete operation be p and its left node be q. There are three possibilities:

(i) *Best Case:* Node q's *flag* is not set to 1 and node p is also not locked. In this case, p takes total $O(1)$ time to set its own *lock* to 0 and q's *lock* to 2. So, the time complexity of delete(p) operation would be $O(1)$.

(ii) *Average Case:* Node q's *flag* is not set to 1 and node p is locked. From Lemma 8, we observe that acquiring the lock of node p would take constant amount of time. Furhtermore, locking node q and remaining operations also take constant time. Therefore, in this case also, the time complexity of delete(p) operation would be $O(1)$.

(iii) *Worst Case:* Node q's *flag* is set to 1 and node p is also not locked. Now, node p can set its *lock* to 0 in constant time. However, node p cannot lock node q as its *flag* is set to 1. In the worst case, all nodes, ahead of node p in the linked list, may be attempting to delete themselves, simultaneously. We conclude, from Lemma 9, if two adjacent nodes are trying to delete themselves simultaneously, then deletion starts from the node ahead in the linked list. Thus, only the head node can be locked. Consequently, the first node in the linked list would be deleted earliest, afterwards, the second node and so on. If k be the number of contending processes in the list, there would be (k-1) nodes ahead of p. Hence, in order to delete itself, p would take $O(k)$ time before it can lock the node ahead of it. Therefore, in the worst case, the time complexity of operation delete(p) would be $O(k)$.

Jointly, from lemma 8, 9, and 10, we conclude that the Exit section completes in at most $O(k)$ steps. As, k is some bounded integer, the algorithm ensures bounded exit property.

G. Bounded Abort

The time complexity of abort action would be $O(k)$, as all steps of abort section of the algorithm take $O(k)$ time. The complete proof follows directly from the proof of bounded exit.

H. Adaptivity

Lemma 11. The search() operation takes at most O(k) remote steps.

Proof. The search() operation returns the address of some node *p* that is the first node, eligible for CS, in the linked list. The criterion for eligibility is that the node should not be in the process of deletion, that is, its *flag* must be set to 0. The search() starts traversing the linked list from the first node and then moves rightward. In the worst case, out of *k* nodes ahead of *p* in the linked list none is eligible for CS. The search() would have to traverse *k* nodes in order to return address of *p*. Therefore, $O(k)$ remote accesses are required for completion of search() procedure.

After entering in the linked list at Line 2 of the algorithm, each process makes at most $O(k)$ remote memory accesses (both delete(p) and search() operations take $O(k)$ time according to Lemma 10 and Lemma 11). Thus, the time complexity of our algorithm is a function of *k*, where *k* is the number of contending processes, *i.e.*, the number of nodes in the linked list. Hence, the algorithm is adaptive.

I. Time Complexity Analysis

In worst case, the number of remote memory accesses required for a process to enter the critical section is $O(k)$, where *k* is the number of contending process. Thus, the time complexity of our algorithm is $O(k)$. The algorithm has $O(n)$ space complexity, where *n* is the total number of processes in the system.

VI. Conclusion

The tree data structure (or its any variant) works well when the nodes are inserted in a random order. It performs poorly on the sequences of operations, such as inserting the nodes in order. Hence, mostly for such applications, implementers generally agree that linked data structures are better options. Moreover, they are significantly easier to implement than any other data structure including tree and its variants. Therefore, we have used a suitable variant of linked data structure, namely doubly linked concurrent list, as underlying data structure. In our approach, by making small change in the pseudo code of Jayanti's algorithm, the use of token number has been completely eliminated. Although, we had to compromise in worst case time complexity, nevertheless, no important property has been sacrificed. Moreover, the proofs of various properties are straightforward and less complex than their counterparts in Jayanti's paper. The space complexity is same; nonetheless, the data structure used in Jayanti's algorithm, namely *f*-array, needs more complicated memory management mechanism than the linked data structures [13], which have been used by us. An inherent characteristic of linked data structures is the structural flexibility possible by manipulating pointers. Methods using arrays in programs may improve the execution efficiency but lack the structural flexibility [14]. On the basis of complexity analysis, we conclude that our algorithm under perform only when the rate of abortion is very high. However, the abortion rate is often quite low in most practical scenarios.

References

[1] P. Jayanti, “Adaptive and efficient abortable mutual exclusion,” in *Proc. 22nd ACM Annual Symposium on Principles of Distributed Computing*, July 13-16, 2003, pp. 295–304.

[2] P. Jayanti, “Efficient and practical constructions of LL/SC variables,” in *Proc. 22nd ACM Annual Symposium on Principles of Distributed Computing PODC 2003*, July 13-16, 2003, pp. 285–294.

[3] D. L. Weaver and T. Germond, *The SPARC Architecture Manual*. Version 9, SPARC International, Inc.

[4] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, Volume 1: Application Architecture Revision 2.1, Oct. 2002.

[5] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, *IBM e-server POWER4 System Microarchitecture*, IBM, Oct. 2001.

[6] MIPS Computer Systems, *MIPS64 Architecture for Programmers*. Volume II: The MIPS64 Instruction Set, Aug. 2002.

[7] R. Site, *Alpha Architecture Reference Manual*. Digital equipment Corporation, 1992.

[8] P. Jayanti, “*f*-arrays: Implementation and applications,” in *Proc. 21st ACM Annual Symposium on Principles of Distributed Computing PODC 2002*, July 21-24, 2002, pp. 270–279.

[9] L. Lamport, “Time, clocks and the ordering of events in a distributed system,” *Communications of the ACM*, 1978, 21(7): 558–565.

[10] P. Tang, P.-C. Yew, and C.-Q. Zhu, “A parallel linked list for shared-memory multiprocessors,” in *Proc. 13th IEEE Annual International Conference on Computer Software and Applications COMPSAC'89*, Sep. 20-22, 1989, pp. 130–135.

[11] M. J. Quinn, *Parallel Computing: Theory and Practice*. 2/e, McGraw-Hill Companies, Inc., 1994.

[12] E. G. Jr. Coffman and P. J. Denning, *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[13] V. J. Marathe, M. Moir, and N. Shavit, “Composite abortable locks,” in *Proc. 20th IEEE International Parallel and Distributed Processing Symposium IPDPS'06*, April 25-29, 2006, pp. 1–10.

[14] H. Maegawa, “Memory organization and management for linked data structures,” in *Proc. 19th ACM Annual Conference on Computer Science*, April 1991, pp. 105–112.