CONCURRENT OPERATIONS IN PERSISTENT SEARCH TREES

BY

HACK HOO NEW

Bachelor of Science in Business Administration

Oklahoma State University

Stillwater, Oklahoma

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1988

CONCURRENT OPERATIONS IN PERSISTENT

SEARCH TREES

Thesis Approved:

Donald D Fisher

Thesis Adviser

M J Folk

Norman N. Durham

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express sincere appreciation to Dr. Donald D. Fisher for his encouragement and advice throughout my graduate program. My special thinks also go to Dr. K. M. George and Dr. Michael J. Folk for serving on my thesis committee.

I thank Yew Lan, my wife, Yu Lik and Soh Kek, my parents, and everyone back in Singapore for their love and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As the construction of large multiprocessor systems becomes feasible, parallel asynchronous algorithms appear more and more in different applications such as concurrent databases, operating systems, and distributed systems. Multiprocessor systems might be used to service the needs of a large number of users simultaneously, or to reduce the time necessary for a single complex job. Therefore, it is important to investigate data structures that support efficient concurrent operations.

In this thesis we present a system that can support a number of concurrent processes which perform search, insertion, and deletion in persistent search trees without causing deadlock and without destroying the integrity of the data structure. A persistent search tree is a search tree which retains both the topology and the items at each update time. Initially the search structure is empty, then at each update time operations are performed on the data structure to create a new version of the search tree. These versions are linearly ordered by the creation time. Generally, queries are allowed in all versions of the persistent search tree, but updates can only be made in the

current version. Persistent search trees are useful in computational geometry [16,17], version management for databases [21], and text editing [13,14].

We can make an ordinary search tree persistent by copying the entire current tree at each update time. However, this method is very undesirable because it takes O(n) time and O(n) space per insertion or deletion in the worst case, where n is the number of nodes in the current version of the tree. Myers [13,14], Krijnen and Meertens [8], Reps, Teitelbaum, and Demers [15], and Swart [17] independently propose a technique, called path copying, that has an O(log n) time bound per operation and O(log n) space bound per update. Path copying requires any node that contains a pointer to a node that is copied must itself be copied. This means that copying one node requires copying the entire path to the node from the root of the tree. The effect of this method is to create a set of search trees, one per update, having different roots but sharing common subtrees. Sarnak and Tarjan [16] improve on the path copying method by introducing a more space efficient method called limited node copying. Searching an item in the resulting data structure takes O(log n) time in the worst case and an update (insertion or deletion) takes O(1) space in the amortized case. Limited node copying requires each node to provide extra space for a small number of auxiliary pointers in addition to the existing left and right pointers. A node is copied with its latest

left and right children only if there is no empty slot for a new pointer to be added to the node in an update.

We further improve on Sarnak and Tarjan solution by providing a facility for concurrent access to the persistent search trees. We use a simple and yet efficient tree locking protocol similar to [5] for process synchronization. The underlying structure for our system is a collection of red-black trees. We choose red-black trees because an insertion or deletion can be efficiently performed in a single top-down pass. The existence of the top-down algorithms provides us a way to predict the portion of the data structure that may be affected by a particular operation in just one top-down pass. Therefore appropriate actions can be taken to block out other tasks from this small portion of the data structure in order to maximize the concurrency. (Throughout this thesis we will use the terms "process" and "task" interchangeably).

Literature Survey

In this section we shall look at some recent studies on concurrent access to binary search trees. In the past decade, data structures that support concurrent search trees have been studied extensively. Kung and Lehman [9] present a set of concurrent algorithms to manipulate the binary search trees using only the exclusive locks. Even though search, insert, and rotate operations are fast and easy, deletion is quite lengthy because deleting an

internal node is done by first moving the node to the bottom of the tree, using successive rotations, and then deleting it. The deletion algorithm is improved by Manber and Ladner [12] who introduce the notion of maintenance processes that relieve the deleting process from the job of restructuring the tree. Furthermore, both shared and exclusive locks are used to minimize the interference between readers and writers, thus the level of concurrency is increased substantially. A reader is a process which is searching through the tree, whereas a writer is a process which is attempting to do an insertion or deletion operation. The deletion algorithm is further improved by Manber [11] who presents a system that manipulates the external binary search trees. Since the data always reside on the leaf nodes, the deletion algorithm is easy to implement because swapping or rotation is never needed. The major draw back of all these implementations is that none of the above solutions uses balanced search trees for the underlying structures. As a result, inefficient data structures can occur after a sequence of updates and thus lead to poor performance.

Ellis presents a set of algorithms for concurrent search and insert in AVL trees [5] and 2-3 trees [6]. These algorithms are more complicated because the balanced properties of the AVL and 2-3 trees are maintained by rotations and splitting nodes respectively. Ellis also uses both the shared and exclusive locks for different

actions to minimize the interference among tasks. Since these search trees are more compact and the interference among tasks is minimized, these algorithms are more efficient than previous algorithms.

Outline of The Thesis

In chapter 2, we present a set of sequential algorithms for manipulating persistent red-black trees. In chapter 3, we present a model that can support concurrent access to persistent red-black trees by employing an efficient tree locking protocol for concurrency control. In chapter 4, implementation issues and simulation results are discussed. Conclusions are given in chapter 5.

CHAPTER II

SEQUENTIAL OPERATIONS IN PERSISTENT RED-BLACK TREES

Red-Black Trees

An external binary search tree is a tree structure that can be used to represent a set of items selected from a totally ordered universe. It consists of a binary tree containing the items of the set in its external nodes, one item per external node, with the items arranged in ascending order from left to right in the tree. In addition, each internal node contains an item in the universe, called a key, such that all items in the left subtree of the node are less than or equal to the key and all items in the right subtree are greater than the key. An item in the tree can be accessed in time proportional to the depth of the tree by starting at the root and searching down along the search path. When arriving at an internal node, the key is compared with the desired item to decide whether to branch left or right. If the item is less than or equal to the key, we branch to the left child; otherwise, we branch to the right child. Eventually an external node is reached; either this node contains the desired item or it is not in the set.

A red-black tree is an external binary search tree in which each node is colored red or black in a way satisfying the following constraints:

(1) All external nodes are black.

(2) All paths from the root to an external node contain the same number of black nodes (black constraint).

(3) Any red node, if it is not a root node, has a black parent (red constraint).

An example of a red-black tree is shown in Figure 1. Note that the square nodes are external nodes and the letter on the right of an internal node denotes the color, 'r' for red and ' ' for black.

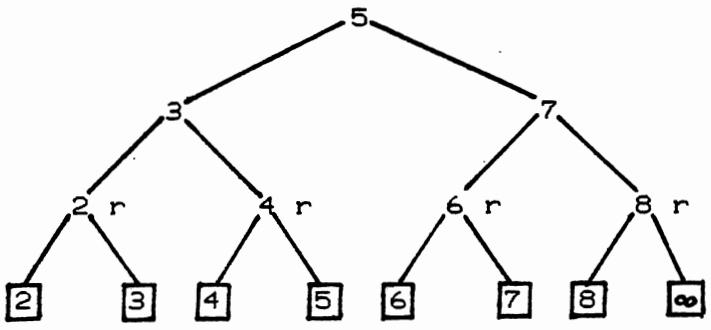


Figure 1. A Red-Black Tree.

Bayer [1] first introduced these trees, calling them binary B-trees. The balance properties of the red-black trees are maintained by color changes or a combination of color and pointer changes (Figures 8 and 10). Rebalancing the tree after an update (insertion or deletion) operation takes O(1) rotations in the worst case and O(1) color changes in the amortized case [20]. Another nice feature of the red-black tree is that an update operation can be done in a single top-down pass [19]. This is, of course, the main reason why we choose red-black trees as the underlying search structure for our concurrent model.

Persistent Red-Black Trees

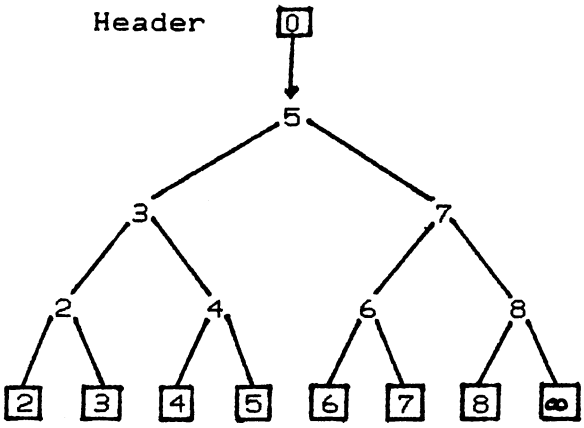
There are basically two ways of making a search tree persistent, namely by employing path copying or limited node copying method. Now, we shall use a simple example to illustrate the pros and cons associated with these methods. As a simple example, suppose we want to insert an item 9 into the red-black tree in Figure 2(a) at (current) time 1, and the initial tree is created at time 0. In order to make the red-black tree persistent with the path copying method, that is to create a new version of the search tree for time 1 having different root but sharing the common subtrees, every node encountered on the search path must be copied. The persistent red-black tree in Figure 2(b) is the resulting structure after the insertion. Note that if we traverse the persistent red-black tree in Figure 2(b) at

time 0, we will find all the items in the initial tree and nothing else; if we traverse at time 1, we will find all the items in the initial tree plus item 9. An item in the resulting tree can be accessed in time proportional to the depth of the tree, that is O(log n). The space per update (insertion or deletion) is also O(log n) since such an operation only copies nodes along the search path.

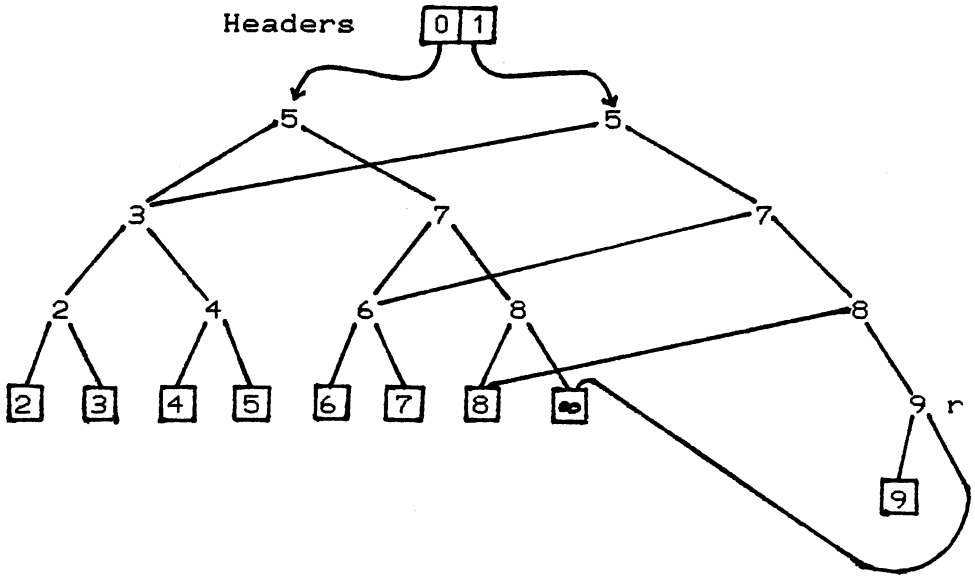
For the same example, if we use the limited node copying method instead of path copying, the resulting tree is shown in Figure 3. With node copying, an update operation only takes O(1) space in the amortize case and O(log n) time in the worst case.

