Fast Lock-Free Linked Lists in Distributed Shared Memory Systems

BY

MOHAMMAD FAROOK

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

FAST LOCK-FREE LINKED LISTS IN DISTRIBUTED SHARED MEMORY SYSTEMS

BY

MOHAMMAD FAROOK

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

In shared memory systems concurrent processes operate on shared data structures. The consistency of the data structures is maintained by mutual exclusion, which is implemented by locking. Locking is an easy way of ensuring consistency of the shared data structures, but it has several drawbacks. The major drawback is that it increases the latency of the all the processes as the shared data structure is accessed in a serial manner. It might also results in deadlocks when the process currently accessing the shared data structure is pre-empted or terminated.

An alternate approach to avoid mutual exclusion by locking is by implementing lock-free data structures. This is possible by using basic synchronization primitives such as Compare & Swap and Double Compare & Swap. Processes can concurrently read and write on these data structures and also maintain the consistency of the data structure. Lock-free data structures reduces the latency of the processes and avoids deadlocks.

This thesis presents lock-free linked lists along with the algorithms that operate on them. The algorithms were simulated and were compared with Greenwald & Cheriton's and Valois' algorithms. The design of the linked lists along with the effective and efficient use of the synchronization primitives in the algorithms provides a improved performance over the other two algorithms. The correctness of the algorithms is also presented in the thesis.

Acknowledgement

I am indebted to my supervisor Dr. Peter Graham for his guidance and support during my thesis.

I would like to thank my committee members Dr. Mark Giesbrecht and Dr. Robert Mcleod for their valuable suggestions.

I would like to express my sincere gratitude to my parents and brothers whose constant source of encouragement and support motivated me throughout my Master's Program.

Finally, to all my friends, who have made my stay at Winnipeg a memorable one, I thank you for your wonderful company.

Contents

List of Figures

Chapter 1

Introduction

Concurrent processes operate on concurrent objects which are shared abstract data types. Processes concurrently operate on these data structures to update or modify them. Concurrent objects play an important role in distributed systems, parallel algorithms, user level thread implementations and multiprocessor operating systems. The correctness of concurrent updates must be ensured.

1.1 Problem and Motivation

In shared memory systems concurrent objects are represented by data structures and algorithms are developed to operate on these data structures. These algorithms modify/update the data structures atomically, thus ensuring consistency of the data structure. Conventionally the consistency of the data structure is maintained by *mutual exclusion*. Mutual exclusion is a way of making sure that if a process is accessing shared memory, a shared data structure, or a shared resource, other processes are prevented from accessing them until the process holding the shared resource relinquishes it.

Mutual exclusion is generally implemented by "locking". In locking, a lock variable is associated with the shared resource. Before any process accesses the shared resource, it tries to obtain the lock. If the particular shared resource is currently not being used by any other process, the requesting process locks the resource, thus obtaining exclusive access to

it. The process then performs the necessary tasks. On completion it releases the lock, so that other processes waiting for the resource can access it. Mutual exclusion by locking is popular because it is simple and well understood. Although it has these advantages, it has inherent weaknesses which degrade the performance of the entire system. Processes access the shared resource in a serial manner as only one process can have access to the shared resource at a time. This may significantly increase the latency of certain processes as they may have to wait a long time to gain access to the shared resource. If the number of processes is large, then the latency may increase further. Furthermore while accessing the shared resources, the processes could also be delayed due to page faults, multitasking preemption, and memory access latency. Locking can also result in other undesirable effects including:

- *Priority Inversion:* Occurs when low priority processes block high priority processes indefinitely, because of indirect priority constraints. For example, consider two processes, H with higher priority and L, with lower priority. If L is in its critical region and H has been scheduled to run, then H will busy wait until L comes out of its critical region. Worse, in such a situation L may continue to be in its critical region if it cannot be scheduled while H is running. Thus H may be blocked indefinitely.

- *Convoying:* This happens when a process holding a lock is de-scheduled. The process might be de-scheduled if it has exceeded its time quantum, or if it encountered a page fault, or due to external interrupts. When such an interrupt occurs other processes capable of running will be unable to progress as the blocked process still holds the lock. Thus, there is a "convoy" of processes waiting on the blocked process holding the lock.

- *Deadlock:* This may occur if processes attempt to lock the same set of objects in different orders. Specifically there are four necessary conditions for deadlock to occur.

They are:

1. *Mutual exclusion condition:* Each resource is either currently assigned to exactly one process or is available.
2. *Hold and wait condition:* Processes continue to hold resources granted earlier while requesting new resources.
3. *No preemption condition:* Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. *Circular wait condition:* There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All the four conditions must be satisfied for a deadlock to occur. If one of them fails, deadlock is not possible.

An alternative to mutual exclusion is to implement concurrent objects using *lock-free* data structures. Operations on lock-free data structures do not require exclusive access, and are thus immune to the problems associated with mutual exclusion caused by process failure and slowdown. A shared data structure is lock-free if its operations do not require mutual exclusion. The algorithms operating on these lock-free data structures ensure that the data structure is accessible to all processes. An inactive process (temporarily or permanently suspended) cannot render the data structure inaccessible. The remaining processes will be able to concurrently operate on the data structure.

Lock-free algorithms for concurrent data structures are implemented using certain synchronization primitives which are provided by the hardware or by the operating system. Synchronization primitives that are commonly available include *test & set*, *fetch & add*, *load-linked & store-conditional* and *compare & swap* (CAS). The first two synchronization primitives cannot be used to implement non-blocking algorithms because it does not

provide the required atomicity in terms of the number of operations it can perform. The one that has been commonly used for non-blocking algorithms is CAS and variants on it. Non-blocking algorithms may also be implemented with the *load-linked & store-conditional* instructions.

A number of lock-free algorithms for shared data structures have been proposed. A detailed discussion of such algorithms is provided in the next chapter. The algorithms that have been proposed to date have drawbacks with respect to performance and memory usage. Some of the algorithms restrict concurrency to reading, so the modifications to the shared data structure must be done in a serial manner. Other algorithms allow processes to concurrently read and update the data structure. Inspite of providing increased concurrency the performance of these algorithms is often affected because of excessive traversal time. This significantly increases the latency of the processes. Another major drawback of some of the proposed lock-free data structures is that they require a large amount of memory.

In this thesis we will design improved lock-free linked lists and develop algorithms to operate on them. Processes operating on these linked lists will be able to concurrently read and update the linked list. The main goal will be to substantially reduce the latency of the processes by significantly decreasing the list traversal time. Another objective is to decrease memory usage. Achieving the objectives mentioned above will depend significantly on the appropriate choice and effective use of proper synchronization primitives.

1.2 Computational Model

The computational model assumed in this thesis is an idealized shared memory multiprocessor as shown in Figure 1.1. A number of processes share a random access memory. The processes may reside on a single processor or may be distributed across different processing elements. We assume that the shared memory is uniformly accessible by all the processes. The performance of the algorithms for lock-free linked lists developed in this thesis will not

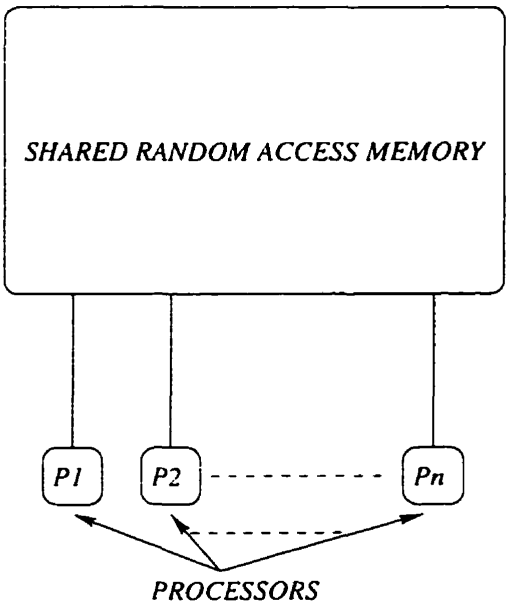


Figure 1.1: A multiprocessor.

take into account specific architectural details such as network delay.

In this thesis we achieve the objectives related to performance and memory utilization. A proof of correctness of the presented algorithms operating on the linked lists is also provided. The performance of our algorithm is compared with other existing algorithms and the results are evaluated.

The rest of this thesis is organized as follows. Chapter 2 provides background and discusses the work already done in this area. The problems in designing lock-free linked lists are elaborated in Chapter 3. The proposed design of lock-free linked lists and an explanation of the algorithms operating on the linked lists is provided in Chapter 4. The correctness of the algorithms is shown in Chapter 5. The simulation of the algorithms along with an analysis of the results is discussed in Chapter 6. Finally conclusions and future work are given in Chapter 7.

Chapter 2

Background and Related Work

This chapter surveys work related to lock-free data structures. The different types of lock-free data structures discussed in the literature are queues, linked lists, and binary trees to name a few.

The choice of a proper synchronization primitive is very important in designing and developing lock-free data structures and algorithms. There are a large number of synchronization primitives available. These can be divided into two classes. The first class of synchronization primitives include those that cannot be used to implement lock-free or wait-free implementations of most common abstract data types such as stacks, queues, and lists. A concurrent object implementation is said to be lock-free if some process must complete an operation after the system as a whole takes a finite number of steps, and it is wait-free if each process must complete an operation after a finite number of steps. Herlihy, in his seminal paper, [7] proves that such primitives cannot be used to develop non-blocking algorithms. The second class of primitives are those that can be used to develop non-blocking algorithms. Section 2.1 discusses the different types of synchronization primitives available. In section 2.2 "universal" methods that are applicable for all kinds of lock-free data structures are discussed. In section 2.3 specific algorithms for lock-free linked lists are reviewed while section 2.4 covers lock-free queues.

2.1 Synchronization Primitives

Mutual exclusion is needed when a shared memory location or shared data structure is accessed by a number of processes. That part of the program which accesses or modifies the shared variable/data structure is called the critical section. It is necessary to ensure that no two processes are in their "critical sections" at the same time. This is to maintain the consistency of the shared variable/data structure. This is achieved by using synchronization primitives. The synchronization primitives are of two types namely, lock-based and lock-free synchronization primitives and these are discussed in the following sections.

2.1.1 Lock-Based Synchronization Primitives

Lock-based synchronization primitives ensure mutual exclusion. Some of the common synchronization primitives are *read/write*, *test-and-set*, and *fetch-and-add*. A survey of synchronization primitives is provided in [4].

Read/write is the simplest of the synchronization primitives, and is included in almost all architectures. The *read* operation copies a value from the shared memory to the local memory of the process, while the *write* operation copies a value from local memory to the shared memory. With extensive software support Reads & Writes can be used to ensure synchronization albeit rather inefficiently.

Test-and-set returns the contents of the shared variable and then stores a specific (normally non-zero) value in the shared variable. The operation of reading the variable and returning it is done atomically. The CPU executing the *test-and-set* instruction locks the memory bus to prohibit other CPUs from accessing the shared variable until the operation is completed [21].

Fetch-and-add is a more complicated operation than *test-and-set*. In this operation a specified variable (suppose it is variable x) is read and a new value ϕ (x) is computed and written back. *Fetch-and-add* is a generalization of *Test-and-set*. It is quite flexible and can

be used for semaphores, highly concurrent objects and also for database synchronization [19].

Transactional Memory is another synchronization primitive which is a general form of the *Load-Linked* and *Store-Conditional* operation. This operation can also be used for implementing lock-free data structures. Herlihy and Moss proposed *transactional memory* [14], which generalizes the idea to multiple shared memory locations at once. Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory. A number of lock-free algorithms [11] have been proposed based on this concept.

2.1.2 Lock-Free Synchronization Primitives

The commonly used synchronization primitives to implement lock-free linked lists are *Compare & Swap*, *Double Compare & Swap*, and *Load-Linked & Store-Conditional*. These primitives are generally referred to as universal primitives. A synchronization primitive is "universal" if it provides a non-blocking implementation of any abstract data type.

2.1.3 Compare and Swap

Compare & Swap (CAS) is a three operand atomic instruction of the form CAS(A,B,C). A, B, and C are single word variables. The algorithm for CAS is shown in Figure 2.1. C is a shared variable and A is a private copy of it made earlier by a process. B is a value which the process is attempting to put in C. It is allowed to do so only if C has not been modified by some other process since this process made a copy of it. If C has not changed since it was last read (i.e. C=A), the new value B is put into C, else CAS returns FALSE, indicating that the process has failed. The CAS algorithm shown in Figure 2.1 assumes all operations execute indivisibly. The CAS algorithm as presented has a problem generally referred to as the A-B-A problem. The A-B-A problem occurs when the value of C is the

same as A even though C has been modified a number of times since the process made a copy of it in A. In such a situation a CAS operation succeeds as the value of C is the same since it was last read. To prevent this error a modified form of CAS called the Double Compare and Swap (DCAS) can be used.

```
COMPARE & SWAP(A, B, C)
BEGIN ATOMIC
    If A = B
        A = C
        return TRUE
    else
        return FALSE
END ATOMIC
```

Figure 2.1: The COMPARE & SWAP synchronization primitive

The DCAS algorithm is shown in Figure 2.2. There are two types of DCAS operations and they are variations of the basic *Compare & Swap* operation. There is a Double-word version, which operates on two adjacent words in memory simultaneously,and a two word version, which operates on two arbitrary words in memory simultaneously. A generic DCAS operation is of the form DCAS(A1, A2, B1, B2, N1, N2). DCAS atomically updates locations A1 and A2 to values N1 and N2 respectively if A1 holds value B1 and A2 holds B2 when the operation is invoked else it returns a false value indicating that the operation failed.

The A-B-A problem associated with a CAS operation occurs when C is the same as A even though C has been modified a number of times since the process made a copy of it. A CAS operation performed in such a case will succeed which can cause errors to occur in many concurrent implementations of objects. In the case of a DCAS operation, A1 contains the value of the variable read, while A2 contains a version number (also referred to as the counter). Whenever a variable is updated the corresponding version number is incremented

```
DCAS(A1, A2, B1, B2, N1, N2)
BEGIN ATOMIC
      If ((A1 == B1) && (A2 == B2)) {
          A1 = N1; A2 = N2;
          return (TRUE) }
      else
          return (FALSE)
END ATOMIC
```

Figure 2.2: The DOUBLE COMPARE & SWAP synchronization primitive.

by one. While updating it reads the current values of the variable last read along with it's version number to ensure that the value has not changed. If some other process had updated the value of the variable by the same value, the current DCAS operation fails, as the version number would have been incremented. Thus the use of version numbers allows us to avoid the A-B-A problem.

2.1.4 Load-Linked and Store-Conditional

The *Load-Linked* and *Store-Conditional* instruction pair are primitives that may be used to implement atomic Read-Write-Modify operations for cached memory locations. The *Load-Linked* instruction operates like a read or load instruction. When it reads a memory location it sets a variable associated with that particular memory location. Any subsequent write operation to that memory location will reset the variable associated with that memory location. The *Store-Conditional* instruction operates like a write operation. This write operation by the instruction *Store-Conditional* will be successful only if the variable has been set otherwise it fails. This synchronization primitive can be used to perform operations similar to other synchronization primitives like the CAS and the DCAS synchronization primitives.

2.2 Universal Methods

Universal methods are general methods for manipulating any lock-free data structure. Lamport [12] gave the first lock-free algorithm for the problem of a single-writer/multiple-reader shared variable. The algorithm was implemented using different synchronization primitives. The commonly used synchronization primitives those just discussed.