By just looking at these examples, it is quite obvious that path copying offers a faster search time on the resulting structure since it only needs to examine the key at each internal node to decide whether to branch left or right. However, searching the persistent red-black tree in Figure 3 is quite complicated and we shall discuss it in detail in the next section. Limited node copying seems to be more space efficient than path copying since fewer new nodes are created in an update. If all things are considered, limited node copying is probably much more difficult to implement than path copying. The foregoing comments will make more sense when we discuss the search, insertion, and deletion algorithms in the following sections.

In a sequential environment, it is hard to say which method is better. However, in a concurrent environment we



(a) A Red-Black Tree Created at Time 0.



(b) The Resulting Tree.

Figure 2. A Persistent Red-Black Tree with Path Copying.

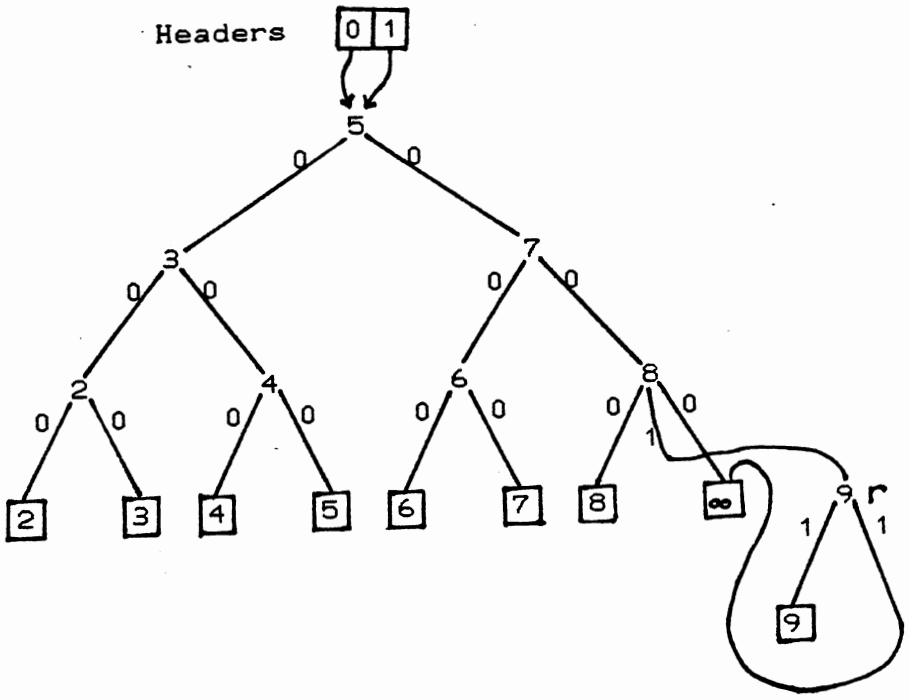
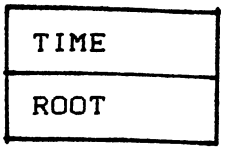


Figure 3. A Persistent Red-Black Tree With Limited Node Copying. The initial tree is as in Figure 2(a). The edges are labeled with their time of creation.

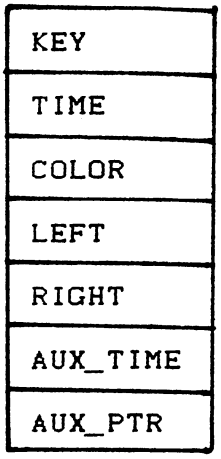
do know that limited node copying usually provides a higher degree of concurrency. It is the copying of the entire search path that makes path copying undesirable. During this lengthy copying process, none of the other tasks are allowed to update the tree in order to avoid the interference and thus no concurrency is possible. One possible solution is to copy nodes as they are encountered during the search phase. However, this solution may create a large number of redundant copying operations when tasks attempt to insert items that are already on the tree or delete items that are not in the tree. On the contrary, limited node copying method usually copies nodes located near the bottom of the tree; moreover, it only copies a small number of nodes at a time. Since the copying process usually occurs near the bottom of the tree, it is possible to allow other tasks to update the portion of the current tree that is not involved in the copying simultaneously. Therefore limited node copying is clearly the better choice for the concurrent environment.

Now, we shall describe the sequential algorithms for persistent red-black trees with limited node copying in order to establish the terminology and because an understanding of these algorithms is necessary to understand the concurrent algorithms. In order to implement the limited node copying method, a number of extra fields are needed in each node and header. Figures 4(a), 4(b), and 4(c) show the structure of a header, internal node, and external

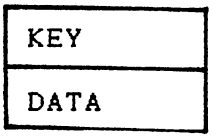
node, respectively.



(a) Header Structure.



(b) Internal Node Structure.



(c) External Node Structure.

Figure 4. Record Structures for Header and Nodes.

TIME is the creation time of a header or an internal node. ROOT is a pointer to the root which is associated with the header. KEY is a key field. COLOR is a flag indicating whether the color of a node is red or black. LEFT and RIGHT are pointers to the left and right children respectively. AUX_PTR is an auxiliary pointer that can be used to point to a left or right child. We shall say a node has a free or empty slot for a new pointer if AUX_PTR contains the 'NULL' value. AUX_TIME is the time associated with the AUX_PTR. DATA is a data field. The extra fields needed with the limited node copying are TIME, AUX_TIME, and AUX_PTR for each internal node, and TIME for each header. Note that each internal node can contain more than one extra pointer and its associated time stamp, but we shall use only one auxiliary pointer throughout this thesis for simplicity.

Sequential Search Algorithm

Searching is the process of locating an item associated with a particular time stamp. Searching a persistent red-black tree with limited node copying is more complicated than searching a persistent red-black tree with path copying or an ordinary search tree. The search proceeds from a header associated with the search time down along the search path. When arriving at an internal node, we not only must examine the key to decide whether to branch left or right, but the multiple left or right pointers as well.

We follow the pointer which points to the desired direction and has greatest time stamp no greater than search time. Eventually the search reaches an external node; either this node contains the item or it is not in the tree. Regardless of the result, the pointer to the parent of the external node is returned so that both the insertion and deletion algorithms can utilize this search procedure to locate the appropriate location for updating.

Sequential Updating Algorithms

An update (insertion or deletion) operation is complicated by the fact that nodes encountered during the pointer changes may not have enough free slots for the new pointers to be added to them. Whenever we attempt to add a pointer to a node X which has no free slot for the new pointer, a new copy of X, say X', is created with the latest values of the left and right pointers of X. Then the pointer to X' must be stored in its latest parent as well. If the parent has no free slot, it too is copied; this copying process is repeated up along the access path until a node with a free slot is found or the root node is copied.

Once a node has been copied, it becomes a dead node. The dead nodes are those not reachable from the current tree root by following the latest pointers, whereas the live nodes are those reachable from the current root. An update operation only affects the live nodes but not the

dead nodes, so information stored in the dead nodes is not destroyed by the succeeding update operations. As the current time increases, the live nodes can become dead but not vice versa.

Sequential Insertion Algorithm

An insertion is the process of adding a new item to the current tree. If the new item is not already in the tree, it is inserted. The insertion algorithm proceeds as follows :

(1) Use the above search procedure to locate the parent of the new node, say V.

(2) Create a new external node containing the new item and a new red internal node, say K, containing the minimum key of the new external node and the child of V which lies on the access path.

(3) Store the pointer to node K in its parent. Since node V may not have an empty slot for the new pointer or it may be created at the current time unit, we should examine the current state of node V before storing the new pointer so that it can be stored in the appropriate field and node. The three possible states of an internal node and the required actions are as follows:

   (a) If node V is created at current time t (Figure 5(a)), the child of node V which lies on the access path is replaced by node K (Figure 5(b)).

   (b) If node V is not created at the current time t

(Figure 6(a)) but has a free slot, the pointer to node K and time t are stored in AUX_PTR and AUX_TIME of node V respectively (Figure 6(b)).

(c) If none of the above is true (Figure 7(a)), then a copy of node V, say V', is created with the latest left and right pointers of node V (Figure 7(b)). Note that one of the pointers of node V' is pointed to node K. A pointer to node V' must also be stored in the latest parent of node V' if it is possible; otherwise, this copying process is repeated until a node with a free slot is reached or the root is copied.

(4) If V is a red node, then the red constraint is violated. To eliminate the violation, we let node K be the current node S and proceed with the following steps:

(a) If a red node S has a black parent P(S), go to step (5).

(b) If P(S) is a root node, color P(S) black and go to step (5).

(c) If P(S) has a red sibling, apply the transformation in Figure 8(a) and let the parent of P(S), be the new current node S and go to step (a).

(d) If S is the left child of P(S) whose sibling is a black node (or symmetric variant), apply the transformation in Figure 8(c) and go to step (5). Note that this transformation requires both

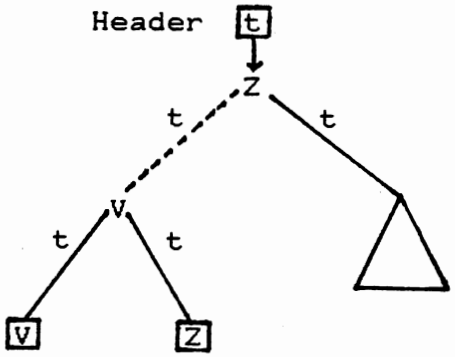
the color and the pointer changes, thus we need to make sure that nodes involved in pointer changes have free slots for the new pointers to be added to them. For a single rotation, we need to copy the promoted node, which is S, if it is not created at the current time and the grandparent of the promoted node, which is P(P(S)), if it has no empty slot and its auxiliary pointer is not pointed to the parent of the promoted node, which is P(S), at the current time.

(e) If node S is the right child of P(S) whose sibling is a black node (or symmetric variant), apply the transformation in Figure 8(d) and go to step (5). Since this transformation requires a double rotation, we need to copy the promoted node, which is the child of S which lies on the access path, if it is not created at the current time, P(S) if it has no empty slot and its auxiliary pointer is not pointed to node S at the current time, and P(P(S)) if it has no free slot and its auxiliary pointer is not pointed to P(S) at the current time.