2.2.1 Logging Method

Herlihy [8] presented the first universal method which we shall refer to as the logging method. The idea is related to that of keeping a write-ahead log in a transaction processing system [3]: the operations share a log of the operations that they execute on the object, and use the log to determine the object's current state. Operations are assigned a strict serial ordering when they are added to the log, thus ensuring that the object will appear to be updated in an atomic manner.

The log is maintained as a linked list. New operations are added to the log by linking a new item onto the list. Determining the current state of the object merely requires traversing the list to reconstruct the current (from the operation's point of view) state of the object. This reconstruction is done using a sequential version of the algorithm on a private copy of the object. Once the new value of the object has been computed for a particular operation, it can be written into the linked list, "check-pointing" the log, so that future operations do not have to reconstruct much of the state of the object.

Plotkin [17] devised a new object called the *sticky-bit*, to transform any serial implementation into a wait-free atomic one. This is used to solve the the consensus problem (The consensus problem is when the processes are reaching a consensus/agreement on which process will link its item on the list next). An atomic sticky bit (ASB) is a register which can hold 0, 1, or "undefined". If several processes are concurrently trying to write into the same ASB, only one of them succeeds. A processor returns "fail" if the value it was

trying to write disagrees with the already written value, and "success" otherwise. ASB is a 3-valued register that supports a restricted variant of an atomic Read-Modify-Write (RMW).

2.2.2 Copying Method

Herlihy [9] also proposed a method for constructing lock-free data structures by "copying". The data structure's representation and operations are written as stylized sequential programs with no explicit synchronization. Each sequential operation is then automatically transformed into a lock-free operation. The algorithms use the *Load-linked* and *Store-Conditional* instruction pair.

The basic methodology for performing operations on the data structure is by changing a pointer to the structure. A process that wants to perform an operation begins by making a private copy of the data structure using the *Load-Linked* instruction. Changes are made to the private copy of the data structure and it tries to replace the old pointer with the pointer to it's private copy, using the *Store-Conditional* instruction to verify whether the pointer has changed. If the pointer has changed, the replacement fails and the process starts the entire operation again. In general, the copying process involves copying the *entire* data structure. The programmer can specify ways to reduce the amount of copying in certain situations. For example, if only the first element in a singly linked list is to be changed, then only the first element needs to be copied. After performing the necessary changes to the element that was copied, the changed element can use the old element's next pointer to logically copy the rest of the list without actually doing the work.

This method can be very expensive as it requires copying the entire data structure in most circumstances. Also in this method concurrent updates are not allowed. Only one process that is trying to update the structure is actually doing useful work at a given time. Therefore, throughput better than the original sequential implementation can never be

achieved.

2.2.3 Caching Method

Barnes [2] proposes lock-free data structures and algorithms for shared memory multiprocessors using caching. In caching the process takes a private copy of the required part of the data structure. The *Store-Conditional* and *Load-Linked* instruction pair are used to develop these algorithms. Caching is combined with cooperation among processes to complete each other's operations. These algorithms work for various kinds of data structures with minimal modifications.

The caching method works along with the cooperative technique to generate lock-free caching algorithms for any data structure. In this method the operation is first performed on a cached version of the structure. Private copies of the cells are made by using the *Load-Linked* instruction. The cells in the cached version are then validated to ensure that they have not changed since they were copied. If they have changed then the process is aborted and restarted again. If the cells have been validated then the corresponding cells are claimed in ascending order. The cells are claimed in ascending order to avoid live-locks. If any of the cells have changed since they were last claimed the entire process is aborted and started again, else the required operations are performed and then claims to all the cells are released. This method is costly in terms of memory requirements and time, because of the requirements of cache and the transfer of data to and from cache.

Cooperative completion is used to handle interference between processes when operations are being performed, apart from maintaining the consistency of the data structure. The cooperative technique also increases concurrency. If a process P_1 wants to change some part of the data structure, it first checks whether another process P_2 was working on the same part of the data structure. If so, P_1 reads P_2's information and cooperates to complete P_2's operation. The operation from P_2 to be performed on each cell of a data structure

is pointed to by an *"opptr"* field of each cell. The *"opptr"* field contains the address of an *"opdesc"* which describes the operation that has to be performed.

The caching method performs better than Herlihy's copying method because in Herlihy's method, it is necessary to copy the *entire* data structure. Barnes' method also allows concurrent updates where as Herlihy's method does not. The caching method, however does have significant overhead which includes copying the required cells to a local cache, validating, and claiming the cells in a particular order.

2.3 Lock-free Linked Lists

Specific algorithms have been developed for manipulating lock-free linked lists. Algorithms developed specifically for a particular type of lock-free data structure perform better and have less overhead than the universal methods that operate on many structures. The main operations performed on linked lists are inserting, deleting, and updating elements. Some of the algorithms proposed allow more than one process to concurrently modify the linked list while others allow only one process to modify the linked list at a time and concurrency is limited to reading elements of the linked list.

2.3.1 Valois's Design

In Valois' design [9] every node in the linked list has an extra node, called the auxiliary node. The auxiliary node is used to maintain the consistency and continuity of the linked list when processes are concurrently operating on the linked list. The auxiliary node consists of only a pointer field, that points to the next node in the list. An empty linked list consists of two dummy nodes, called first and last with an auxiliary node between them. The structure of the linked list is shown in Figure 2.3.

Operations on linked-lists are performed only after the necessary pointers are read. The pointers are read into a structure referred to as a *cursor*. The first pointer in the cursor,

target, points to the cell the cursor is currently visiting. The second pointer *pre_aux*, points to the auxiliary cell immediately preceding *target* and the third pointer *pre_cell* points to the last normal cell. The CAS instruction is used to perform insertion and deletion operations atomically.

A process that wants to insert a node (a node is inserted along with an auxiliary node) reads the required pointers into the structure, *cursor*. A node can be inserted only between an auxiliary node and a normal cell. Figure 2.4 shows a new node being inserted between an auxiliary node and a normal node.

Suppose that a process wants to insert a node named *NEW_NODE* with an auxiliary node *new.aux* between nodes B and C. The process uses the cursor C that maintains the pointers, *c.pre_aux* and *c.target* to perform the insert operation. The node, *NEW_NODE* is initialized with the new data value. The *new* node is made to point to *new_aux* and the next field of *new_aux* is made to point to *target*

The process then uses CAS to atomically check to ensure that *c.pre_aux* still points to *c.target*. If the pointers have not changed, then *c.pre_aux* is set to point to *new* and a TRUE value is returned to the process. If the pointers have changed since they were last read, the operation fails and a FALSE value is returned to the process which must then restart the operation from the beginning. This ensures the consistency of the linked list during insertion.

When a process wants to delete a node, the *cursor* is passed to the deletion routine. The node to be deleted is the *target* node in the *cursor* structure. The process atomically

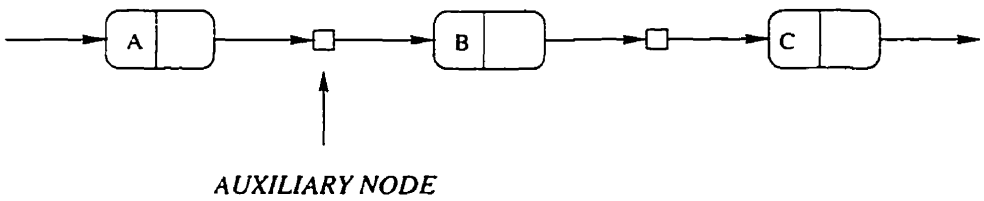


Figure 2.3: Structure of a Valois linked list

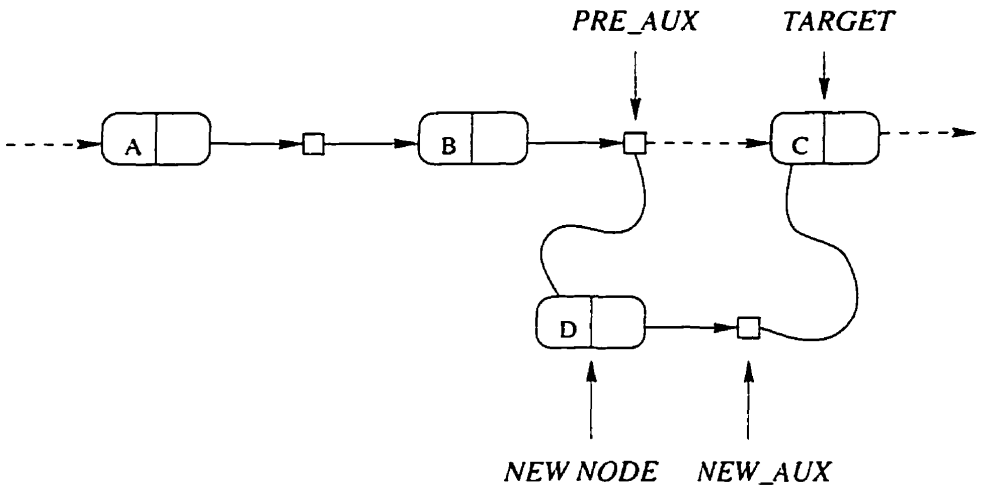


Figure 2.4: Inserting a new cell and auxiliary node

checks if the *cursor* is valid. A cursor is valid if the pointers maintained by the cursor have not changed since they were last read. If the cursor is valid the process deletes *target* and leaves the associated auxiliary cell in the linked list. A TRUE value is also returned to the process informing it that the operation was successful. If the cursor is invalid a FALSE value is returned as the list has been updated and the pointers have changed. In such a case the process has to re-read the pointers in *cursor* and retry the operation.

Successive deletion operations may leave a chain of auxiliary nodes. These chains of auxiliary nodes have to be removed to ensure efficiency of list traversal. To facilitate the removal of auxiliary nodes, each normal cell of the linked list is provided with another link, called *back_link*. This field is used in the following way. Initially it is set to NULL. After a

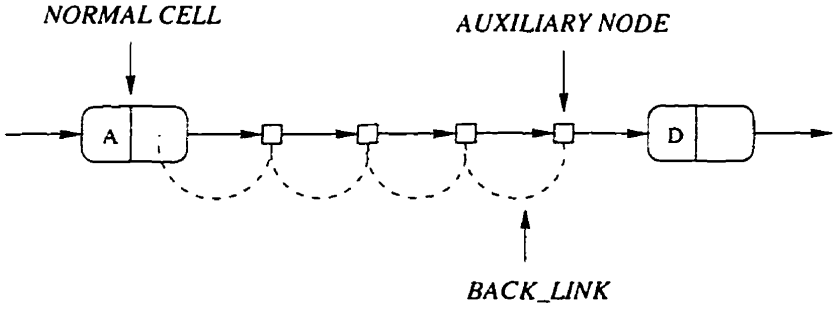


Figure 2.5: Deleting extra auxiliary nodes

cell is deleted. *back_link* is made to point at *pre_cell*. Using these *back_links*, the list may be traversed through the chain of auxiliary nodes until a normal cell is reached as shown in Figure 2.3. The entire chain of auxiliary nodes may then be deleted.

The main advantage of Valois' design is that it allows processes to concurrently traverse the linked list and allows processes to insert or delete nodes anywhere in the linked list. It can also avoid the A-B-A problem by maintaining a count field (representing a version number) in each of the nodes. The major disadvantage of the design is additional overhead, in both time and space. The space overhead is because each node has an auxiliary node associated with it. The time overhead is due to the increase in latency of the processes performing the operations. The increase in latency is due to the following reasons:

- The time required to read the pointers of a node in the linked list, by a process is twice the time compared to reading a linked list that does not have an auxiliary node for every normal node.
- The deletion operation on the linked lists leaves a chain of auxiliary nodes. Processes may spend considerable time in deleting the auxiliary nodes from the linked list.

2.3.2 Greenwald and Cheriton's Design

Greenwald and Cheriton [6] describe a lock-free linked list structure on which processes can concurrently operate. Their design uses the DCAS instruction along with version numbers to avoid the A-B-A problem. The linked list includes a version number which is incremented whenever an operation is performed on the linked list by a process. This design was used in a multiprocessor operating system designed and implemented by them. The algorithms for concurrently deleting and inserting nodes are described in the following section.

During deletion, a process traverses the linked list from the head node until it finds the node that has to be deleted. If the node is not found, it checks the current version number of the list to determine if the list has been changed. If the version number has changed then

the process tries the delete operation again noting the current version number in case a concurrent insert of the desired node has completed. If the version number has not changed and the process has scanned the entire list and the node could not be found, the algorithm returns a NULL value to the process. The NULL value indicates that some other process has already deleted it.

Inserting an element is done by traversing the linked list from the head node till it finds the node after which the new node is to be inserted. The process also notes the current version number of the linked list. At the time of inserting the new node it checks to ensure that the version number has not changed. If the version number has not changed then the new node is inserted otherwise a FALSE value is returned to the process indicating the linked list has been modified since the process last read from the list. On receiving a FALSE value the process restarts the insert operation from the beginning.

When the node is found, the process atomically checks to determine if the version number is unchanged since it was last read by the process, and if the node has not been deleted. If these checks are true, the node is deleted from the list and the version number of the list is incremented to indicate that the list has changed. The other processes which have read the relevant pointers to delete other nodes fail as the version number has changed. These processes have to re-read the list and retry the delete operations.

The main advantage of Greenwald and Cheriton's design is that it is simple and easy to implement. The algorithm also avoids the A-B-A problem by using version numbers. The major disadvantage of this method is the concurrency is limited to just reading and not deleting (writing or updating in general), as each time a process makes changes to the list, the version number is incremented and other processes have to re-read the pointers.

2.3.3 Massalin and Pu's Method

Massalin and Pu [15] also present lock-free algorithms based on the DCAS instruction. In this method nodes are deleted only in "safe" situations. A delete operation is "safe" if the deleted node's link pointers continue to be valid. Deletion of nodes from the middle of the linked list is done in two steps. In the first step, the nodes that are to be deleted are marked and left in the list. In the second step, if the previous node is not marked for deletion, then the original node is deleted.

The main disadvantage of this method is that concurrency is limited to just one process. This increases the latency of the processes operating on the linked lists. Another approach has been proposed where the second step is done the next time the list is traversed. In this approach the traversal time for other processes will increase if the number of nodes that have been marked for deletion are many, but have not been removed from the linked list.

2.4 Lock-free Queues

The main operations to be performed on queues are enqueing and dequeuing. Enqueing involves inserting elements at one end of the queue and dequeuing is the process of deleting elements from the other end. The algorithms developed allow two processes, one for enqueing and the other for dequeuing to proceed simultaneously.

Prakash *et al.* [18] present a method for designing non-blocking algorithms for any concurrent data structure using the *Compare & Swap* operation as the basic synchronization primitive. In their general method, many processes are allowed to concurrently access the data structure, but modifications to the data structure are made in a serial order. To overcome the problem of a slow process leaving the data structure in an intermediate state and blocking fast processes, the algorithm is modified so that the entire state information necessary to complete any operation is globally available. If a fast process finds a situation

where it would normally have to wait for a slow process to complete its operation, the fast process can help it to finish the operation and proceed with its own operation. The main disadvantage is that concurrency is restricted to just one process while the data structure is being manipulated. Apart from this, as the processes help each other to complete the operations their latency also increases. This overhead is absent if the processes do not assist other processes in completing their operations.

Valois [22] presents a list-based, non-blocking algorithm which provides concurrency by keeping a dummy node at the head (dequeue end) of a singly-linked list, thus simplifying the special cases associated with empty and single-item queues. In this method the tail pointer is allowed to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or reusing dequeued nodes(A situation is said to be "safe" if operations can be performed without breaking the list). If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. To avoid this problem, Valois suggests associating a reference counter with each node. A process increments the reference count if it is accessing the node and decreases it when it is not going to access the node any more. Thus, a node is freed only when no pointers in the data structure or temporary variables point to it.