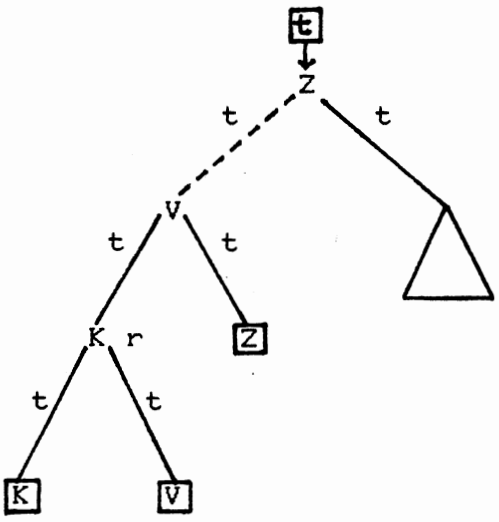
(5) End.

Notice that Figure 8(a) is the only nonterminal case. Thus, it is possible to apply this transformation a number of times, then followed by one application of transformations in Figure 8(b), 8(c), or 8(d) if necessary. The

maximum number of rotations is two -- that is one application of transformation in figure 8(d), so rebalancing after an insertion can be done in O(1) rotations.

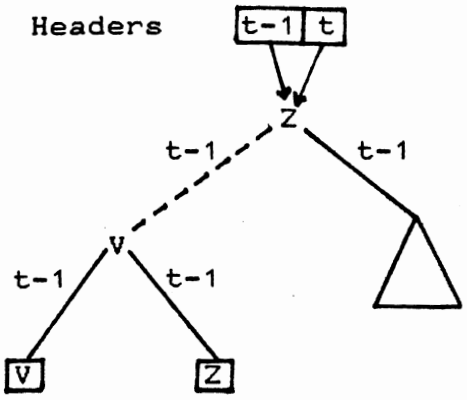


(a) Node V is created at current time unit. The dotted line between two nodes means that they are separated by zero or more nodes. The triangular nodes are subtrees.

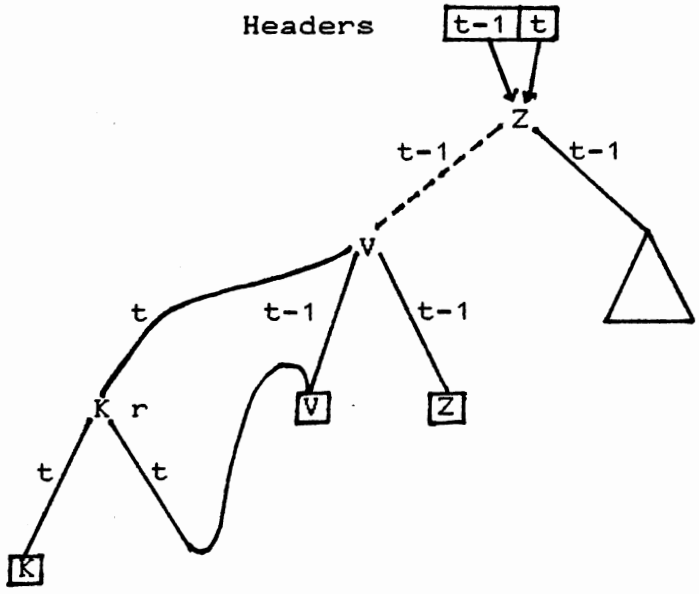


(b) The pointer to node K is inserted in the LEFT of node V.

Figure 5. An Internal Node Created in Current Time Unit.

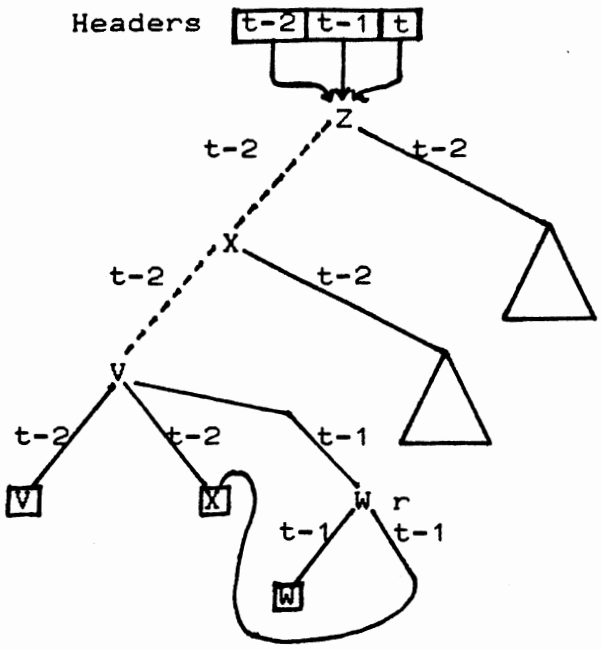


(a) Node V is created at time t-1. "t-1" is read as "<= t-1." The dotted line between two nodes means that they are separated by zero or more nodes. The triangular nodes are subtrees.

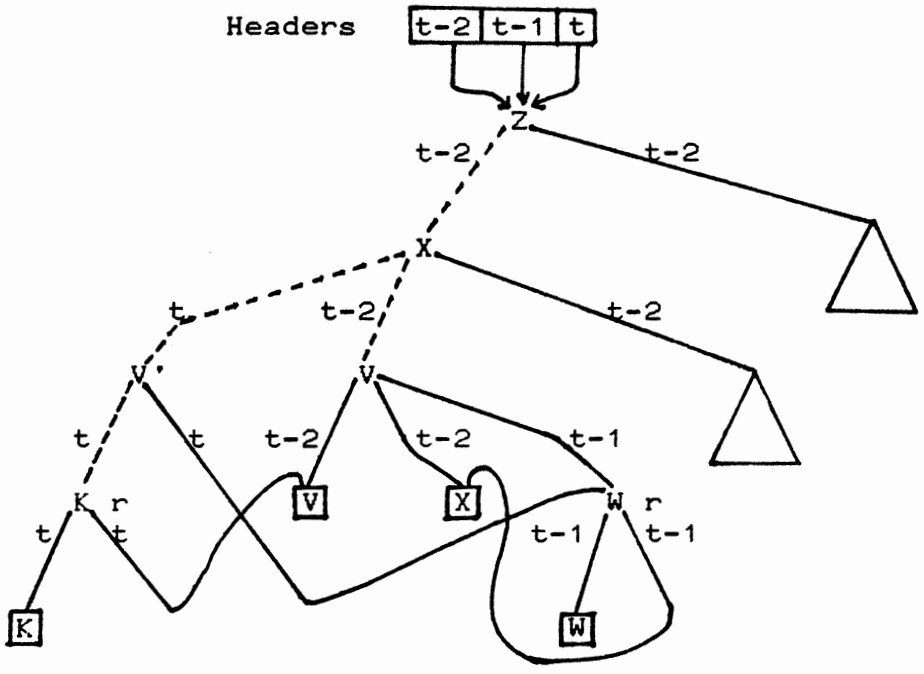


(b) The pointer to node K is inserted in the AUX_PTR of node V. The letter 'r' denotes a red node.

Figure 6. An Internal Node with a Free Slot.



(a) Node V has no free slot for the new pointer. "t-2" is read as "<= t-2 and < t-1".



(b) A copy of V, node V', is created with pointers pointing to node K and W.

Figure 7. An Internal Node without a Free Slot.

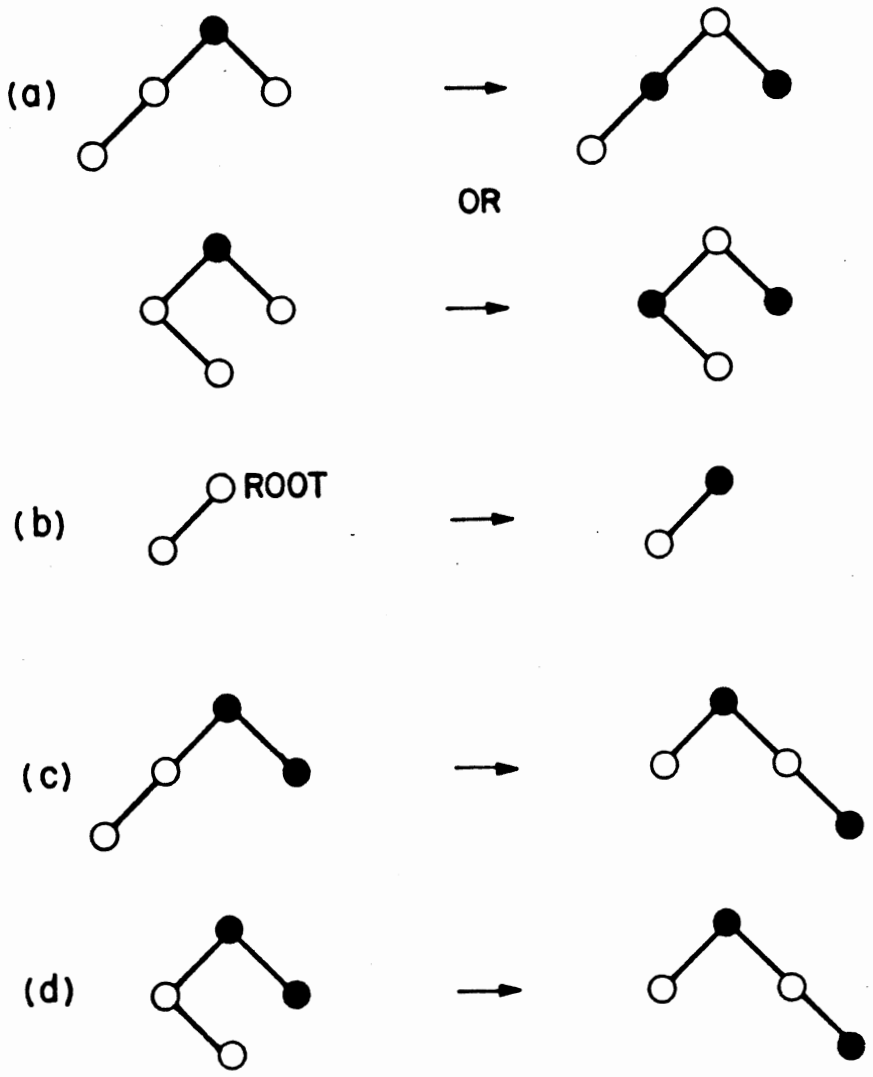


Figure 8. The Rebalancing Transformations in Red-Black Tree Insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black. In cases (c) and (d) the bottommost black node can be missing.

Sequential Deletion Algorithm

Deletion is the process of removing an item from the current tree. If the item to be deleted is found in an external node, it is deleted from the tree. Although the deleted item is no longer accessible in the current tree and the trees created in the future, it is still accessible in the trees created prior to the deletion time. Deletion is usually more complicated than insertion because deleting a node with two children from a binary search tree is done by first moving the node to the bottom of the tree, by swapping the item with its inorder predecessor or successor which has only one child, and then deleting it.