Michael and Scott [16] propose a lock-free queue using the CAS synchronization primitive and a modification counter to avoid the ABA problem. Their algorithm implements the queue structure as a singly linked list having two dummy nodes, a head and a tail. The algorithm does not allow the tail pointer to point to a dequeued node nor any of it's predecessors. It is due to this that the dequeued node can be easily reused. The consistency of the values is maintained by initiating a sequence of read statements that re-checks the earlier values read from the queue to be sure that they have not changed. The re-reads may result in significant overhead.

Lamport [13] also gives a *wait-free* implementation for FIFO queues but restricts the

concurrency to a single enqueuer and a single dequeuer. Gottlieb *et al.* [5] gives an algorithm for enqueueing and dequeuing which provides a high degree of parallelism. This implementation permits concurrent enqueing and dequeuing processes, but it is blocking, since it uses critical sections to synchronize access to individual queue elements. Stone [20] also gives an algorithm for enqueing and dequeuing using the *Compare & Swap* synchronization primitive. In his design, however it is also possible for a slow process to block other processes that are enqueing and dequeuing. Herlihy and Wing [10] give non-blocking algorithms that allows arbitrary numbers of enqueuers and dequeuers to operate on the queues.

Chapter 3

Designing Lock-Free Linked Lists

An important aspect of designing lock-free linked lists is developing efficient algorithms to operate on the linked lists. Efficiency can be improved by minimizing the traversal time and/or by reducing the space overhead. Reducing traversal time decreases the latency of the processes operating on the linked lists. Space occupied by the linked lists can be large, depending on the application, and conserving space under such conditions may be important. A number of problems arise in designing lock-free linked lists and developing non-blocking algorithms to operate on them. This chapter discusses the various design and synchronization problems.

3.1 Design Problem

Design complexity is attributable to the need to maintain the consistency of the linked list while concurrent operations are being performed on it. The major problems occur during concurrent updates to the same part of the list. These include deletion of adjacent nodes, concurrent insertion of nodes between the same pair of nodes, and concurrent insertion and deletion of adjacent nodes.

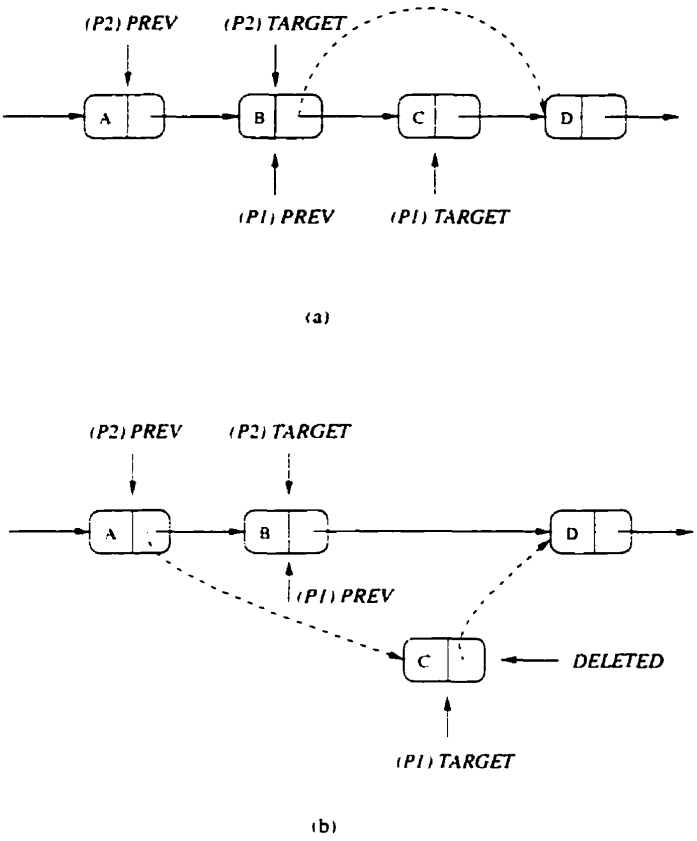


Figure 3.1: Concurrent Deletion of nodes B and C.

3.1.1 Concurrent Deletions

A process trying to delete a node, maintains pointers to at least two nodes, called *target* and *prev*. *Target* is the node to be deleted and *prev* is the node immediately preceding the *target*. The linked list will be in an inconsistent state if adjacent nodes are deleted simultaneously by concurrent processes without concurrency control. An example of the concurrent deletion of adjacent nodes by different processes is shown in Figure 3.1 which illustrates the situation where two processes are concurrently trying to delete nodes which are adjacent to each other. Assume that process 1 is trying to delete node C and process 2 is trying to delete B. Process 1 reads pointers associated with B(prev) and C(target). Process 2 reads pointers A(prev) and B(target) (see Figure 3.1[a]).

Suppose that process 1 is successful in deleting node C. After deleting node C process

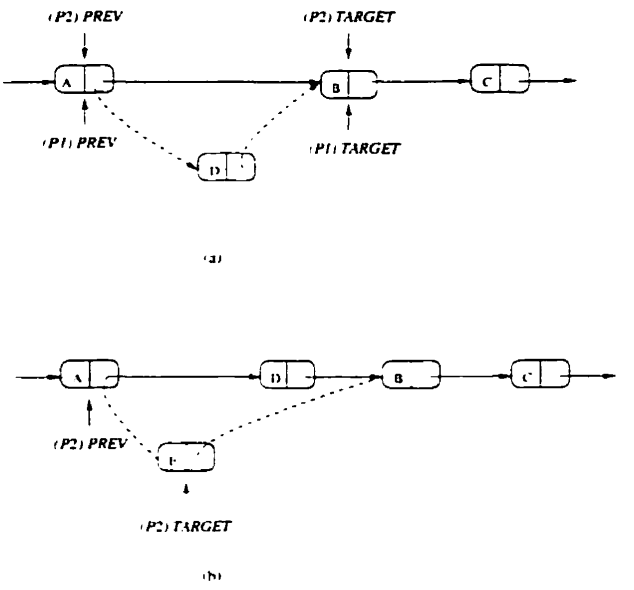


Figure 3.2: Concurrent insertion of nodes D and F.

1 updates the pointers. The updated linked list is shown in Figure 3.1(b). When process 2 tries to delete node B it will swing the pointer field of node A to node C (already deleted) and the linked list will be in an inconsistent state as shown in Figure 3.1(b). The inconsistent state is because the linked list is broken and all the nodes of the linked list are no longer connected.

3.1.2 Concurrent Insertions

Concurrent processes trying to insert nodes between adjacent nodes maintain two pointers, called *prev* and *target* in the same way as the processes trying to delete the nodes. The main problem associated with concurrent insertions arises when two processes try to insert nodes between the same set of adjacent nodes. In such situations, the insert operation of one of the processes is undone.

Figure 3.2 shows two processes trying to concurrently insert nodes between two adjacent nodes. Suppose that Process 1 wants to insert node D between A(prev) and B(target), and process 2 wants to insert node F between A(prev) and B(target). Assume process 1 succeeds

in inserting node D between A and B and updates the pointers as shown in figure 3.2(a). Later if process 2 inserts node F, it will undo the insert operation of process 1 as node A's pointer field will be set to point to F and node F's pointer field will be set to point to B (Figure 3.2(b)), thus eliminating node D from the linked list.

The drawback of the methods suggested above can be overcome by combining operations. In this method, any process that wants to perform an operation checks to see if there are any pending operations of other processes. If such an operation exists it will first complete the operation of the previous process before it proceeds with its own operation.

3.1.3 Concurrent Insertion and Deletion

Insertion of a node and simultaneous deletion of an adjacent node by concurrent processes may also result in an inconsistent linked list. An example of such a situation is shown in Figure 3.3. Suppose that there are two processes, where process 1 is trying to delete node B and process 2 is trying to insert node D, between nodes A and B. Process 1 reads the pointers associated with the node to be deleted as *target* and the node preceding the target node as *prev*. Similarly, process 2 which is trying to insert a node between nodes A and B maintains two pointers, *target* associated with node B and *prev* associated with node A. Concurrent insertion and deletion can take place in one of two possible ways depending on which process completes first. They are:

1. **Deletion followed by Insertion**

   Figure 3.3(a) shows a process deleting an element and another process trying to insert an element just before the element that is being deleted. If process 1 succeeds in deleting node B, B is deleted from the linked list. But the process that has already read the pointers of nodes A and B to insert a node, inserts the node D between A and B. The linked list after these operations is not in a consistent state as shown in

Figure 3.3(b) as node A's pointer points to node D and D points to the deleted node B.

2. **Insertion followed by Deletion**

   In the situation described above, if the process that is inserting an element succeeds first and then the other process performs the deletion operation, the linked list will again be in an inconsistent state. This is shown in Figure 3.4. The initial state of the linked list is shown in Figure 3.4(a). If process 2 succeeds then it will insert the node D between nodes A and B. Then if process 1 deletes node B, it will make A's pointer point to node C, thus undoing the insertion of process 2. This results in an inconsistent linked list with D lost as shown in Figure 3.4(c).

A similar situation to the one shown above could also exist if a new node to be inserted lies between B and C, and node B is being deleted. Assume that process 2 wants to insert a node D between B and C while process 1 wants to delete B. Suppose process 2 succeeds first and inserts D between B and C as shown in Figure 3.5(b). If process 1 then deletes B, the previous node insertion is undone as node A points to node C as shown in Figure 3.5(c) which results in an inconsistent linked list. The problems associated with concurrent insertions and deletions have to be avoided. The general approach to overcome these problems is to allow only one of the concurrent processes to successfully complete its operation. Other processes trying to concurrently delete or insert nodes adjacent to the node whose pointers have changed since they were last read fail. These processes have to re-read the updated pointers and retry the operations.

Another method to resolve the problems stated above is by "operation combining". The basic idea of this technique is that processes "announce" their intention to perform an operation in a globally shared array, and a policy is implemented so that no process can starve. The policy of not allowing a process to starve is called wait-free synchronization.

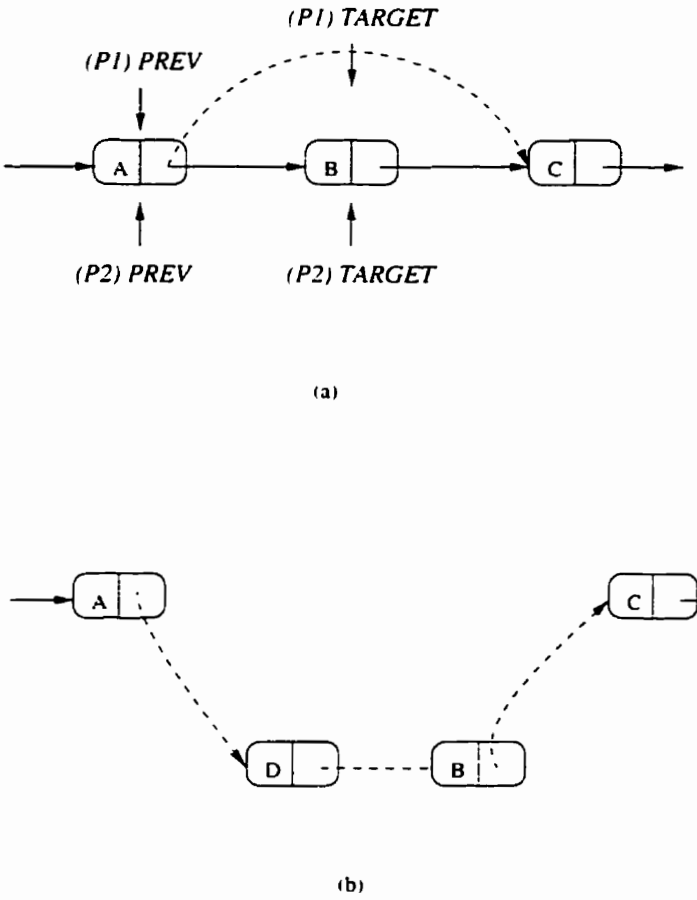


Figure 3.3: Concurrent Insertion and Deletion of nodes.

In this policy before performing its own operation, a process performs any incomplete operations of other outstanding processes. The main drawback of the first method is that some of the processes will have to restart the entire operation again. In the second approach the latency of each of the processes increases because of the time required to perform the operations of the other processes. Apart from this it also results in wasted parallelism as processes can only read concurrently while updates are done in a serial manner.

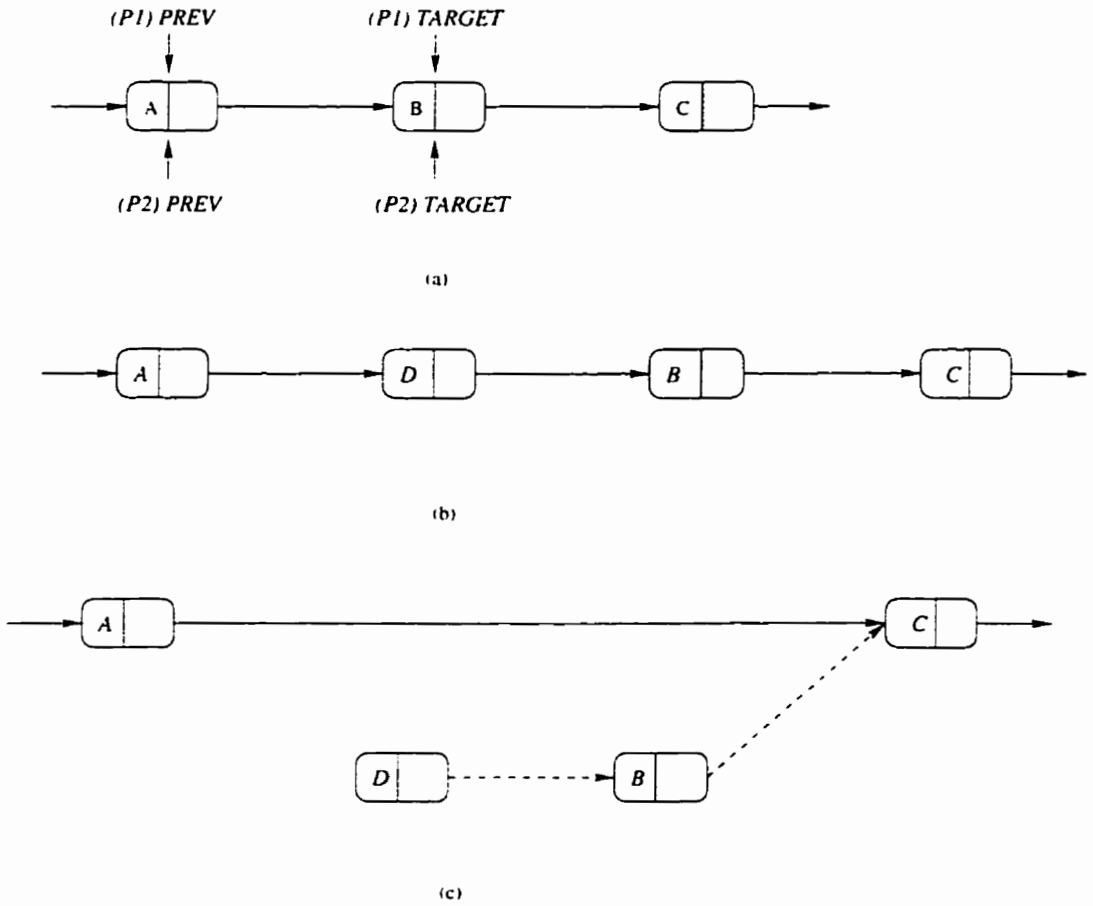


Figure 3.4: Concurrent Insertion and Deletion of nodes.

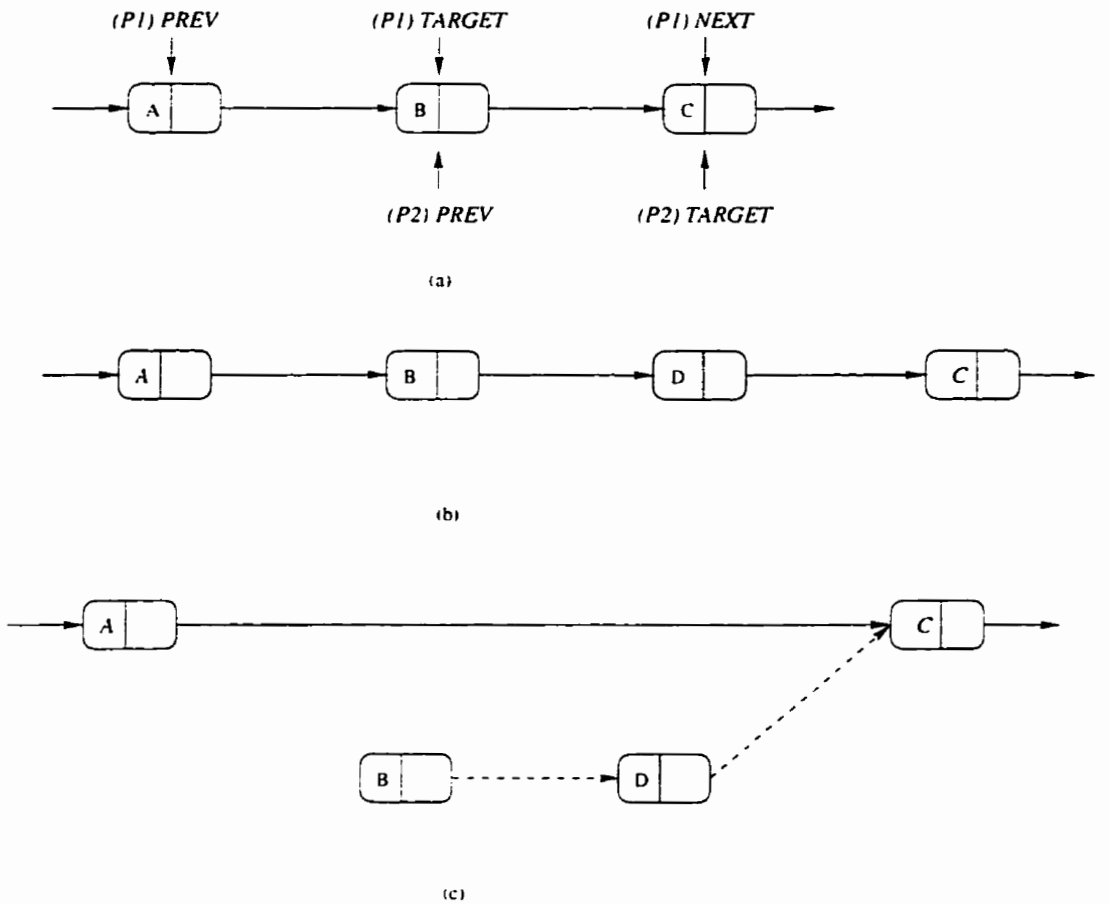


Figure 3.5: Concurrent Insertion and Deletion of nodes.

Chapter 4

Proposed Design

The structure of the linked list in our design includes two dummy nodes, head and tail. The head node points to the first node in the linked list while the tail points to the last node in the linked list. An empty linked list consists of these two dummy nodes pointing to one another.

Each process that wants to manipulate a particular node, the *target*, maintains three pointers, *prev*, *target*, and *next*. *Prev* points to the node preceding the *target* node and *next* points to the node that immediately follows the *target* node. The addresses of these nodes are obtained by traversing the linked list from the head node of the linked list. Maintaining the *prev* pointer will enable improved efficiency in both space and time.

4.1 Structure of the Linked List

Each node of the linked list consists of four fields, a data field, two pointer fields and a counter field. The structure of the linked list is shown in Figure 4.1. One of the pointers is used to maintain an updated linked list so that the pointers do not point to any node that has been deleted from the linked list or the ones that have been marked for deletion, thus maintaining the consistency of the linked list. The other pointer is used to facilitate concurrent processes traversing the linked list even if some of the nodes in the linked list have been marked for deletion. This condition arises when a process is at a particular node

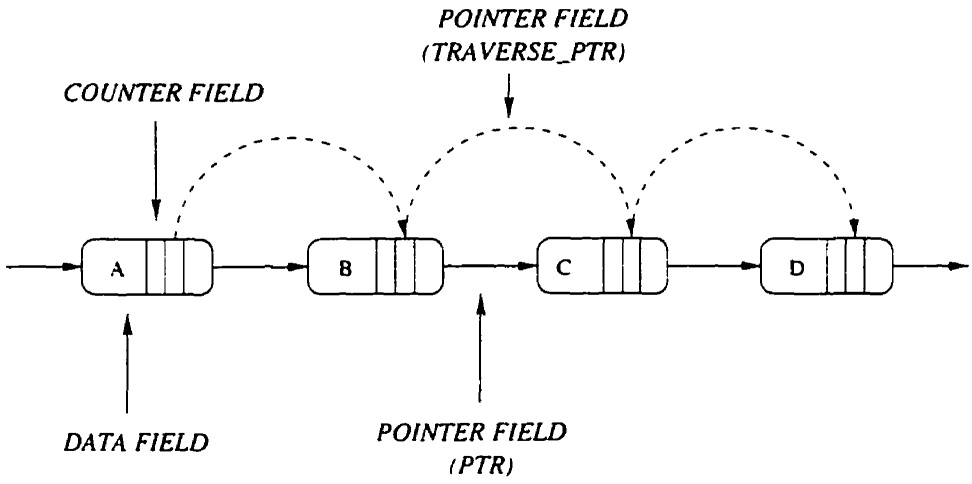


Figure 4.1: Structure of the Linked List.

and another process happens to mark the same node for deletion, by making its pointer field null and making the pointer of *prev* point to *target*. In this situation the first process may still traverse the linked list using the spare pointer thereby increasing concurrency. Allowing delete operations on interior nodes of the list is harder, because a node may be deleted and deallocated while another process is traversing it. If a deleted node is then allocated and reused for some other purpose, its new pointer values may cause invalid memory references by the other processes that are still traversing the list. To avoid this problem a counter is used. Whenever a process is visiting a node it increments the counter field and decrements it when it leaves the node. Therefore any process trying to deallocate a node that is currently being visited by another process, checks the counter field first. If the counter field is zero, the process deallocates it as no process is visiting it. If the counter field is non-zero, then the process defers the deallocation.

4.2 Linked List Traversal

The algorithm for traversing the linked list is shown in Figure 4.2. An empty list consists of two nodes *FIRST* and *LAST*, the pointer field of *FIRST* points to *LAST*. If the list is empty the *FIRST* and *LAST* nodes are returned as *prev* and *target* respectively (lines 2 -

6 of the algorithm). When a process finds the key by traversing the linked list using the *ptr* pointer, it returns pointers to *prev*, *target*, and *next* nodes, the key being in the *target* node (shown in lines 10 - 13 of the algorithm). If the required node is not found, the algorithm returns a false boolean value indicating that the search was unsuccessful. If the *ptr* field of any node is null and the node is currently being visited by another process then the process will use the spare pointer *traverseptr* to traverse the linked list. The process then checks if the data of the node following this node has the *key*. If the process does not have the key, then it decrements the counter field of the current counter node (pointed to by *prev*) by one. On decrementing it, if the counter value is zero, the node is deleted (shown in lines 15 - 21 of the algorithm). It then increments the counter field of the next node, before visiting it. If a process visiting a node has its *ptr* field null, and if the *key* is present in the node that immediately follows it then the process has to re-read the pointers by traversing from the start of the linked list (shown in lines 23 and 24 of the algorithm). This is indicated by returning a "CURSOR_AGAIN" flag to the calling process. "CURSOR_AGAIN" indicates that the specified key is present in the list and the process has to traverse again to find it. Traversal is necessary as the current *prev* pointer of the process points to the node that precedes the key and that node has been marked for deletion. If the *ptr* field is not null and the node immediately following it does not have the *key* then the search is continued (This is shown is lines 28 - 31 where the pointers are advanced along the list).

4.3 Deleting Elements from the Linked List

The problems associated with deleting elements from the linked list concurrently are overcome by using the *Double Compare & Swap* (DCAS) synchronization primitive. Recall that the *Double Compare & swap* synchronization primitive atomically operates on two memory elements. The algorithms for deleting nodes from the linked list namely, TRY_DELETE and DELETE are shown in Figures 4.3 and 4.4 respectively. The DELETE algorithm

```
    Algorithm CURSOR(key)
1.    prev = FIRST;
2.    if (prev → ptr == LAST) /* Checks for empty list */
3.    {
4.          target = LAST;
5.          return (TRUE);
6.    }
7.    while (prev → ptr != LAST) /* traverses till end of the list */
8.    {
            /* checks to see if the node following prev is the required key */
            /* assumes McCarthian AND */
9.        if ((prev → ptr != NULL) && (prev → ptr → data == key)) {
10.             target = prev → ptr;
11.             next = target → ptr;
12.             return TRUE;
13.       }
            /* traverses the list using "traverseptr" is the current node has been marked
               NULL and the node following it is not the required node */
14.       if (( prev → ptr == NULL) && ( prev → traverseptr → data != key)) {
15.             prev → counter--;
16.             temp = prev;
17.             prev → traverseptr → counter++;
18.             prev = prev → traverseptr;
19.             if ((temp → counter == 0) release (temp);
              Continue;
20.       }
            /* checks to see if "prev" points to "key" using traverseptr */
21.       if ((prev → ptr == NULL) && ( prev → traverseptr → data == key)) {
22.             prev →counter--;
23.             if (prev → counter == 0) release (prev);
24.             return (CURSOR_AGAIN);
25.       }
26.       else {
27.             if (prev != FIRST)
28.                 prev → counter--;
29.             prev → ptr → counter++;
30.             prev = prev → ptr;
31.       }
32.   }
33.   return (FALSE);
```

Figure 4.2: Algorithm for Traversing the Lock-Free Linked List.

```
ALGORITHM TRY_DELETE(prev, target, next)
{
1.    r ← DCAS(prev → ptr, target → ptr, target,next, next, NULL);
2.    if (r == TRUE) {
3.       prev → counter−−;
4.       if (target → counter == 0) release (target);
5.    } else {
6.       if ((target → ptr != next) && (target → ptr != NULL)) {
7.          next = target → ptr;
8.          return DELETE_AGAIN;
9.       }
10.       prev → counter−−;
11.       if ((prev → counter == 0) && (prev → ptr == NULL))
12.          release(prev);
13.       return TRAVERSE_AGAIN;
14.    }
15.    return r;
```

Figure 4.3: TRY_DELETE Algorithm.

locates the required target node that has to be deleted and on finding it initiates the TRY_DELETE function. The TRY_DELETE function tries to delete the target node.

In the TRY_DELETE algorithm (Figure 4.3), the DCAS operation checks if the pointers *prev*, *target*, and *next* have changed. If the pointers have not changed, the pointer field of the *target* node is set to NULL, indicating that the node is deleted. The pointer field of node *prev* is then made to point to *next* and TRUE status is returned. If any of the pointers have changed then the DCAS operation returns a FALSE status flag. When a process receives a FALSE status flag from the DCAS operation, it checks to see if it is because the *next* node has changed since it was last read. If the *next* node has changed, then it reads the new next node and updates the pointer field of the *target* node by making it point to the new *next* node. It then returns a status flag "DELETE_AGAIN" indicating to the DELETE function, that the three pointers *prev*, *target*, and *next* have been updated and

```
ALGORITHM DELETE(key)
1.    flag = TRUE:
2.    while (flag == TRUE) {
3.       repeat {
4.          r = CURSOR(key):
5.       } until (r != CURSOR_AGAIN):                  // locate position for deletion
6.       if ( r == FALSE) return FALSE:
7.       repeat {
8.          t = TRY_DELETE(prev.target.next):         // attempt deletion
9.       } until (t != DELETE_AGAIN):
10.       if ( t != CURSOR_AGAIN) flag = FALSE:        // deletion successful
11.   }
12.   return t:
```

Figure 4.4: DELETE Algorithm.

the TRY_DELETE function has to be invoked again to delete the node. If a FALSE value is returned by DCAS it is because the *prev* pointer has changed. A "CURSOR_AGAIN" status is then returned to the DELETE function. On receiving this the DELETE function restarts the entire operation from the beginning. If the FALSE value returned by the DCAS function is because the *target* node has changed, then a status flag "FALSE" is returned indicating that the *target* node has changed.

The DELETE function (Figure 4.4) takes a single parameter, the *key* of the element to be deleted. The node having the value *key* is the node that has to be deleted. The delete function tries to determine if the required *key* exists in the list by initiating the CURSOR function. The cursor function returns one of the three status flags, namely CURSOR_AGAIN, FALSE or TRUE. The CURSOR_AGAIN flag indicates that the pointers have been updated and the *key* has to be searched for again from the start of the list. A FALSE flag indicates that the required *key* was not found. IF a TRUE flag is returned, then the TRY_DELETE function is invoked to delete the node that has the *key*. The TRY_DELETE function returns one of the four status flags namely,

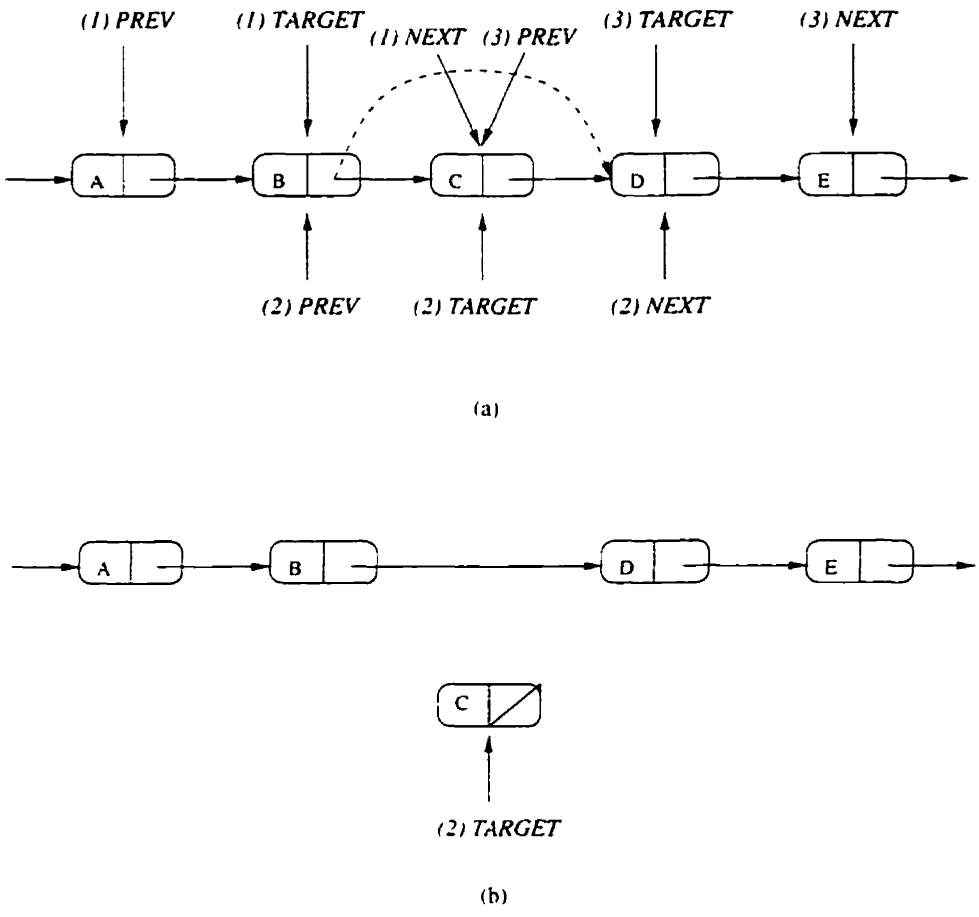


Figure 4.5: Concurrent Deletion of nodes B, C, and D.