Fortunately, with our implementation of the red-black tree the items (data) are always stored in external nodes, so deleting an item from a red-black tree is much easier than deleting an item from binary search trees which do not use external nodes. To delete an item I from a red-black tree (Figure 9(a)), we first find the external node containing the item, then delete the item from the tree by replacing its parent with its sibling (Figure 9(b)). Note that if the parent contains the key I, then the internal and external nodes containing I are deleted; otherwise, the internal node containing I may still remain in the tree. This kind of deletion is called external deletion.

Our deletion algorithm for the persistent red-black trees is based on this external deletion technique. The deletion algorithm proceeds as follows :

(1) Use the above search procedure to locate the parent of the desired item I.

(2) Replace P(I), the parent of node I, by S(I), the sibling of node I. In other words, we store the pointer to S(I) in P(P(I)), the grandparent of I. We should examine the current state of node P(P(I)) so that the pointer to S(I) can be stored in the appropriate field and node. The three possible states for P(P(I)) and the appropriate actions are the same as node V in step (3) of the insertion algorithm, so we shall not repeat them here.

(3) If the replaced node P(I) is a red node, go to step (4). If the replacing node S(I) is a red node, then change its color to black and go to step (4). Otherwise, the black constraint is violated and appropriate actions are required to bring the tree back to balance. To eliminate the black constraint, we let the replacing node S(I) be the short node X and proceed with the following steps :

(a) If X has a black parent P(X) and a black sibling S(X) with two black children, then apply the transformation in Figure 10(a) and let node P(S) be the (new) short node X, and repeat step (a).

(b) If X has a red sibling S(X) (or symmetric variants), apply the transformation in Figure 10(b). Note that this transformation requires

a single rotation, so nodes involved in the pointer changes need to be copied if they do not have free slots for the new pointers. (See step 4(d) of insertion algorithm for more copying requirements).

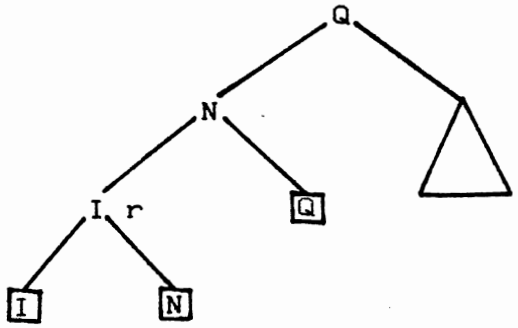
(c) If the right child of S(X) is a red node (or symmetric variant), apply the transformation in Figure 10(d) and go to step (4). Note that this transformation also requires a single rotation, so see step 4(d) of insertion algorithm for more copying requirements.

(d) If the left child of S(X) is a black node (or symmetric variant), apply the transformation in Figure 10(e); otherwise, apply the transformation in Figure 10(c). Note that transformation 10(e) requires a double rotation, so see step 4(e) of insertion algorithm for more copying requirements.

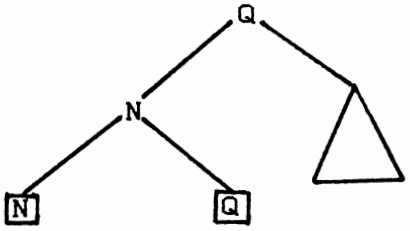
(4) End.

Note that transformations in Figure 10(a) and (b) are the only two nonterminal cases. The transformation in Figure 10(a) is used to move the shortness up the tree, one level at a time, until it no longer holds. Then we perform the transformation in Figure 10(b) if it applies, followed if necessary by one application of Figure 10(c), (d), or (e). The maximum number of rotations needed is three, that is, one application of transformation in Figure 10(b)

followed by one application of Figure 10(e). Thus, rebalancing after a deletion can be done in O(1) rotations.



(a) Initial red-black tree.



(b) The resulting tree.

Figure 9. An Example of External Deletion.

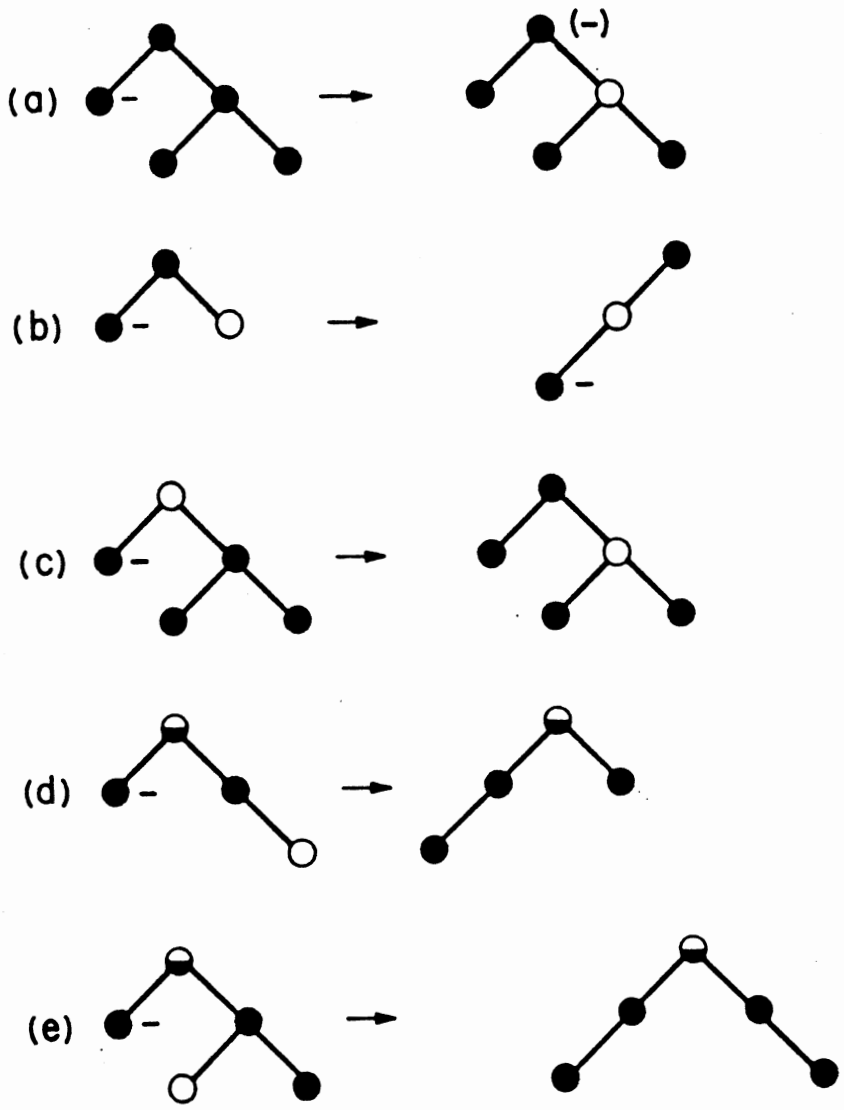


Figure 10. The Rebalancing Transformation in Red-Black Tree Deletion. The two ambiguous (half-solid) nodes in (d) have the same color, as do the two in (e). Minus signs denote short nodes. In (a), the top node after the transformation is short unless it is the root.

CHAPTER III

CONCURRENT OPERATIONS IN PERSISTENT RED-BLACK TREES

In this chapter, we present a concurrent system that can support multiple independent tasks concurrently accessing and updating the shared persistent red-black tree. A simple and yet efficient tree locking protocol is used for synchronizing the concurrently executing tasks. A top-down updating approach is utilized to enhance the system performance.

Computational Model

Our system is designed for implementations on multiprocessor systems with a shared global memory to which an unbounded number of tasks have access. However, it can also be implemented on uniprocessor systems that support multiprogramming without any modification. In addition, each task should have access to a local memory for its local processing.

We make no assumptions concerning the absolute or relative speeds at which the tasks are executed. It is possible that some tasks are much "slower" than others or that a task "goes to sleep" for a period of time and "wake

up" later. The reason for ignoring timing is that time-dependent bugs are extremely difficult to identify and correct. However, we assume that the tasks are reliable so that no task ever fails in the middle of the computation.

In order to maintain the overall correctness, concurrently executing tasks must communicate. Communication allows execution of one task to influence execution of another. We use shared global memory for interprocess communication. In our system, we allow tasks to set certain bits (or a bit) of a shared variable so that others can detect and then react accordingly. The effect of the interprocess communication will be much clearer when we discuss the implementation of our tree locking protocol in chapter 4.

Concurrency Control Mechanism

In a concurrent environment, it is essential to allow as many users to run their programs in parallel as possible to enhance the throughput. Unfortunately, if we allow many tasks to have unrestricted access to a shared structure, then it is impossible to guarantee the integrity of the data structure. Thus, the system must monitor, synchronize, and control the concurrently executing tasks so that overall correctness can be maintained. This process is called concurrency control.

We use locks for concurrency control. Other concurrency mechanisms include optimistic [10] and timestamping

[2] methods; however, locking is by far the most commonly used mechanism in practice. The basic idea is quite simple. When a task needs an assurance that some node that it is interested in will remain in a stable state during an operation, it places a lock on that node. The effect of the lock is to block other tasks out of the locked node, and thus in particular to prevent them from modifying it. Thus, the holder of a lock is able to carry out its processing with the certain knowledge that the node in question will not change in some unpredictable manner.

Tree Locking Protocol

Our tree locking protocol is similar to [5]. We use three types of locks : (i) read locks for readers to prevent writers from doing rotations; (ii) write-exclusion locks for writers to exclude other writers along the access path from the starred node, the highest node that may be modified in an update (insertion or deletion), to the place of update; and (iii) exclusive locks for writers to exclude readers from nodes modified during rotations. All locks only apply to internal nodes, since the access to an external node is controlled by its parent, which is an internal node. The relationships among these three types of locks can be summarized by means of a compatibility graph (Figure 11). An edge between any two nodes means that two different tasks may simultaneously hold these locks on the same node of a tree. Thus, a node may be read

locked by a number of readers while it is write-exclusion locked by a writer. However, if a writer holds an exclusive lock on a node, then no other tasks can hold any types of locks on the same node.

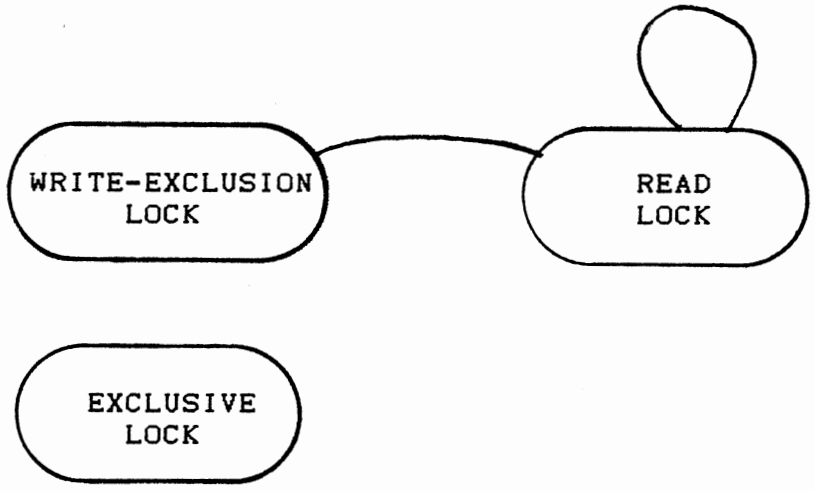


Figure 11. Compatibility Graph for Locks.