DELETE_AGAIN, CURSOR_AGAIN, FALSE or TRUE. The DELETE_AGAIN flag indicates that the TRY_DELETE function has to invoked again, as the pointers have been updated. The CURSOR_AGAIN flag indicates that the *prev* pointer has changed and hence the *key* has to be searched for again from the start of the linked list. A TRUE flag indicates that the required node having the *key* was deleted. A FALSE flag indicates that the required key was not found as it had been deleted by some other process.

Concurrent deletion of nodes is illustrated in Figure 4.5. Consider a situation where three processes are concurrently trying to delete one node each, and the nodes are adjacent to each other. Assume that process 1 is trying to delete B, process 2 is trying to delete C, and process 3 is trying to delete D all concurrently. Process 1 reads pointers associated with

nodes A(prev), B(target) and C(next). Process 2 reads B(prev), C(target), and D(next). Process 3 reads C(prev), D(target), and E(next) (see Figure 4.5(a)). Suppose that process 2 is the first one to be successful in deleting node C. After deleting C it updates the relevant pointers using DCAS. The updated linked list is shown in Figure 4.5(b). If process 1 now tries to delete node B it fails because B(*target.ptr*) no longer points to C. This process will have to re-read the current value of B (*target.ptr*), assign it to *next*, and retry the DELETE operation. This is shown in lines 4 and 5 of the TRY_DELETE algorithm. Process 1 does not have to traverse the linked list to read the pointers again. This results in a substantial gain in performance for this process compared to other existing algorithms. This is possible only if a process is concurrently deleting a node which precedes the node that has already been deleted. If process 3 now tries to delete node D it fails as D's previous node (*prev.ptr*) C points to null. As a result process 3 will have to read the updated pointers by re-traversing the list from the *head* node.

The process which successfully deletes a node is responsible for deallocating the memory associated with that node.

4.4 Inserting Elements in The Linked List

The algorithm for inserting elements in the linked list is straightforward. The algorithm uses the *Compare & swap*(CAS) synchronization primitive. CAS returns TRUE if the operation was successful otherwise it returns FALSE. The algorithm consists of two parts. TRY_INSERT and INSERT. The new node is already initialized to contain the new data value. The TRY_INSERT algorithm tries to insert the new node immediately before the node that has the data value "key" in its data field. The pointer field of the new node is made to point to *target*. Then the CAS operation checks if *prev.ptr* still points to target. If it does then *prev's* pointer field is made to point to the new node and TRUE is returned to the INSERT algorithm. If any of the pointers *prev* or *target* have changed then the

CAS operation returns a FALSE flag. The process then checks to determine if the *target* pointer has changed since it was last read. If so it updates its *target* pointer and returns a RETRY flag to the INSERT algorithm indicating that the pointers have been updated and the TRY_INSERT operation has to be invoked again. The algorithm returns a FALSE flag if the CAS operation fails because the *prev* pointer has changed.

The INSERT algorithm searches for a node with the desired key value (called the *target* node) before which the new node has to be inserted. Valois' and Greenwald also adopt the same approach for inserting nodes into the linked lists. This is done by traversing the linked list using the CURSOR function. The CURSOR function returns a FALSE value if the desired node is not found. If the node is found it returns a TRUE value along with three pointers *prev*, *target*, and *next*. If the node with the particular key is found TRY_INSERT invokes the TRY_INSERT function to insert the node before the *target* node. If it receives a RETRY flag from the TRY_INSERT function it invokes the TRY_INSERT function again as the pointers have been updated. In our algorithm, in case of duplicate values we choose the first duplicate value we encounter. For example, in deletion, if a process wants to delete a node with a particular key, the first node in the linked list whose data value matches the "key" is deleted. If there are duplicate "key" values in the list those are not deleted. For insertion process, suppose that a process wants to insert a new node immediately before a node that has a certain "key". The insertion algorithm will insert the new node immediately before the first "key" it finds in the linked list.

```
ALGORITHM TRY_INSERT(prev, target, value)
1.    {
2.     node = new_node();
3.     node → data = value:
4.     node → counter = 0:
5.     node → ptr = target:
6.     node → traverse_ptr = target:
7.     r = CAS(prev → ptr. target. node):          /* try to insert the node */
8.     if ( r == FALSE) {
9.           release (node):
10.           if (prev → ptr != NULL) {          /* Has target been marked for deletion */
11.              target = prev → ptr:             /* Get the address of the new target */
12.              return RETRY:
13.        }
14.     }
15.     prev → counter−−:
16.     return r:
17.   }
```

Figure 4.6: TRY_INSERT Algorithm.

```
ALGORITHM INSERT(key, value)
1.     repeat {
2.     r = CURSOR(key):
3.     } until ( r != CURSOR_AGAIN):
4.     if ( r == TRUE)
5.     repeat {
6.          t = TRY_INSERT(prev. target. value):
7.     until (t != RETRY):
8.     }
9.     return (t):
```

Figure 4.7: INSERT Algorithm.

Chapter 5

Correctness

The basis for proving the correctness of the algorithms for inserting and deleting nodes from the linked list are the behaviors of the CAS and DCAS instructions. The following theorems and lemmas establish the correctness of the presented algorithms.

5.1 The Deletion Case

Lemma 1: The DCAS operation ensures that a process fails if and only if any of the nodes or pointer fields of the nodes, *prev* and *target* were changed or deleted since they were last read by the process executing the DCAS.

The DCAS instruction performs the following sequence of operations atomically. It checks if *prev.ptr* points to *target* and *target.ptr* points to *next*. If the pointers *prev.ptr* and *target.ptr* have changed since they were last read, the DCAS operation fails.

The changes in the pointers must be due to one of the following reasons:

- One or more of the three nodes, *prev*, *target* and *next* could have been deleted.
- New nodes could have been inserted between *prev* and *target* or *target* and *next*.
- The pointer fields of any of the three nodes could be NULL.

□

Lemma 2: The DCAS operation guarantees that, if none of the nodes (*prev*, *target*, and *next*) or the pointer fields of the nodes *prev* and *target* have changed since they were last read, then *target* will be deleted.

The DCAS instruction atomically checks if *prev.ptr* points to *target* and *target.ptr* points to *next*, to ensure that the pointers have not changed since they were last read. If the checks are true, *prev.ptr* is made to point to *next*, thus eliminating *target* from the linked list. The pointer field of *target* is set to NULL to indicate that *target* is deleted. □

Lemma 3: Processes operating on disjoint sets of nodes (*prev*, *target*, and *next*) concurrently, successfully delete their respective *target* nodes and the consistency of the linked list is also maintained, if none of the nodes *prev*, *target*, and *next* have been deleted and the pointers of *prev* and *target* have not been changed since they were last read.

Each process reads three nodes *prev*, *target*, and *next* to perform the delete operation. These nodes for each of the processes are different for every process, the node to be deleted, *target*, lies between *prev* and *next*. Since the pointers to the nodes *prev* and *target* have not changed, the *target* node of each of the processes is deleted. This is possible because of *Lemma 2*. When the *target* node of each of the processes is deleted, for each of the processes, *prev.ptr* is changed to point to *next* and *target.ptr* is set to NULL. This eliminates all the *target* nodes from the linked list. This update of pointers ensures that the linked list is left in a consistent state. □

The example in Figure 5.1(a) shows two processes, with the three pointers each, *prev*, *target*, and *next*. Process 1's target node is B and process 2's target node is E. After deletion of the two target nodes the linked list is in a consistent state as shown in Figure 5.1(b). The order of deletion of the nodes does not have any effect on the consistency of the linked list.

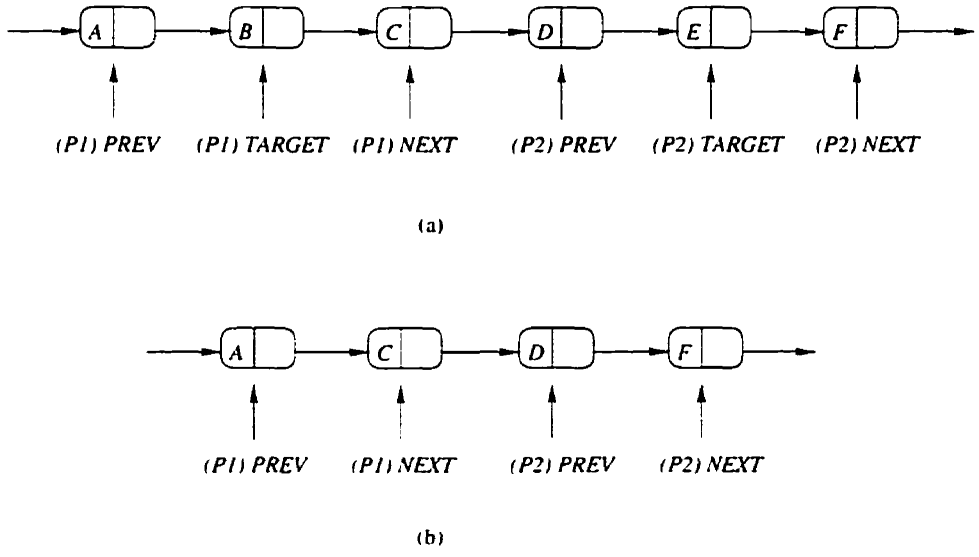


Figure 5.1: Processes operating on different set of nodes.

Lemma 4: When two or more processes are trying to delete the same node, then only one process succeeds in deleting the node while the others fail.

When two or more processes are trying to delete the same node, then the *prev*, *target*, and *next* pointers of all the nodes will be the same. In such a situation only one process will be successful in deleting the *target* node. The process that succeeds is the one that finds the pointers of *prev*, *target*, and *next* unchanged. The other processes trying to delete the same node fail, as atleast one of the pointers *prev*, *target*, and *next* would have changed. □

Lemma 5: When processes try to delete adjacent nodes, one process succeeds for each node deleted (some of the processes succeed while other processes fail) and the linked list is left in a consistent state.

Consider a situation where two processes are trying to delete a node each, and these nodes are adjacent to each other. Suppose that the first process succeeds in deleting the *target* node from the list (via *Lemma 2*) then the second process must fail due to *Lemma 1*. The reason for the failure of the second process may depend on the position of it's *target* node with respect to the node that was deleted by the first process.

- The first situation is when the *target* node of the second process precedes the node that is deleted by the first process. The DCAS instruction of the second process under these conditions fails as the pointer field *target.ptr* will not be pointing to *next* which it had read before. This is because the second process's *next* node is the first process's *target* node which was deleted.

- The second situation is when the second process is trying to delete it's *target* node. that immediately follows the node that has been deleted by the first process. In such a situation the second process fails because it's *prev* node is the first process's *target* node and *target.ptr* has been set to NULL by the first process. This will result in the failure of the DCAS operation (*Lemma 1*) of the second process as it's *prev.ptr* no longer points it's *target* node. (its *prev.ptr* has been set to NULL.)

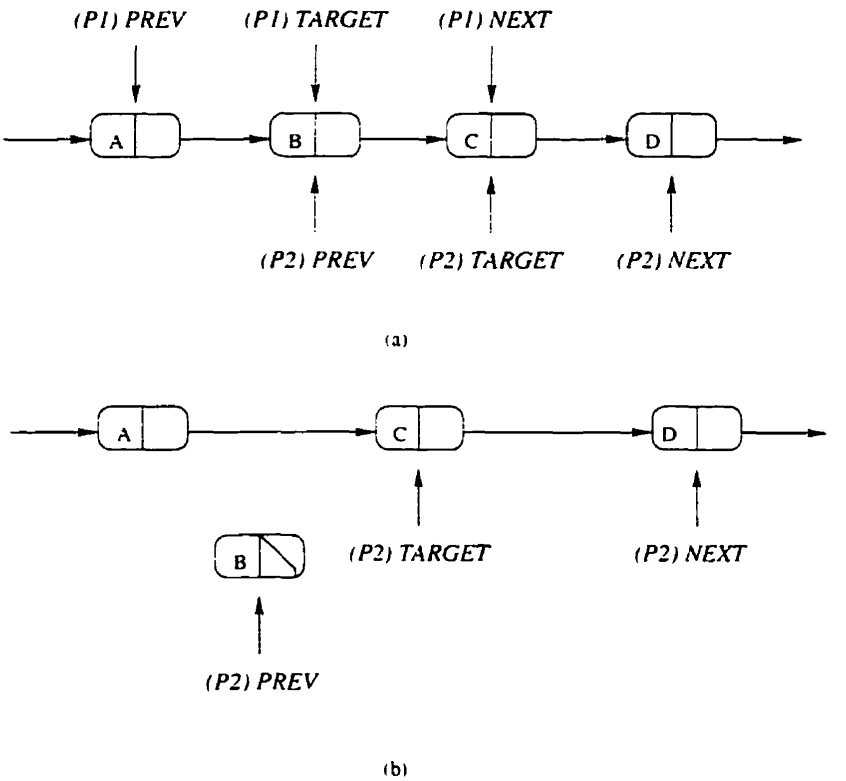


Figure 5.2: Processes trying to delete adjacent nodes.

Similarly when more than two processes are trying to delete nodes that are adjacent to each other, some of the processes succeed in deleting their *target* nodes while the other processes fail. The processes that succeed are those whose pointers *prev.ptr* and *target.ptr* have not changed and the nodes *prev*, *target* and *next* have not been deleted as in this situation the DCAS operation succeeds and the *target* node is deleted (*Lemma2*). The processes which fail are those whose pointers have changed due to the deletion of the adjacent node or whose pointers have changed (as described in the previous paragraphs). □

The above lemma is illustrated with an example in Figure 5.2. Figure 5.2(a) shows two processes, which are trying to delete adjacent nodes. Process 1 is trying to delete node B while process 2 is trying to delete node C. Suppose that process 1 is successful, then it sets B's pointer field to NULL. Later if process 2 tries to delete C, it fails as the DCAS operation fails because the *prev* node's pointer does not point to *target* (node C) any longer (it is null). This is shown in Figure 5.2(b).

Theorem 1: When one or more processes try to delete nodes then exactly one process succeeds in deleting each node and the consistency of the linked list is also maintained.

The different situations in which processes could attempt deletion are

1. If processes are operating on disjoint sets of nodes, then each process deletes its respective target node because of *Lemma 3*.

2. If two or more processes are trying to delete the same node, then only one process succeeds in deleting the node while others fail because of *Lemma 4*.