The following is a more detailed discussion on how these locks actually work together :

(1) A request from task T_i for a read lock on node X will be granted if no task holds the exclusive lock on X. If T_i fails, it will be put to "sleep" for a short period of time so that other tasks can use the CPU for some useful work. After waking up, T_i will be put back to a ready list for another try.

(2) A request from task T_i for a write-exclusion lock on node X will be granted if no task holds the

write-exclusion lock on X. If T_i fails, it too will be put on a sleep list for a short period of time. After waking up, it will be placed on a ready list for another try.

(3) A request from task T_i for an exclusive lock on node X will be granted if T_i already holds the write-exclusion lock on node X and no other task holds any read locks on X. Similarly, if T_i fails, it has to try later.

In the next chapter, we will discuss how a tree locking protocol is actually implemented on a UNIX system. For now we shall assume that a locking or unlocking operation is done in one indivisible step. That is, when a task is performing a lock operation, no other task is able to interfere with its processing.

Deadlocks

Locks have been widely used for synchronizing concurrently executing tasks, but not all locking protocols are free of deadlocks. A task is said to be in the state of deadlock if it is waiting for a particular event that will not occur. Figure 12 shows a deadlock involving two tasks T_1 and T_2. An arrow from a node X to a task T means that node X has been locked by task T, and an arrow from a task T to a node X means that task T is waiting to lock node X. This locks allocation graph (Figure 12) illustrates a deadlocked system : T_1 has a lock on node X_1 and needs a

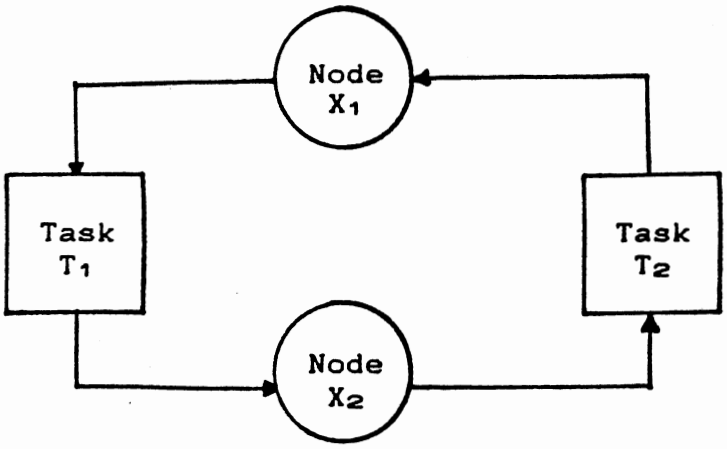


Figure 12. A Locks Allocation Graph.

lock on node X_2 to continue, T_2, on the other hand, has a lock on node X_2 and needs a lock on node X_1 to continue. Each task is waiting for the other to release the lock before it can proceed. This circular wait is the characteristic of a deadlocked system.

One way to prevent deadlocks from occurring is to deny the possibility of a circular wait. Our tree locking protocol does exactly that by enforcing the following rules :

(1) If a node X is not a header of a tree, then a

task T can lock node X only if it already has a lock on the parent of X.

(2) Once a task T releases a lock on a node X, it cannot subsequently obtain the same type of lock on node X again on the same access.

These rules imply that a task T can release a lock on a node X only after it has obtained the locks it needed on the descendants of X. This handshake between locking children and unlocking their parent is called lock coupling. Note that lock coupling implies that locks are obtained in header-to-leaf order. This leads to the key fact about our tree locking protocol : If task T_1 locks a node X before task T_2, then for every descendent V of X in the tree, if both T_1 and T_2 lock V, then T_1 locks V before T_2.

Now we want to show that our tree locking protocol is free of deadlocks by the following informal arguments. First, note that if a task T_1 is waiting for a header, it cannot be involved in a deadlock since it holds no locks and thus no tasks could be waiting for it. Now, suppose task T_1 is waiting for a lock currently held by task T_2 on node X other than the header. Since T_2 must release the lock on node X before T_1 can lock it and T_2 is not allowed to lock node X again on the same access, no circular wait can ever exist. Thus our tree locking protocol is not prone to deadlocks.

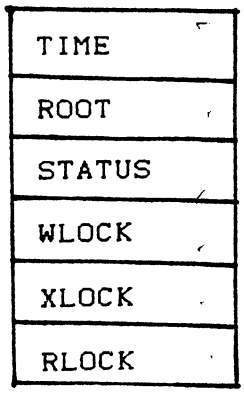
Data Structure

The data structure consists of a forest (a collection of red-black trees), a list of free headers, a list of free internal nodes, and a list of free external nodes. In order to implement our tree locking protocol, a number of lock and flag fields are added to each header and node. Figures 13(a), (b), and (c) show the new structures for the header, internal node, and external node, respectively.

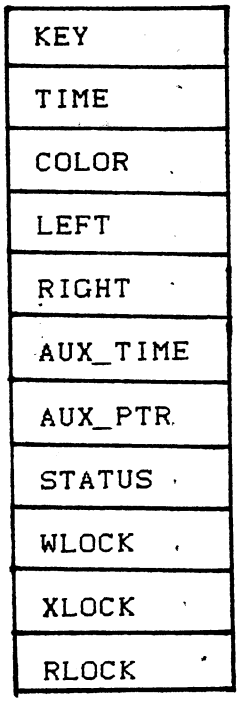
STATUS is a flag indicating whether a header or internal node is free or in use. WLOCK, XLOCK, and RLOCK are fields for write-exclusion, exclusive, and read locks, respectively. All other fields are defined as in chapter II. In the next chapter we will discuss how some of these locks can be combined into a single field for efficient implementation.

Concurrent Search Algorithm

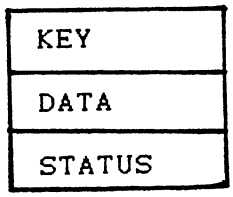
Searching is the process of locating an item associated with a particular time stamp. Although we use the same definition as the sequential search algorithm, this algorithm is not utilized by concurrent insertion and deletion for locating the place of update because different sets of rules are required for top-down insertion and deletion. Furthermore, the result returned by this search may be outdated before it has actually been used. Thus, this search algorithm is also called a weak search. Weak search should not be used if up-to-date information is



(a) Header Structure



(b) Internal Node Structure



(c) External Node Structure

Figure 13. Record Structures for Header and Nodes.

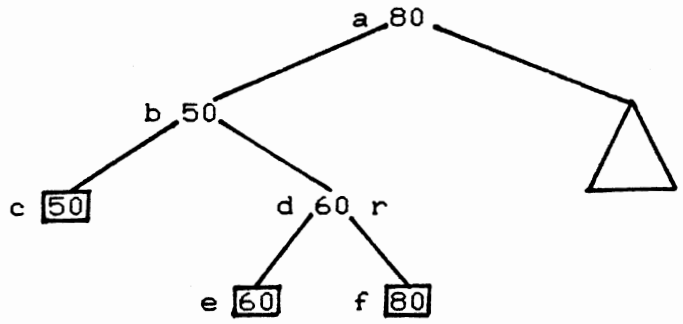
essential.

Although search tasks do not modify the shared data structure, the sequential search algorithm described in the previous chapter may not work consistently in a concurrent environment. When many types of operations such as insertions and deletions are allowed to work on the tree simultaneously with searching, the result returned by a search operation may not be correct if nodes on the search path have been modified. To see this, consider the following example : Task T_1 wants to search for item 80 in the red-black tree in Figure 14(a) and it is currently at node 50, and task T_2 has just inserted node 70 into the tree (Figure 14(b)) and it is about to do a rotation. For simplicity, we shall assume that the initial tree is created in the current time unit and all the above operations are to perform at current time. (Figure 14(c) shows the resulting tree after a single left rotation).

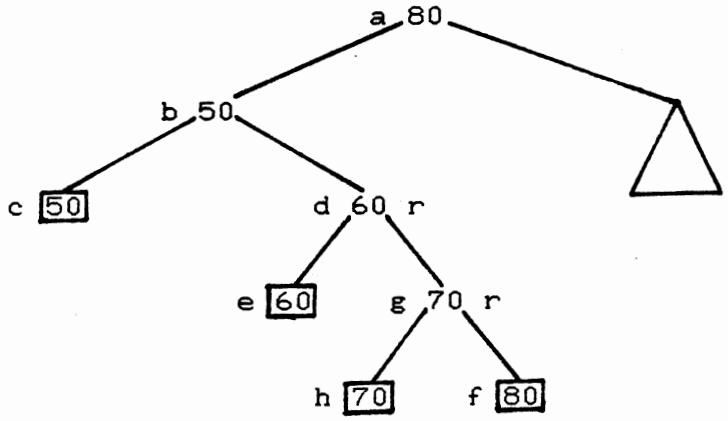
The following is one of the possible interleaved execution sequences for T_1 and T_2.

Steps	Task T_1	Task T_2
1	current_ptr = b	
2		b->RIGHT = e
3		d->LEFT = b
4	80 > current_ptr->key	
5	current_ptr = e	
6		a->left = d
7	80 != current_ptr->KEY	
8	"node not found"	

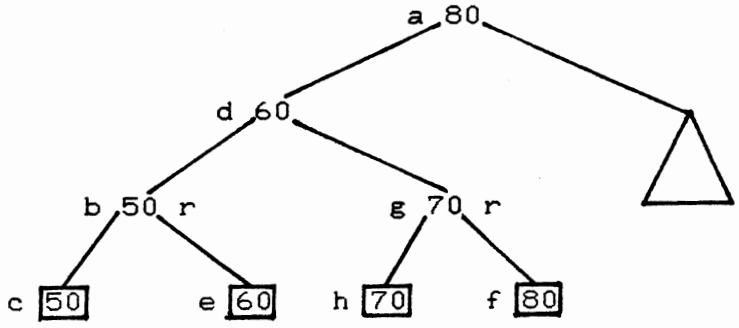
According to the above search sequence, the item 80 is



(a) A red-black tree. The letter on the left of a node denotes the address and the letter on the right denotes the color, 'r' for red and ' ' for black. The triangular node denotes a subtree and the square nodes are external nodes.



(b) After inserting node 70.



(c) After a single left rotation.