3. When processes are trying to delete nodes that are adjacent to each other then one process succeeds for each node deleted. Furthermore some processes succeed while others fail while maintaining the consistency of the linked list due to *Lemma 5*.

The above cases show all the different possibilities in which processes delete the nodes while maintaining the consistency of the linked list.

□

5.2 The Insertion Case

This section shows the different situations in which processes try to insert nodes into the linked list. The lemmas and theorem prove that in all the cases the consistency of the linked list is maintained. They also determine the conditions where certain processes succeed in inserting nodes, while others fail.

Lemma 6: The CAS instruction ensures that the operation of inserting a node fails if any of the nodes (*prev* and *target*) have been deleted or the pointer field of the node *prev* has been changed since it was last read.

The new node, *node*, to be inserted between nodes *prev* and *target* is initialized before inserting. The initialization is done by loading the required data value into its data field. The pointer field of *node*, *node.ptr*, is set to point to *target*. The CAS instruction then atomically checks if *prev.ptr* points to *target*. If the pointer field *prev.ptr* has changed or if the *target* node has changed then the instruction returns a FALSE value, indicating that the CAS instruction failed. □

Lemma 7: The CAS instruction succeeds in inserting a new node between the nodes *prev* and *target* if the pointer to the node *prev* has not changed and if neither of the nodes *prev* and *target* have been deleted.

The CAS instruction checks atomically to see if *prev.ptr* points to *target*. If the check is successful, *prev.ptr* is made to point to *node*, the new node. This ensures that the new node is inserted as *prev.ptr* points to *new* and *new.ptr* points to *target*. □

Lemma 8: Processes trying to insert new nodes between two adjacent nodes, *prev* and *target* succeed, when *prev* and *target* for each of the processes are all disjoint.

For each of the processes, if the pointer *prev.ptr* has not changed and the nodes *prev* and *target* have not been deleted since they were last read, *Lemma 7* guarantees that the new nodes are inserted between the *prev* and *target* nodes of each of the processes. The

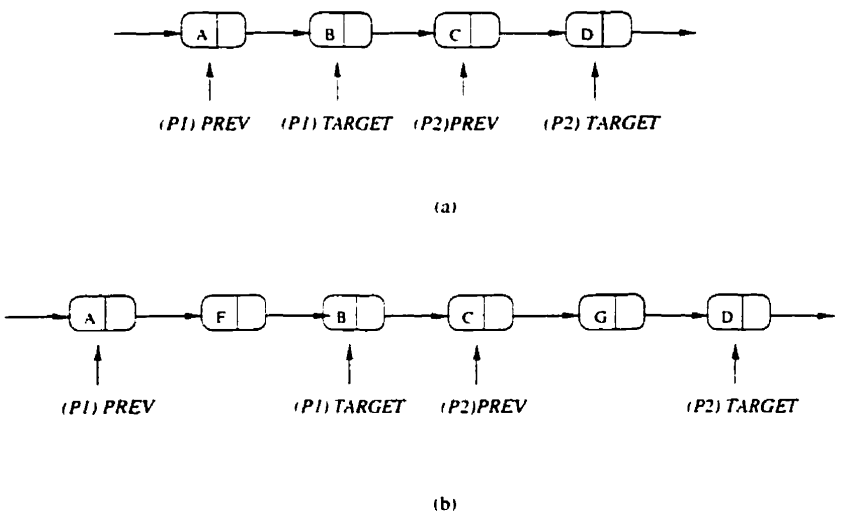


Figure 5.3: Processes trying to insert nodes adjacent to each other.

linked list is in a consistent state because for each of the processes, their *prev.ptr* points to *new* and *new.ptr* points to *target*. This ensures that the new nodes have been inserted between the *prev* and *target* nodes of each of the processes, which in turn ensures the consistency of the linked list. □

Figure 5.3 illustrates lemma 8. Two processes each have pointers *prev* and *target* pointing to disjoint sets of nodes. As the pointers are different both the processes succeed in inserting nodes between their respective *prev* and *target* pointers. Process 1 inserts node F between A and B while process 2 succeeds in inserting node G between C and D as shown in Figure 5.3(b).

Lemma 9: When processes try to insert nodes between the same set of adjacent nodes *prev* and *target*, only one of the processes succeeds while the others fail.

Consider a situation where two or more processes have read the same set of nodes, *prev* and *target* to insert a new node between them. In such a situation, only that process from among the processes competing to insert the nodes, which finds the pointer of the node *prev.ptr* has not changed and the nodes, *prev* and *target* have not changed succeeds. The CAS operation then successfully inserts the new node *new*, due to *Lemma 7*. The other processes when they subsequently try to insert, fail as the pointer field *prev.ptr* and the *target* node have changed (see *Lemma 6*). □

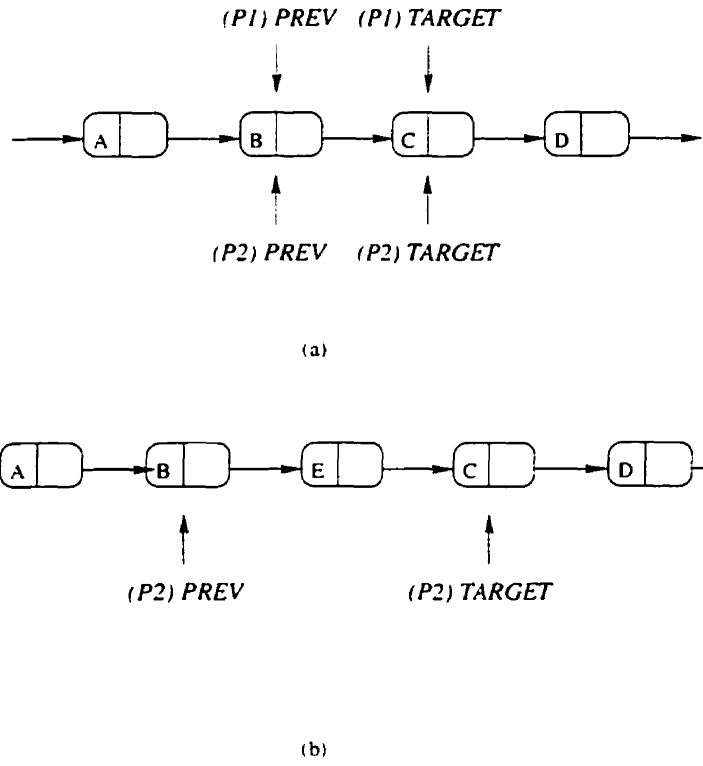


Figure 5.4: Processes trying to insert nodes between the same set of adjacent nodes.

In the example shown in Figure 5.4, process 1 and 2's *prev* and *target* pointers are the same. If process 1 succeeds then it will introduce a new node between nodes B and C. Later when process 2 tries to insert a new node between B and C, it fails as B's pointer

field has changed, it points to the new node E and not to C.

Theorem 2: When processes try to insert nodes in the linked list, a process is able to insert a node, if the consistency of the linked list can be maintained.

The different situations under which processes could insert nodes are:

- If two or more processes are trying to insert nodes between disjoint set of adjacent nodes, then each of the process is successful due to *Lemma 8*.
- When processes are trying to insert nodes between the same set of adjacent nodes, then just one process will be successful while others fail because of *Lemma 9*. The above two situations are the different possibilities in which nodes can be inserted and these guarantee the consistency of the linked list.

□

5.3 Concurrent Insertions and Deletions

Lemma 10: When processes try to concurrently insert and delete nodes then all the processes succeed if none of the processes are accessing the same set of nodes.(i.e., *prev* and *target* for the processes trying to insert the nodes and *prev*, *target* and *next* for the processes trying to delete the nodes are all distinct).

Consider a situation where each of the processes trying to insert a node, each reads the set of nodes *prev* and *target*. As these sets of nodes are disjoint for each of the processes, *Lemma 8* guarantees that all the processes succeed in inserting the new nodes.

For the processes trying to delete the *target* node from the set of nodes (*prev*, *target*, and *next*) that each process reads, neither the nodes nor the pointer fields of these nodes have changed. *Lemma 3* guarantees that in such a situation all the processes succeed in deleting their *target* nodes. Hence all the processes, those that are trying to delete and those that are trying to insert, succeed. □

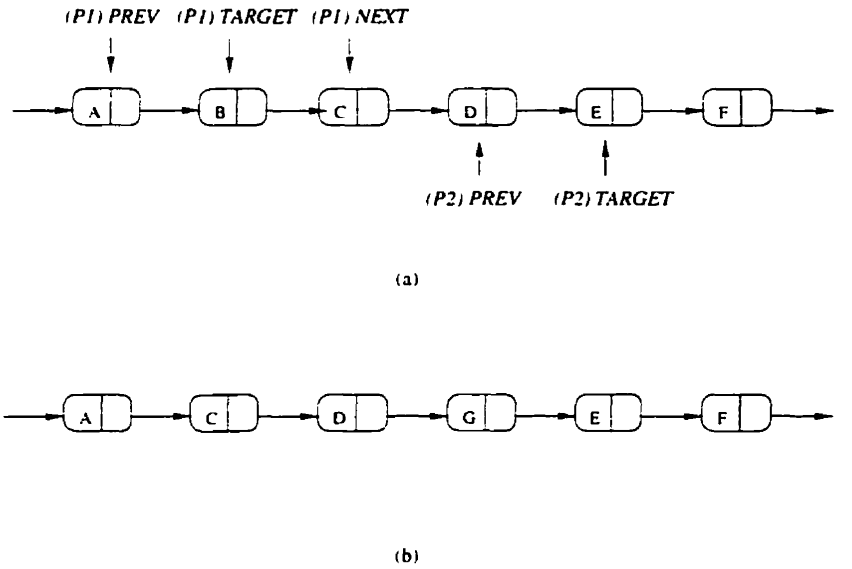


Figure 5.5: Pointers of processes pointing to distinct nodes trying to insert and delete nodes.

The example in Figure 5.5 illustrates the above lemma using two processes. Process 1 has it's three pointers *prev*, *target*, and *next* positioned at nodes A, B, and C respectively. Process 2's pointers *prev* and *target* are at nodes D and E respectively. Process 1 successfully deletes node B as none of the three pointers would have changed. Process 2 is also successful in inserting the new node G between D and F. Both the processes are successful as the pointers do not point to the same nodes. Thus, irrespective of the type of operations, all the processes will be successful as long as the pointers of each of the processes are pointing to distinct nodes.

Lemma 11: When processes try to concurrently insert and delete nodes then all the processes succeed although their pointer values are not distinct if either or both the following conditions hold:

- **Condition 1:** For the processes trying to insert nodes between the set of nodes *prev* and *target*, and the set of processes trying to delete the *target* node from the set of nodes that each such process maintains, namely, *prev*, *target*, and *next*, the *target*

node of the process trying to insert the node, and *prev* node of the process that is deleting the node are the only nodes overlapping for all the processes.

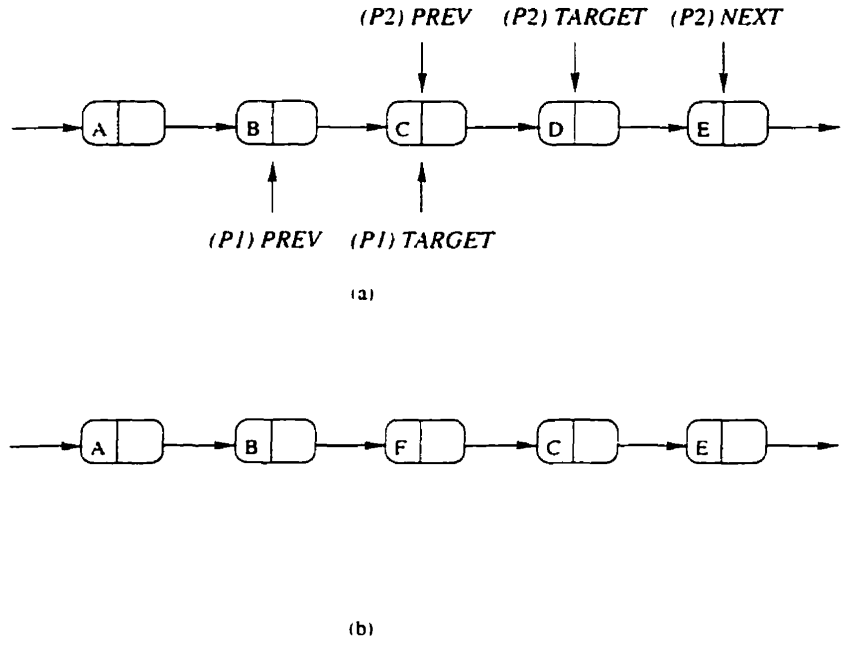


Figure 5.6: Processes trying to insert between the same set of adjacent nodes.

- **Condition 2:** For processes trying to insert nodes between the set of nodes *prev* and *target*, and processes trying to delete the *target* node from the set of nodes that each process maintains, namely, *prev*, *target*, and *next*, the *prev* node of the process trying to insert the node, and *next* of the process that is deleting the node are the only nodes overlapping for all the processes. □

Condition 1 is illustrated in Figure 5.6. Process 1 has to insert a node between nodes B and C, while process 2 is trying to delete D. In the situation illustrated, the *target* pointer of process 1 overlaps with the *prev* pointer of process 2. Suppose that process 1 succeeds first, then it inserts a new node F between B and C. Later, when process 2 tries to delete node D it will also successfully delete it. This is because process 2's pointer fields, *prev*,

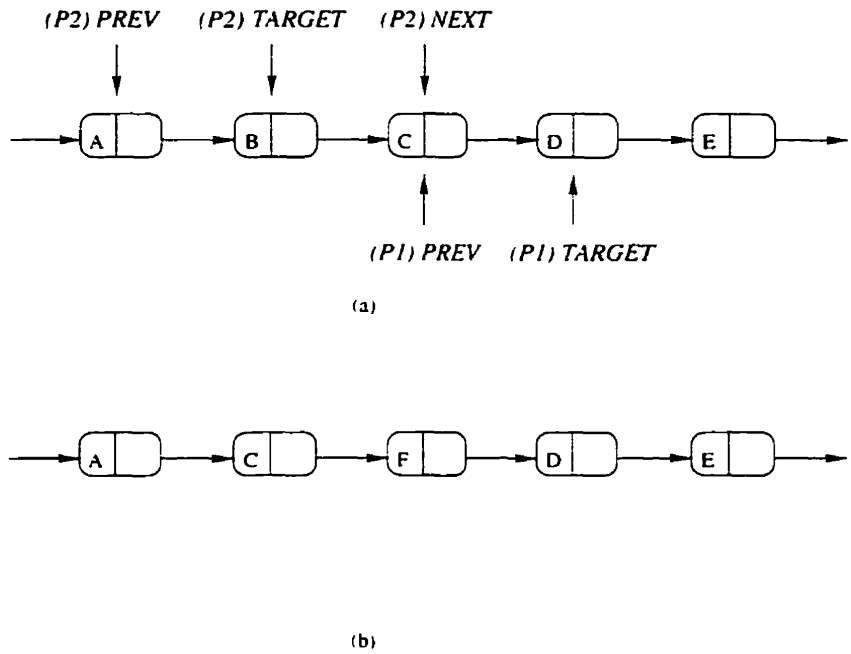


Figure 5.7: Processes with partially overlapping pointers trying to insert and delete nodes.

target, and *next* have not been changed by process 1's operation. The linked list after completing both the operations is shown in Figure 5.6(b).

Figure 5.7 illustrates Condition 2. This is similar to the first condition except that the *next* node of process 2 overlaps with *prev* node of process 1. Even in this case both the processes will complete their respective operations as the pointers of both the processes do not change.

In the situations described above, pairs of processes exist having one common node between them. The common node is present between a process trying to delete a node and another process trying to insert a node. The common node could be in one of two positions:

1. When the node *target* of the process trying to insert a node is the same as the node *prev* of the process trying to perform a delete operation.

2. When the node *prev* of the process trying to insert a node is the same as the node

next of the process trying to perform a delete operation.

From the given conditions it can be seen that for the process trying to insert a node between it's *prev* and *target* nodes, the following conditions are true. The conditions are:

- The nodes *prev* and *target* for the process trying to insert a node between them have not been deleted.
- The pointer field of *prev* has not been changed.

All the processes trying to insert succeed as long as the conditions of *Lemma 4* are satisfied.

For the process trying to delete its *target* node the following conditions are true:

- The nodes *prev*, *target*, and *next* are not deleted.
- The pointers *prev.ptr* and *target.ptr* do not change.

As the above conditions are satisfied the processes trying to delete their *target* nodes succeed according to *Lemma 3*. The above conditions are satisfied irrespective of the order in which the two processes perform their respective operations. As the same condition applies to all pairs of processes having a common node between them, all the processes succeed in performing their respective operations.

Lemma 12: When two or more nodes from the set of nodes that each process reads to insert a new node or to delete a node are the same, then some of the processes succeed while other processes fail. The processes that succeed are those whose set of pointers have not changed.

A process trying to insert a node between the nodes *prev* and *target* succeeds due to *Lemma 7* if the following conditions are true:

- The nodes *prev* and *target* for the process trying to insert a node between them have not been deleted.

- The pointer field of *prev* has not been changed.

All the processes trying to insert succeed as the conditions of *Lemma 7* are satisfied. If any of the conditions are violated then *Lemma 6* does not hold and hence the processes fail. Similarly for the each of the processes trying to delete their *target* node, the processes succeed due to *Lemma 2* if the following conditions are true:

- The nodes *prev*, *target*, and *next* are not deleted.
- The pointers *prev.ptr* and *target.ptr* do not change.

If any of the above conditions are violated then the delete operation fails due to *Lemma 1*. Therefore, only those operations which satisfy the stated conditions succeed while the other processes fail. □

An example of the situation handled by this lemma is illustrated in Figure 5.2. Figure 5.2(a) shows two pointers of each of the processes overlapping. In such a situation the first of the two processes is the one that succeeds, the process that executes its operation second fails as the pointers would have changed.

Lemma 13: If a process tries to insert a value in an empty list then it is inserted between the "FIRST" node and the "LAST" node of the linked list

The process trying to insert a node in a linked list specifies the value of the "LAST" node's data value which is unique. The process is returned the FIRST node as the *prev* node and the *last* node as the *target* node. Since the pointers *prev* and *target* cannot have changed since they were last read the new node is inserted between the FIRST node and the LAST node, and the new node is the only node between the two dummy nodes. □

An example is shown in Figure 5.8. An empty linked list consists of the two nodes, *FIRST* and *LAST*. When a process wants to insert a node, it's *prev* pointer points to

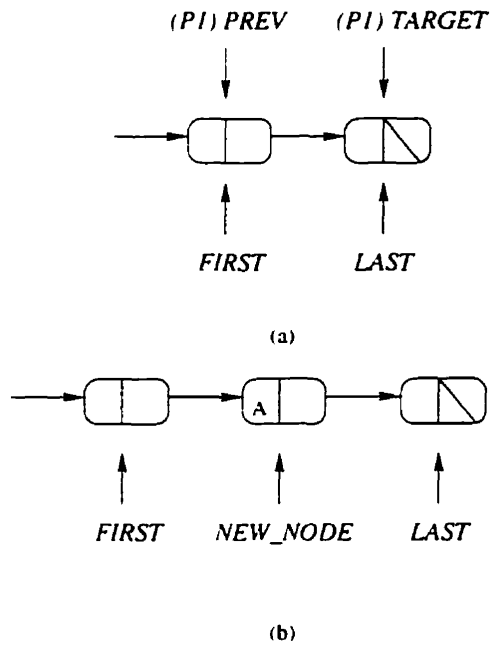


Figure 5.8: Inserting an element into an empty linked list.

FIRST and target pointer points to *LAST*. At the time of inserting the new node if the two pointers have not changed, the new node is inserted between *FIRST* and *LAST*.

Lemma 14: If a process attempts deletion from an empty list then the process fails

When a process tries to delete an element from an empty list, the process fails as the search operation for the particular node fails. Hence the process fails. □

Theorem 3: When processes try to concurrently insert and delete nodes, processes successfully perform the operations of inserting or deleting elements if consistency of the linked list can be maintained.

The different situations in which processes insert and delete nodes are

1. When processes try to concurrently insert and delete nodes then all the processes succeed if none of the processes are accessing the same set of nodes as shown by *Lemma 10*.

2. Processes trying to concurrently insert and delete nodes succeed under certain conditions when the pointers of the processes overlap as shown in *Lemma 11*.

3. When two or more nodes from the set of nodes that each process reads to insert a new node or to delete a node are same, then some of the processes succeed while others fail. This is due to *Lemma 12*.

4. If a process tries to insert a node in an empty list then it is inserted between the "FIRST" and "LAST" node of the linked list because of *Lemma 13*.

5. If a process attempts deletion from an empty linked list then the process fails because of *Lemma 14*.

The above situations enumerate the various possibilities in which processes concurrently insert and delete nodes and illustratehow the consistency of the linked list is maintained in each case. □

In the algorithms designed the concurrent operations of the processes ensures that progress will always be made, i.e. there is no situation where no concurrent process will succeed.

Chapter 6

Experimental Results

The overall goal of our experiments was to assess the performance of our algorithm and compare it with the existing algorithms of Valois [23] and Greenwald and Cheriton [6]. The algorithm was simulated rather than running on actual parallel machines. This was necessary for a number of reasons. First it was difficult to obtain access to machines that had the required architecture. Second, even if the required machines could be obtained, the performance of the algorithm could not be accurately determined because of load imbalances which could give incorrect results. Moreover one of our goals was to have control over the load (obtained in the simulation by determining the time slice of each of the processes). By having control over the time slices of the processes we could easily generate an environment having high contention. It was easier to control the time each process gets to access the data structure via simulation.

6.1 Experimental Set-up

In our experimental set-up 20 processes were created to operate on the linked lists. The only exception was when the size of the list was only 1000 nodes, in which case 10 processes were used. This was to ensure that each process performs a substantial number of operations as only 1000 nodes were considered. The initial size of the linked list on which the processes operated varied from 1000 nodes to 25000 nodes. The total number of operations performed

by all the processes on the linked list varied from 500 to 10000. The operations were either delete or insert operations. In actual situations the number and type of operations vary. Hence the following different combinations of operations were performed on different initial sizes of the linked list:

- All the operations performed by each of the processes were delete operations.
- All the operations performed by each of the processes were insert operations.
- 75% of the operations performed by the processes were delete operations and 25% were insert operations.
- 50% of the operations performed by the processes were delete operations and 50% were insert operations.
- 75% of the operations performed by the processes were insert operations and 25% were delete operations.

The experimental details are summarized in Figure 6.1.

The experiments were initiated by constructing a linked list having a fixed number of nodes. Each of the processes then traverses the linked list for a given time-slice. The time-slice is determined randomly and is related to the number of nodes a process must travel for each time slice. Once a process' time-slice expires it allows other processes to traverse the linked list until it is scheduled again. The scheduling was done in a round robin manner. Each process runs until its assigned time slice expires. If the process has not completed its operation at the end of the time-slice, then it is pre-empted and another process is run. If the process has completed its operation and if the process has other operations to be done, then the process will initiate the new operation when it is scheduled next. If all the operations have been completed by the process then the process is never

Length of the List	100% Deletions	100% Insertions	75%Insertions 25% Deletions	50% Insertions 50% Deletions	25% Insertions 75% Deletions
1000 nodes	500 Dels	500 Ins	375 Ins 125 Dels	250 Ins 250 Dels	125 Ins 375 Dels
2000 nodes	1000 Dels	1000 Ins	750 Ins 250 Dels	500 Ins 500 Dels	250 Ins 750 Dels
4000 nodes	2000 Dels	2000 Ins	1500 Ins 500 Dels	1000 Ins 1000 Dels	500 Ins 1500 Dels
10,000 nodes	4000 Dels	4000 Ins	3000 Ins 1000 Dels	2000 Ins 2000 Dels	1000 Ins 3000 Dels
25,000 nodes	10000 Dels	10000 Ins	7500 Ins 2500 Dels	5000 Ins 5000 Dels	2500 Ins 7500 Dels

Figure 6.1: Experimental Details.

re-scheduled. Once the process finds the node which has to be deleted or a node before which a new node has to be inserted it attempts to do the respective operation.

6.2 Analysis of Results

The results of the experiments are shown in Figures 6.2, 6.3 and 6.4. The performance of the new algorithm is consistently better than Valois' algorithm and Greenwald & Cheriton's algorithm. The new algorithm performs better than the other two algorithms in all the cases. The main reasons being that the new algorithm needs only n nodes to represent n nodes in the linked list. Apart from this, the algorithm also provides concurrent read and write access to all processes. The performance of the algorithm is also enhanced in the deletion process of the new algorithm. If a process fails to delete a node because the *next* node has changed, the updated *next* node is obtained and the delete process is initiated again without re-reading the list.

Valois' algorithm performs better than Greenwald's algorithm. Although Valois' algorithm allows processes to concurrently read and write, its major drawback is that it

requires "$2n$" nodes to represent "n" nodes of the linked list. This considerably increases the traversal time for the processes. The performance of the algorithm also deteriorates because each process spends considerable time in deletion of the auxiliary nodes. This is necessary to prevent a chain of continuous auxiliary nodes.

Greenwald & Cheriton's algorithm allows processes to concurrently read the linked list but the write operations are performed in a serial manner. Moreover whenever a write operation is performed all the other processes have to re-read from the start of the linked list. This takes considerable time and therefore Valois' algorithms performance deteriorates even further.

The graphs shown in Figures 6.5 through 6.9 are plotted for varying load conditions and varying operations of the table shown in Figure 6.4. It is seen that all the graphs are similar in nature, i.e., our algorithm has the least latency, followed by Valois' and Greenwald & Cheriton's algorithm. When the number of nodes are less, it appears like the latency of all the three algorithms is the same although in reality our algorithm performs better for all the cases. This is because the latency was calculated using system time. The system, for example returns a time of 1 second if the time is either 1.1 seconds or 1.9 seconds. As the number of nodes increases the differences between the three algorithms is prominent and the performance of our algorithm shows a marked improvement over the other two.

Length of the List	100% Deletions	100% Insertions	75%Insertions 25% Deletions	50% Insertions 50% Deletions	25% Insertions 75% Deletions
1000 nodes	0.67	0.67	0.33	0.33	0.33
2000 nodes	1	1.67	1.67	2	1.67
4000 nodes	6.33	4.67	6.33	7	7
10000 nodes	34.33	37	33	33.33	34.67
25000 nodes	240.67	143	200.33	222.33	213.67

Figure 6.2: New Algorithm (Time in seconds).

Length of the List	100% Deletions	100% Insertions	75%Insertions 25% Deletions	50% Insertions 50% Deletions	25% Insertions 75% Deletions
1000 nodes	1	1	1.33	1.67	1
2000 nodes	3.67	4	5	5.33	34
4000 nodes	14.33	18.67	22.33	22	20.67
10000 nodes	73	123.67	116.33	103	97
25000 nodes	584	408.67	503	529	496.33

Figure 6.3: Valois' Algorithm (Time in Seconds).

Length of the List	100% Deletions	100% Insertions	75%Insertions 25% Deletions	50% Insertions 50% Deletions	25% Insertions 75% Deletions
1000 nodes	1	2	1.67	1.67	1.67
2000 nodes	10	15	12.33	15.67	12.33
4000 nodes	38.67	52.33	46.67	53.67	43
10000 nodes	319	479	282.67	244	251
25000 nodes	783.67	694.33	869.67	869.67	861

Figure 6.4: Greenwald's Algorithm (Time in Seconds).

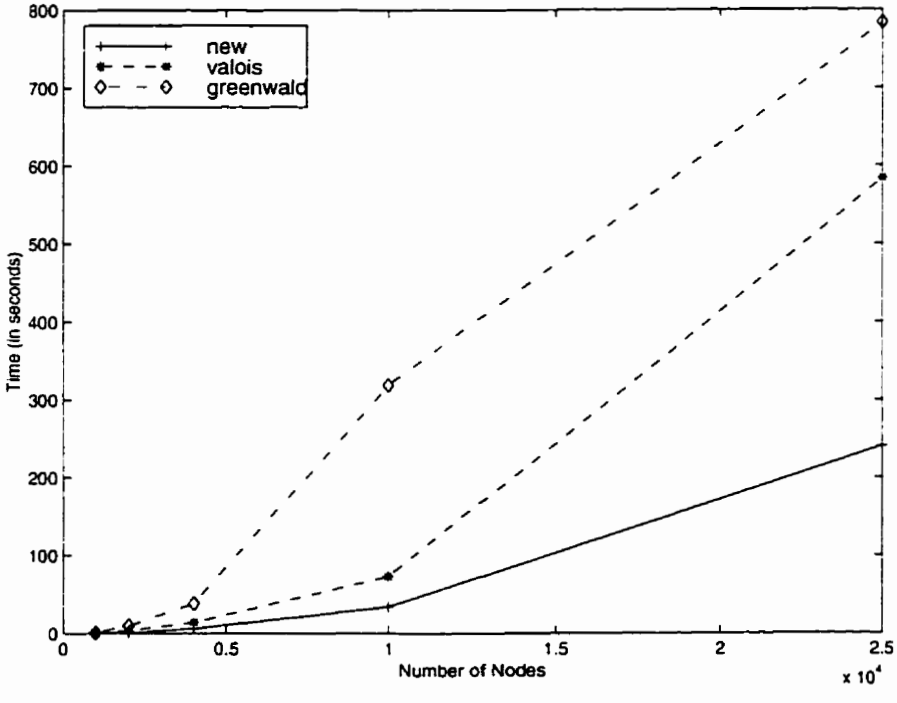


Figure 6.5: 100% deletion operations.

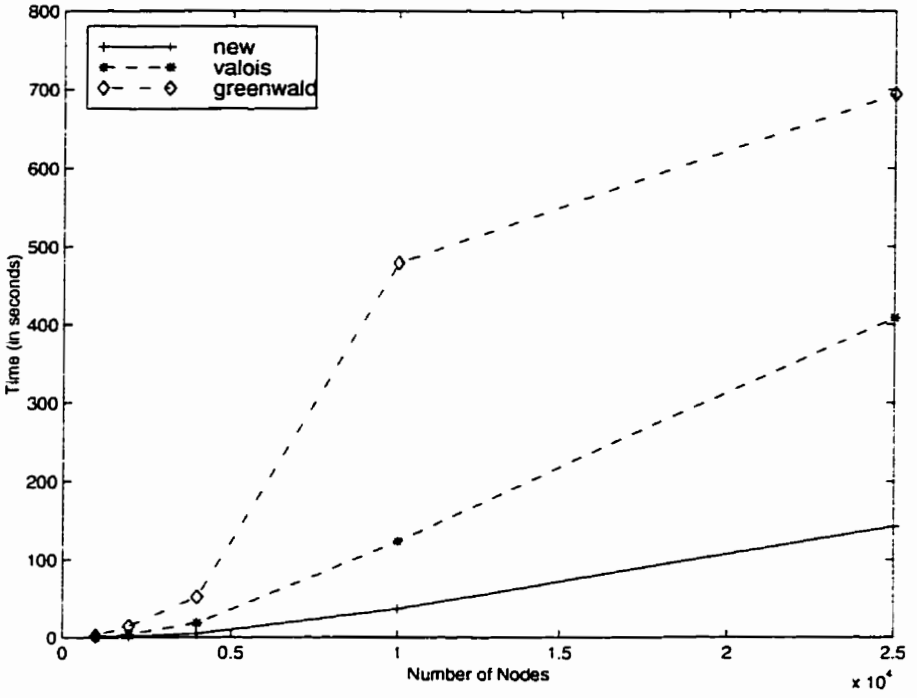


Figure 6.6: 100% insertion operations.