Figure 14. An Example of Incorrect Branching.

not in the red-black tree, but there is indeed an external node containing the item 80 in the tree. This incorrect result is obviously caused by the actions of T_2. More precisely, it is caused by the pointer changes during the rotation. To prevent other writers such as T_2 from interfering, readers need to place a read lock on each node on the access path before doing anything else. Since search tasks do not modify the shared data structure, we can therefore allow many readers to share a node with a writer, which is not doing a rotation, at the same time.

The concurrent search algorithm is basically an enhanced version of the sequential search. The only thing that is added to the concurrent version is the placement of the read locks on the header and the nodes which lie on the search path. The concurrent search proceeds as follows. Initially, we place a read lock on the header associated with the search time, then a read lock is placed on the root of the tree and the read lock on the header is then released. Note that a read lock can be released only after a read lock has been placed on one of its children. This locking and unlocking process is repeated down along the access path until an external node is reached. The external node either contains the item or something else. The read lock on the parent of the external node is eventually released. (C code for the concurrent search algorithm is listed in the Appendix).

Searching can be performed in all versions of the

persistent search tree. If the queries are done in trees which contain only the dead nodes, then readers can never be blocked. If, on the other hand, readers are searching in the current tree, the only time they can ever be blocked is when they are trying to lock nodes involved in rotations. As we stated earlier, rebalancing a red-black tree after an update only takes O(1) rotations, so readers in our system are rarely blocked.

Concurrent Updating Algorithms

In a sequential environment, an updating algorithm can be done in either bottom-up or top-down fashion. A bottom-up algorithm is usually preferred because it is easier to implement and the lengthy bottom-up pass cannot interfere with or hold up any other tasks since the executing task is the only task operating in the tree. However, in a concurrent environment, many tasks need to operate in the shared data structure simultaneously, so a bottom-up updating algorithm can no longer be used. Top-down updating, on the other hand, is able to eliminate the lengthy bottom-up pass by identifying the highest node, say H, that may be modified in an update in just one top-down pass. Once node H is determined, a simple locking protocol can be used to block out other tasks from the subtree rooted at node H and free the rest of the tree to any other updating tasks. This isolation of one updating operation from the effects of all other updating operations in the

tree structure ensures the overall correctness of the system.

As we stated in previous chapters, one way to make an ephemeral search tree persistent is by employing the path copying method. Path copying requires the entire access path to be copied at each update time, so it is difficult to achieve a high degree of concurrency even if we incorporate the top-down updating algorithm into the path copying. Node copying, on the other hand, is specially designed to avoid copying the entire access path at each time an update occurs. In particular, most of the copying occurs near the bottom of the tree and at each time only a small number of nodes is copied. Thus, top-down updating of a persistent red-black tree with limited node copying should provide a high degree of concurrency.

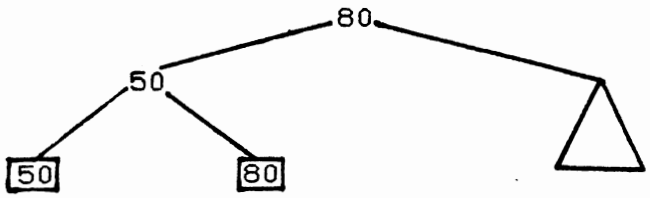
Concurrent Insertion Algorithm

Insertion is the process of adding an item to the shared persistent red-black tree in the current time unit. Now, we shall use a simple example to illustrate why the sequential insertion algorithm may not work consistently in a concurrent environment. To see this, consider the following example : Suppose task T_1 wants to insert node 60 and task T_2 wants to insert node 70 into the tree in Figure 15(a) at the same. For simplicity, we shall assume that the tree in Figure 15(a) is created in the current time unit. If the sequential insertion algorithm described in

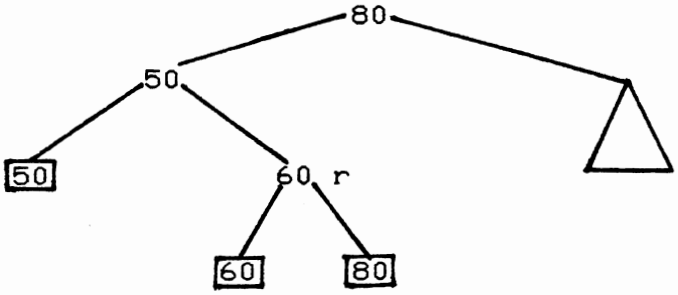
the previous chapter is used, the resulting tree could be in Figure 15(a), 15(b), or 15(c) depending on the interleaved executions of T_1 and T_2.

However, we can make some minor changes to the sequential insertion algorithm in such a way that it will work correctly in a concurrent environment. Similar to the search task, an insertion task needs to place locks of some kind on nodes that it is interested in so that no other tasks can modify them. For an insertion, write-exclusion locks should be used on nodes along the access path since we are only interested in blocking out other writers but not the readers. After inserting a new item, rotations may be necessary to bring the search tree back to balance. In order to prevent readers from getting confused during rotations and branching to the incorrect children, a small number of exclusive locks should be used in nodes involved in the rotations.

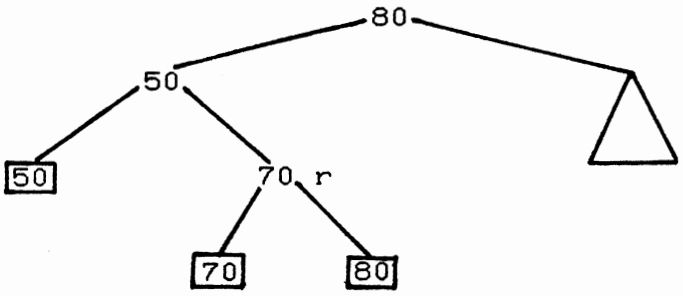
The concurrent insertion algorithm proceeds as follows. Initially we let the current header be the starred node, the highest node that may be modified during an insertion, and place a write-exclusion lock on it, then we place another write-exclusion lock on the root node and proceed from the root node down along the access path. Note that write-exclusion locks are placed from the header to the parent of an external node, but some of these locks can be released early if the following conditions are met. When arriving at a black node, say X, that has a black



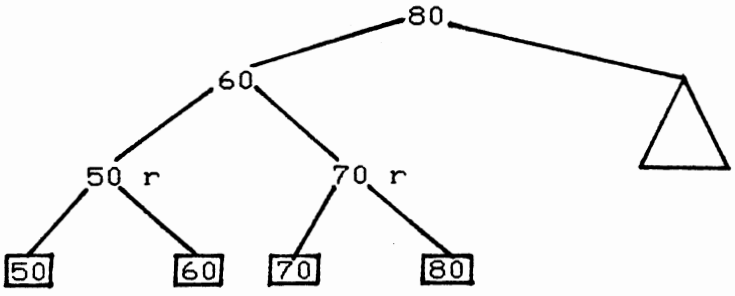
(a) A red-black tree.



(b) After inserting items 60 and 70.



(c) After inserting items 60 and 70.



(d) After inserting items 60 and 70.

Figure 15. An Example of Incorrect Insertions.

child, we release the write-exclusion locks from the starred node down to the lowest node, say Y, which is not lower than the parent of X and has a free slot for a new pointer, and then let Y be the (new) starred node. This process is repeated until an external node is reached. If the external node does not contain the new item, let V be the parent of the external node and continue from step (2) to step (5) of the sequential insertion algorithm in chapter II; otherwise, the insertion fails.

Note that in the cases (Figures 8(c) and (d)) where a rotation is necessary, we need to convert some write-exclusion locks to exclusive locks before performing the rotation to prevent readers from branching to the incorrect children. More precisely, a single rotation requires exclusive locks on the critical node (node P(S) in step(4) of the sequential algorithm) and its parent (node P(P(S))) and a double rotation requires an additional exclusive lock on the child of the critical node which lies on the search path (node S). All locks are released after the update. (C code for the concurrent insertion algorithm is listed in the Appendix).

Concurrent Deletion Algorithm

Deletion is the process of removing an existing item from the shared persistent search tree. If an item is removed from the persistent search tree, it is inaccessible from the current tree and the trees created in the future,

but it is still accessible from the trees created before the deletion time.

Concurrent deletion is similar to concurrent insertion but slightly more complicated. It also requires the placement of write-exclusion locks along the search path to block out other writers. The concurrent deletion algorithm proceeds as follows. Initially, we let the current header be the starred node, the highest node that may be modified during a deletion, and place a write-exclusion lock on it, then place another write-exclusion lock on the root node and proceed from the root node down along the search path. When we arrive at a red node X that has a red child, we release the write-exclusion lock from the starred node to the lowest node, say Y, which is not lower than the parent of X and has a free slot for a new pointer, and then let Y be the (new) starred node. This process is repeated down along the access path until an external node is reached. If the external node contains the item to be deleted, let node P(I) be the parent of the external node and continue from step (2) to step (4) of the sequential deletion algorithm in chapter II; otherwise, the deletion fails.

Similarly, in the cases (Figures 10(c), (d) and (e)) where rotation is necessary, we need to convert a small number of write-exclusion locks to exclusive locks. These nodes are the same as in the previous section and we shall not repeat them here. All locks are released after the update. (C code for the concurrent deletion algorithm is

listed in the Appendix).

CHAPTER IV

THE IMPLEMENTATION OF A TREE LOCKING PROTOCOL AND SIMULATION RESULTS

As stated in the previous chapter, our tree locking protocol uses three types of locks for different actions to maximize the concurrency. In this chapter we shall discuss how these three types of locks can be implemented on a Perkin Elmer 3230 running under UNIX System V operating system. The problem associated with uncontrolled access to shared resources and its solutions are presented.

The Implementation of a Tree Locking Protocol

Since a write-exclusion lock is used by only one writer to block out all other writers from a header or internal node, we can conveniently use a binary integer, say WLOCK, to represent a write-exclusion lock. If the WLOCK contains a value of 0, it is not locked; otherwise, it is locked.

The read and exclusive locks can be represented by a counter called RXLOCK, since a number of readers may simultaneously read lock a header or internal node and the read locks are not compatible with the exclusive lock. The

counter is incremented by 1 when a read lock has been granted and it is decremented by 1 when a read lock has been released. A positive count, say n, of the counter means that n read locks have been issued; a negative count means that an exclusive lock has been issued; a zero count means that no task has held any read or exclusive lock on the node. Alternatively, we can also use a field with n+1 bits, say NRXLOCK, to denote the read and exclusive locks, where n is the number of tasks allowed to operate on the shared structure. The i^{th} bit of NRXLOCK is set if task T_i holds a read lock and the most significant bit is set if the exclusive lock has been locked. Although the latter uses more space, it has the advantage of maintaining the information of the holders of read locks and thus it can prevent tasks from releasing read locks that they have never locked. Therefore, we shall use the latter for our implementation.

With the above implementation, a read lock clearly has a higher priority than an exclusive lock. We can, of course, use an extra field, say XFLAG, to indicate whether a request for an exclusive lock has been issued so that a request for a read lock will not jump ahead of the waiting request for the exclusive lock. Although this method is quite fair, we shall not use it because it is quite difficult to implement efficiently and cleanly in a concurrent environment.