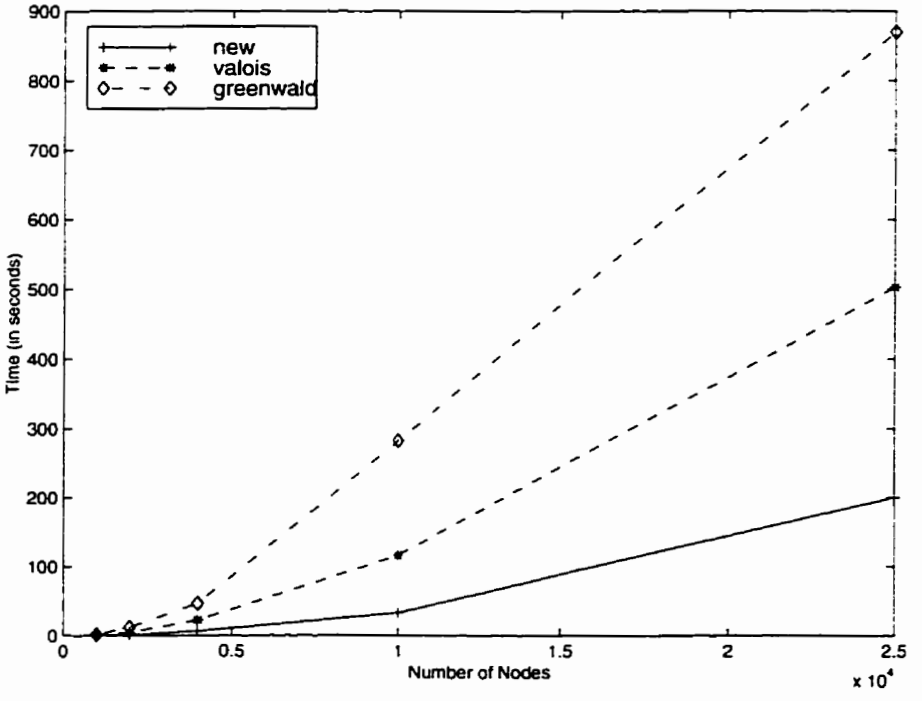


Figure 6.7: 75% of the operations are insertion and 25% are deletions.

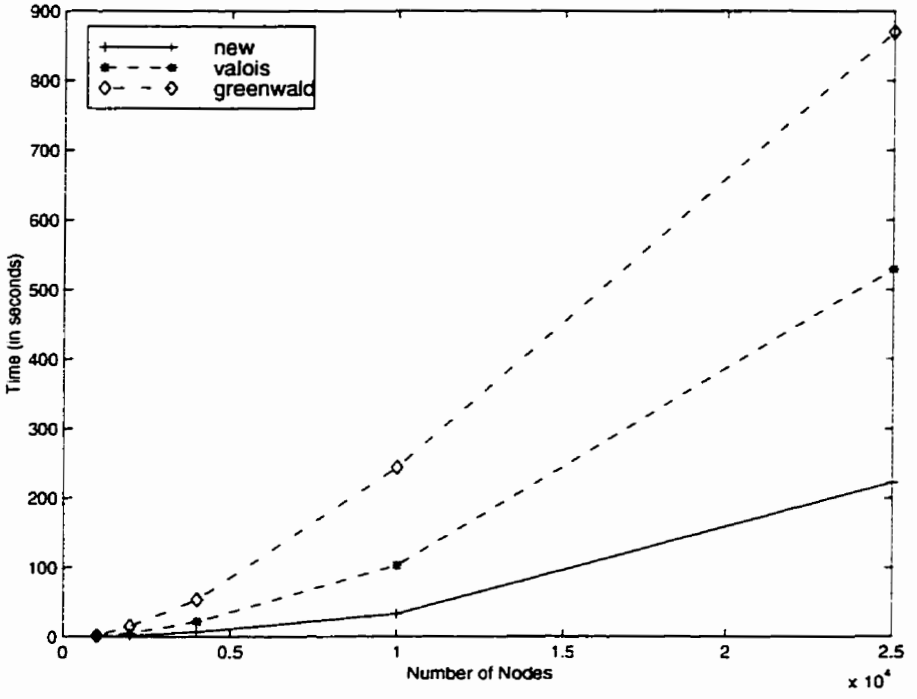


Figure 6.8: 50% of the operations are insertion and 50% are deletions.

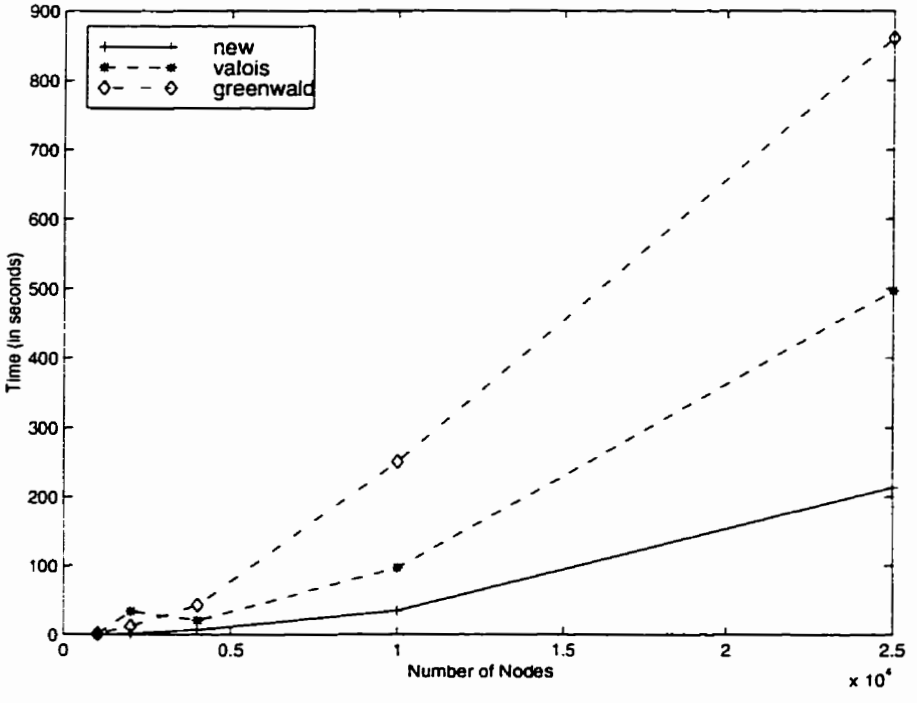


Figure 6.9: 25% of the operations are deletions and 75% are insertions.

Chapter 7

Conclusions and Future Work

Their is a growing need to find better techniques and algorithms to handle manipulation of concurrent objects in distributed environments. The use of lock-free synchronization for manipulating concurrent objects enhances their performance as it overcomes the inherent drawbacks of spin locks where delays occur in the critical section.

In this thesis we have presented algorithms for manipulating singly linked lists with concurrent processes operating on them without the use of mutual exclusion. The algorithms described include traversal of the linked list, inserting elements into the linked list and deleting elements from the linked list. *Double Compare & Swap* and *Compare & Swap* are the two synchronization instructions that have been used in implementing the algorithms. Our algorithm was simulated along with Valois's and Greenwald & Cheriton's algorithms under varying load conditions and different combinations of operations.

Effective and efficient use of the synchronization instructions has resulted in our algorithm achieving better performance compared to Valois and Greenwald & Cheriton's algorithms. The main reasons for this are:

- The traversal time is reduced as we require only n nodes in a linked list to represent n nodes.
- The algorithms allow concurrent read and write operations by processes.

Valois' algorithm allows concurrent read and write access to the processes, but has several drawbacks which affect it's performance:

- Increased traversal time as Valois' algorithm requires $2n$ nodes in a linked list to represent n nodes.
- Each process spends considerable time removing auxiliary nodes from the linked list. This is to prevent the formation of continuous chains of auxiliary nodes in the linked list.

In comparison to our algorithm and Valois' algorithm, Greenwald & Cheriton's algorithm does not perform well. The main reasons for it's poor performance are:

- The algorithm only allows processes to concurrently read the linked list but the updates to the linked list are done in a serial manner.
- Each time the linked list is updated, which may be due to addition of a node or deletion of a node, the process that failed has to start the entire operation again by searching from the starting of the linked list.

In our algorithm if *next* node has changed, the updated *next* node is read and the operation is continued. The process fails and re-read is initiated only if *prev* and/or *target* nodes have changed. Another advantage of our algorithm is that it requires less time to delete nodes and maintain consistency of the linked list compared to Valois' algorithms.

7.1 Future Research

One of the main areas of further research is in developing lock-free algorithms for other data structures such as queues, trees and other abstract data types. The main focus while developing such algorithms for the above mentioned data structures might be concentrating on reducing the latency and thus increasing the performance. Research could also be

focussed towards finding different synchronization instructions which would help in developing better lock-free algorithms. Methods could be developed to make these algorithms wait-free, since non-blocking algorithms cannot prevent starvation of individual processes. It would also be interesting to asses the performance of our algorithm under real conditions.

Most of the research focus in lock-free synchronization has been in the area of developing lock-free algorithms for basic data structures. Other data structures such as trees and their variations (such as trees with a local stack) can also be considered. This could also be extended to other areas such as databases. The feasibility of applying lock-free synchronization methods for manipulating databases (especially distributed databases) could be looked into. The application of lock-free concepts to optimistic concurrency control [1] in distributed databases could significantly reduce the access time which would result in better throughput.

Bibliography

[1] A. Adya, R. Gruber, Barbara. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. *SIGMOD*, 24(2):23–34, 1995.

[2] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

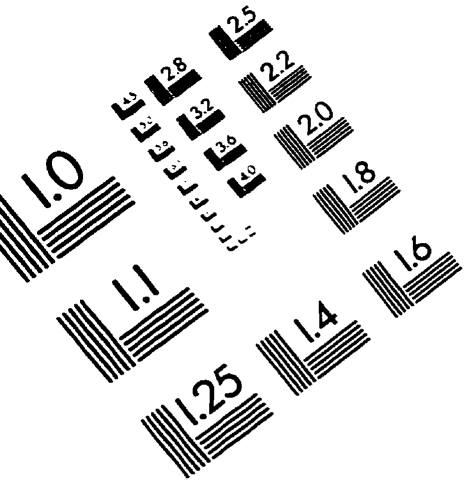
[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

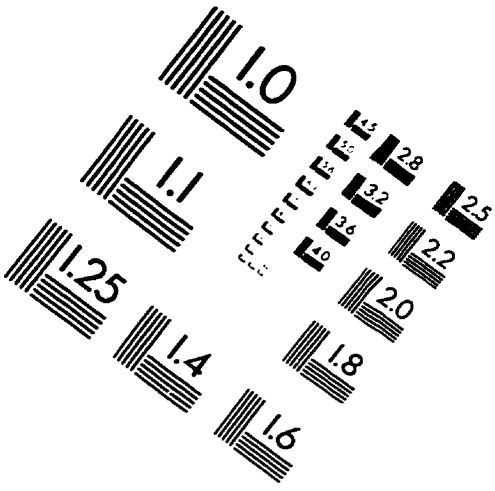
[4] A. Glew and W. Hwu. A Feature Taxonomy and Survey of Synchronization Primitive Implementations. Technical Report CRHC-91-7, University of Illinois at Urbana-Champaign, 1991.

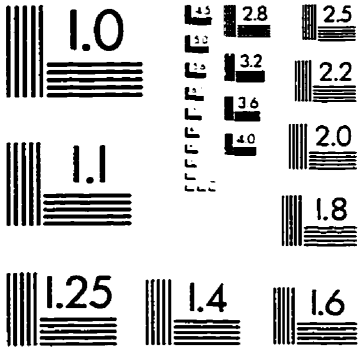
[5] A. Gottlieb, B. D. Lubachevsky, and Rudolph L. Basic Techniques for the Efficient Coordinaton of Very Large Numbers of Cooperating Sequential Processor. *ACM Transactions on Programming Languages and System*, 5(2):164–189, April 1983.

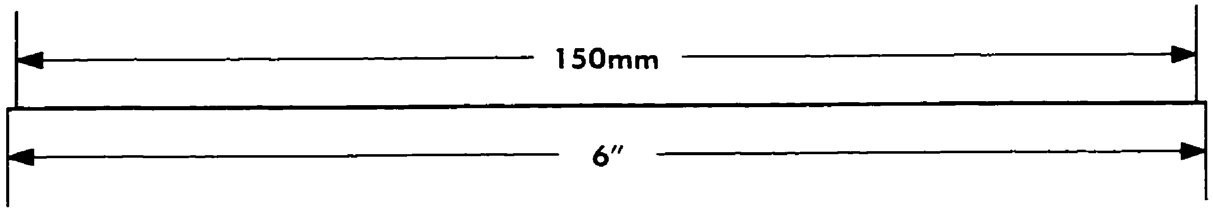
[6] M. Greenwald and D. Cheriton. The Synergy Between Non-Blocking Synchronization and Operating System Structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation. USENIX*, pages 123–136, October 1996.

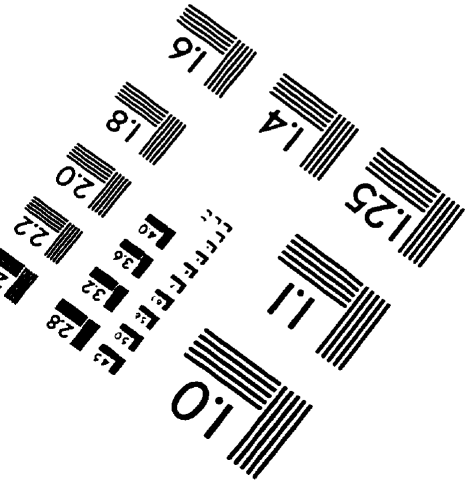
IMAGE EVALUATION
TEST TARGET (QA-3)

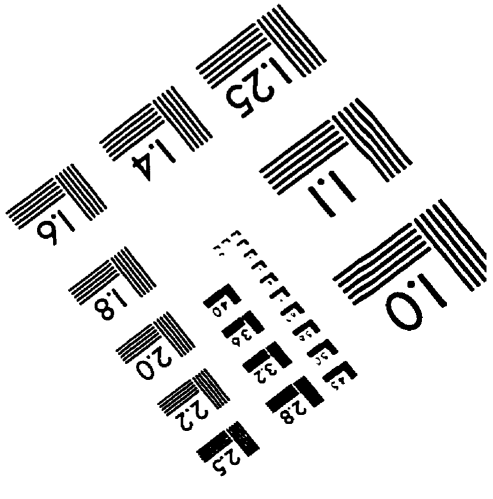












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989