Mutual Exclusion and Critical Section

In the previous chapter, we assumed all the "lock" and "unlock" operations are atomic actions -- that is, these operations are performed in one indivisible step. This assumption is essential in a concurrent environment because interference from any other task may corrupt the integrity of the shared variables. To see this, consider the following example : Suppose initially that the RXLOCK of header x is equal to one, written as x.RXLOCK = 1, that both task T_i and T_j attempt to read lock header x concurrently. The codes which implement the locking for both tasks can be written as follows:

T_i: x.RXLOCK = x.RXLOCK + 1; T_j: x.RXLOCK = x.RXLOCK + 1;

It would seem reasonable to expect the final value of x.RXLOCK, after T_i and T_j have executed concurrently, to be 3. Unfortunately, this will not always be the case because assignment statements are not generally implemented as indivisible operations. It is possible that the above assignment statements might be implemented as a sequence of the following three instructions:

(1) Load a register with the value of x.RXLOCK.

(2) Add a value to x.RXLOCK.

(3) Store the result in x.RXLOCK.

Now, suppose T_i executes the load and add instructions, thus leaving 2 in an accumulator. Then T_i loses the CPU

(through a quantum expiration) to T_J. T_J now executes all three instructions, thus setting x.RXLOCK to 2. T_J loses the CPU to T_I which then continues by executing the store instruction, that is, storing 2 into x.RXLOCK. Because of uncontrolled access to the shared variable x.RXLOCK, the final value of x.RXLOCK = 2 is obviously incorrect.

The key to preventing such a problem is to find some way to prohibit more than one task from reading and writing the shared variable at the same time. In other words, we need to give each task exclusive access to x.RXLOCK. While one task increments the shared variable, all other tasks desiring to do so at the same moment should be kept waiting; when that task is finished accessing the shared variable, one of the tasks waiting to do so should be allowed to proceed. In this fashion, each process accessing the shared data excludes all others from doing so simultaneously. This is called mutual exclusion.

So for the same example, we can rewrite it as follows:

```
T_I: enter_mutual_exclusion;        T_J: enter_mutual_exclusion;
     x.RXLOCK = x.RXLOCK + 1;             x.RXLOCK = x.RXLOCK + 1;
     exit_mutual_exclusion;               exit_mutual_exclusion;
```

The codes between the statements enter_mutual_exclusion and exit_mutual_exclusion are called the critical sections. In other words, a critical section is a piece of code that performs the actual accessing or updating of a shared variable. While one task is in the critical section, the

other tasks attempting to enter the same critical section have to wait and other tasks may continue their executions outside the critical section. Enforcing mutual exclusion is one of the keys in maintaining the overall correctness of a concurrent system. Many solutions have been proposed, but we shall discuss only those related to our implementation.

Semaphores

The concept of a semaphore was first introduced by Dijkstra [4]. A semaphore S is a protected variable that, apart from initialization, can be accessed only by the atomic operations P and V. The classic P and V operations on S, written as P(S) and V(S), respectively, are as follows:

```
P(S) : if (S > 0)
         S = S - 1;   /* it is safe to enter crit. sect.*/
       else
         "append the requesting task to a waiting queue"


V(S) : if ("the waiting queue is not empty")
         "let one of the waiting tasks enter crit. sect."
       else
         S = S + 1;
```

P and V operations are commonly implemented in the kernel where the process state switching is controlled so that each P(S) or V(S) can be done in one indivisible operation. Rather than discussing the implementation of

P(S) and V(S), let us show how a semaphore S_x can be used to ensure the exclusive access to the RXLOCK of header x. (We assume S_x has a value of 1 initially).

```
T1: P(Sx);                          Tj: P(Sx);
    x.RXLOCK = x.RXLOCK + 1;            x.RXLOCK = x.RXLOCK +1;
    V(Sx);                              V(Sx);
```

With the semaphore S_x and its P and V operations, the x.RXLOCK can no longer be corrupted in any circumstances. If we use the same scenario as before, even though T_1 loses the CPU to T_j and leaving the value 2 in the accumulator, T_j will not be able to pass the operation P(S_x) since the S_x has a value of 0. Thus once T_1 regains the service of CPU, it can save the value of 2 in x.RXLOCK and then execute V(S_x) to allow T_j to continue. Note that a semaphore count of zero means that there is a task currently accessing the shared variable; a semaphore count of 1, on the other hand, means that no task is accessing the shared variable and thus it is safe to enter the critical section. The kind of semaphore is called a binary semaphore.

Test-and-Set Instruction

Many machines, especially those designed with multiple processors in mind, provide special hardware instructions that allow one to test and modify the content of a variable in one memory cycle. This kind of instructions, often called test-and-set, once initialed will complete all of

these functions without interruption.

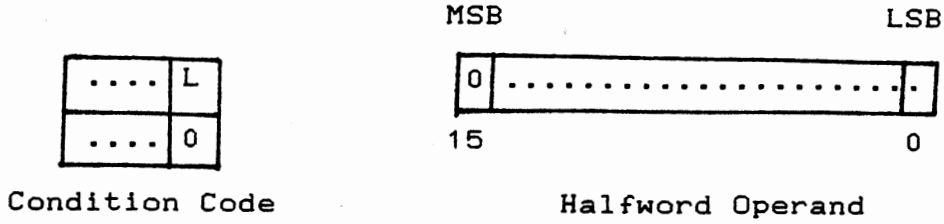
Our working machine, Perkin Elmer 3230, has a similar instruction called TS that performs the following functions on a halfword operand in just one memory cycle :

(1) Read the halfword operand from the memory.

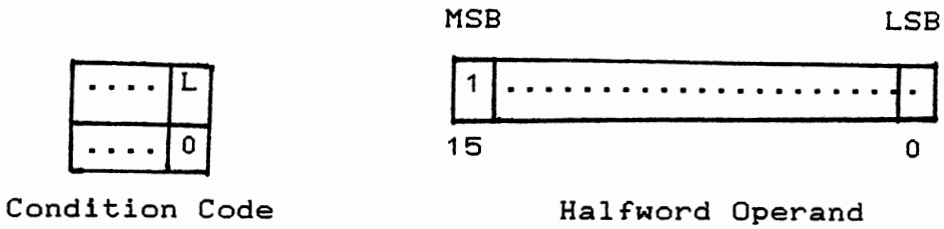
(2) Set the condition code to the value of the most significant bit (MSB) of the operand. (Note that the condition code reflects the state of the MSB at the time of the memory read).

(3) Set the MSB to 1 and store the operand in memory with no change to all other bits.

Figures 16 and 17 are two examples showing the effects of the TS instruction.

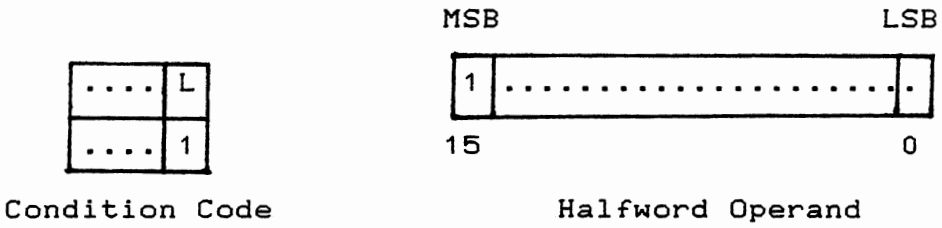


(a) Before executing TS instruction.

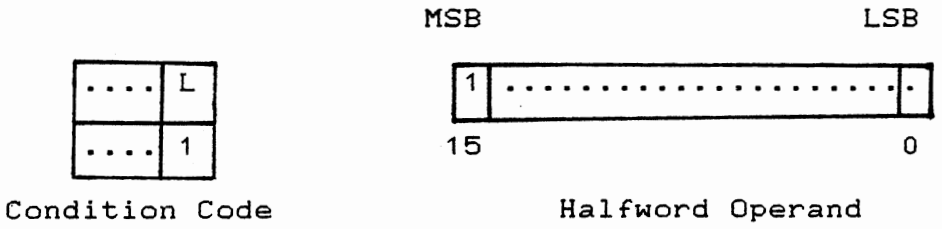


(b) After executing TS instruction.

Figure 16. Executing TS instruction with the MSB of the operand equal to 0.



(a) Before executing TS instruction.



(b) After executing TS instruction.

Figure 17. Executing TS instruction with the MSB of the operand equal to 1.

Now, we want to show how TS instruction can be used to enforce mutual exclusion. First, we need a function, written in assembly language, which executes the TS instruction with a halfword operand and returns the condition code to the caller. We shall call this function tas( ) and it can be invoked from a C program as follow:

```
condition_code = tas(&halfword_variable);
```

The complete listing of this function is given in the Appendix and we shall not repeat it here. Next, we need to define the lock fields with halfword in length and no sign

stored in the most significant bits. All these can be accomplished by the following C declaration:

```
unsigned short   wlock,
                 rxlock;
```

On the PE 3230, an unsigned short is stored in two bytes without sign.

Now, we want to show how tas( ) can be used to implement a write-exclusion lock. Consider the following C code:

```
while ( tas( &wlock ) )
     sleep( 1 );
```

Note that the address of wlock is passed to function tas( ). A value of 0 is returned from function tas( ) if wlock has not been locked by another task and it is now locked by the caller. The caller can then proceed with its processing on the locked node. A value of 1 is returned by tas( ) if wlock has been locked by another task and thus the caller has to try until it has the lock. In order to prevent the caller from continuously calling tas( ), we have decided to put the calling task to sleep so that another task can use the CPU service for some useful work. Continuously testing a variable waiting for some value to appear is called busy waiting.

Unlocking a write-exclusion lock can be done by storing a value of 0 in wlock. No special instruction is

needed.

The implementation of read and exclusive locks is slightly more complicated than the above implementation. The following C code illustrates the implementation of exclusive lock:

```
while ( !done ) {            /* initially done = 0 */
  while ( tas( &rxlock ) )
     sleep( 1 );
  if ( rxlock == 0x8000 )    /* any read locks ? */
     done = 1;               /*  No -- succeed */
  else
     rxlock = rxlock & 0x7fff; /* Yes -- try again */
}
```

Note that an exclusive lock is obtained by first setting the MSB of rxlock and then checking for the presence of the read lock. If any reader holds a read lock on the node (Figure 18(a)), the task fails to secure the exclusive lock on the node and it has to try again later; otherwise (Figure 18(b)), it succeeds in acquiring the exclusive lock.

Unlocking an exclusive lock can be done by storing a value of 0 in rxlock. No special instruction is necessary.

Finally, we want to show the implementation of read locks by employing the function tas( ). Consider the following C code:

```
while ( tas( &rxlock ) )
      sleep( 1 );
rxlock = rxlock | 0x0001;    /* assume 1st task */
rxlock = rxlock & 0x7fff;    /* reset MSB to 0 */
```

A read lock is obtained by first setting the MSB of rxlock and then setting the taskth bit of rxlock. Eventually, the MSB is reset so that other readers may subsequently read lock the same node. With this implementation, a read lock has a higher priority than an exclusive lock, since a request for an exclusive lock is denied whenever there is a read lock on the node and a request for a read lock, on the other hand, is granted even if there is an appending request for an exclusive lock.

Unlocking a read lock is similar to locking it. The C code is as follows:

```
while ( tas( &rxlock ) )
      sleep( 1 );
rxlock = rxlock & 0xfffe;    /* assume 1st task */
rxlock = rxlock & 0x7fff;    /* reset MSB to 0 */
```

Note that when an exclusive lock has been placed on a node, the MSB of rxlock is always set until the exclusive lock has been released. Thus, the conflicting relationship between read and exclusive locks can be maintained. Also note that the above C codes are simplified versions of the actual codes. The simplified versions are used to enhance the readability and convey a few important ideas. The complete listing of these codes can be found in the Appendix.

Enforcing mutual exclusion by the test-and-set instruction can suffer from starvation. However, it is not likely

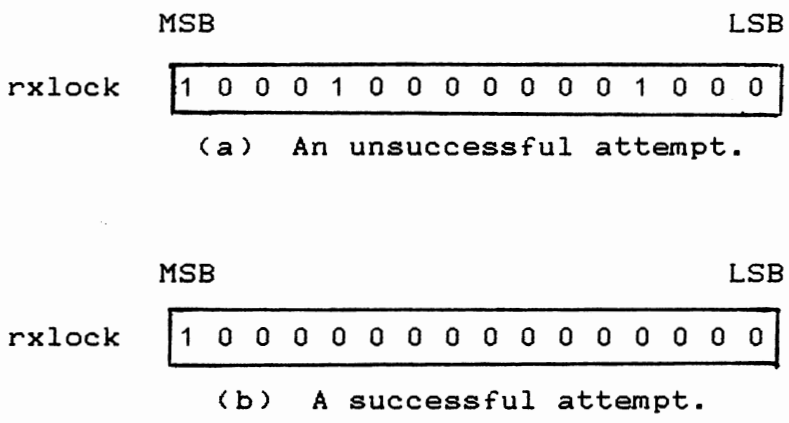


Figure 18. An Example of a Request for Exclusive Lock.

to occur on the multiprocessor systems. Furthermore, with persistent search structure, we can limit the maximum number of attempts (by any task) to n times by maintaining a counter for the number of transactions in a particular time stamp, where n is the number of transactions in a particular time unit. The counter is incremented by 1 when a transaction is added to the system in the current time unit; it is decremented by 1 when a task finishes its transaction. The current time is not allowed to increase if there are unfinished transactions in the current time unit. Of course, no transaction is allowed to be added to the system while the system is waiting to increase the current time stamp.

Simulation Results

The concurrent algorithms described in chapter III have been implemented in a simulation program, and several

experiments have been carried out. Analyzing the behavior of concurrent algorithms is quite difficult because there are many possible execution sequences or interleavings. Thus even with a fixed input, there are a great number of possible interleavings and consequently a number of different behaviors. The approach taken here is to run the simulator a number of times and then compute averages.

Overall, we found that our algorithms support a fairly high degree of concurrency. For example, given a tree of 500 nodes and four independent tasks, each executing 10 operations (4 searches, 3 insertions, and 3 deletions) per time stamp with random keys for a total of 10 time stamps, about 0.23 percent (1.76/750) of the total requests for write-exclusion locks and 0.033 percent (0.13/400) of the total requests for read locks were delayed. None of the requests for exclusive locks were blocked.

When the number of operations per time stamp increases from 10 to 20 (7 searches, 7 insertions, and 6 deletions) and everything else remains the same, only about 0.15 percent (2.50/1650) and 0.028 percent (0.20/705) of the total requests for write-exclusion and read locks were delayed, respectively.

When the size of the tree increases to about 1000 nodes, the numbers do not fluctuate too much from the above. We also found that most of the delays occurs at the header and root levels. This finding is not surprising since locks are obtained in header-to-leaf order and thus

all tasks accessing the tree must first hold the locks to the header and then the root node before they can proceed down along the access path.

CHAPTER V

CONCLUSIONS

In this thesis, we present a concurrent system that can support multiple independent tasks concurrently accessing and updating shared persistent red-black trees without destroying the integrity of the shared structure and without causing deadlocks. The high level of concurrency is achieved by using three types of locks, namely read lock, write-exclusion lock, and exclusive lock, to guard against different actions. Locks are used whenever a task needs assurance that nodes which are accessed will not change in some unpredictable manner. With our tree locking protocol some locks can be released before the holder finishes its operation and subsequently other tasks can acquire these locks and proceed with their executions. As a result, many tasks can be operating in the shared search tree at the same time.

The limited node copying method is used because it is suitable for concurrent implementations. Unlike the path copying method, limited node copying requires a node to be copied only if it has no free slot for a new pointer to be added to it; moreover, most of the copying occurs near the bottom of the tree. Thus, we can easily incorporate the

limited node copying method into a top-down updating algorithm. A top-down updating algorithm eliminates the lengthy bottom-up pass of a bottom-up algorithm by identifying the highest node H, which may be modified in an update, during the top-down pass. Once node H is determined, a simple locking protocol can be used to protect the subtree rooted at node H and free the rest of the tree to other tasks. Thus, our implementation provides a high degree of concurrency.

Since our tests are conducted on a uniprocessor system which supports multiprogramming, we have no way of predicting the performance of our concurrent algorithms on the multiprocessor systems. Thus, the performance of our algorithms on multiprocessor systems requires further study.

The major draw back of our updating algorithms is blocking out too many updating tasks from the current header and root. Thus, developing a method that allows many writers to share a header or a root can be a challenging topic. It is left for future study.

REFERENCES

[1] Bayer, R. Symmetric binary B-trees: data structure and maintenance algorithms. Acta Informatica 1, 4(1972), 290-306.

[2] Bernstein, P. A., and Goodman N. Timestamp-based algorithms for concurrency control in distributed database systems. Proc. 6th International Conference on Very Large Data Bases, (1980).

[3] Deitel, H. M. An Introduction to Operating Systems. Addison-Wesley, Reading, Mass., 1984.

[4] Dijkstra, E. W. Solution of a problem in concurrent programming control. Comm. ACM 8, 9(1965), 569.

[5] Ellis, C. Concurrent search and insertion in AVL trees. IEEE Trans. Comput. C-29, 9(1980), 811-817.

[6] Ellis, C. Concurrent search and insertion in 2-3 trees. Acta Informatica, 14(1980), 63-86.

[7] Fisher, D. D. Persistent search trees. Lecture Notes on Data and Storage Structures, Department of Computing & Information Sciences, Okla. State Univ., (1986).

[8] Krijnen, T., and Meertens, L. G. L. T. Making B-trees works for B. IW 219/83, The Mathematical Centre, Amsterdam, The Netherlands, (1983).

[9] Kung, H. T., and Lehman, P. L. Concurrent manipulation of binary search trees. ACM Trans. on Database Syst. 5, 3(1980), 354-382.

[10] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. ACM Trans. on Database Syst. 6, 2(1981), 213-226.

[11] Manber, U. Concurrent maintenance of binary search trees. IEEE Trans. Software Eng. SE-10, 6(1984), 777-784.

[12] Manber, U., and Ladner, R. E. Concurrency control in a dynamic search structure. ACM Trans. on Database Syst. 9, 3(1984), 439-455.

[13] Myers, E. W. AVL dags. Tech. Rept. 82-9, Department of Computer Science, Univ. of Arizona, Tucson, (1982).

[14] Myers, E. W. Efficient applicative data types. In Conference Record Eleventh Annual ACM symposium on Principles of Programming Languages, (Salt Lake City, Utah, Jan. 15-18, 1984), 66-75.

[15] Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. ACM Trans. Prog. Lang. Syst., 5(1983), 449-477.

[16] Sarnak, N., and Tarjan, R. E. Planar point location using persistent search trees. Comm. ACM 29, 7(1986), 669-679.

[17] Swart, G. Efficient algorithms for computing geometric intersections. Tech. Rept. #85-01-02, Department of Computer Science, Univ. of Washington, Seattle, (1985).

[18] Tarjan, R. E. Amortized computational complexity. SIAM J. Alg. Disc. Math. 6, 2(1985), 306-318.

[19] Tarjan, R. E. Efficient top-down updating of red-black trees. Tech. Rept. CS-TR-006-85, Princeton Univ., (1985).

[20] Tarjan, R. E. Updating a balanced search trees in O(1) rotations. Inform. Process Lett. 16, 5(1983), 253-257.

[21] Verma, V., and Lu, H. A new approach to version management for databases. Proceedings AFIPS 1987 NCC, vol. 56, AFIPS Press, Arlington, Va., 645-651.

APPENDIX

SIMULATION PROGRAM

```
/*-------------------------------------------------------------------------*/
/* pstdef.h - definitions for concurrent persistent red-black trees.       */
/*-------------------------------------------------------------------------*/

#define  ERROR       -1         /* standard return values */
#define  OK          1
#define  FAIL        0
#define  NO          0
#define  YES         1

#define  NULL         0         /* initial values */
#define  NULLKEY     -1
#define  NULLTIME    0xff
#define  NULLID      -1
#define  INTERVAL    1

#define  STK_OVFL    100       /* flags for error handler*/
#define  OUT_NODE    101
#define  OUT_HDR     102

#define  RED         0x8001    /* node colors */
#define  BLACK       0x8002
#define  FREE        0x0000    /* node/header status */
#define  USED        0x8000
#define  UNLOCK      0x0000    /* lock and unlock */
#define  LOCK        0x8000
#define  XLOCK       0x8000

#define  STACK_SIZE  50        /* size of stack */

#define  MAX_NODE    4600      /* # of nodes in shared memory */
#define  MAX_USER    10        /* max. # of tasks */
#define  MAX_KEY     0xfffff   /* max. key */
#define  MAX_HEADER  30        /* max. # of header nodes */
#define  SHMKEY1     0x1111    /* shared memory key */
#define  SHMKEY2     0x2222    /* shared header key */
#define  SHMKEY3     0x3333    /* shared system info. block key */
```

```
/*-------------------------------------------------------------------------*/
/* pstext.h - header file contains external variables.                     */
/*-------------------------------------------------------------------------*/

extern HDR_PTR         header;  /* header pointer */
extern short           ttime;   /* transaction time */
extern unsigned short  task,    /* current task id */
                       flag;    /* a flag */
```

```
/*-------------------------------------------------------------------------*/
/* pstmac.h - macros for concurrent persistent red-black tree.             */
/*-------------------------------------------------------------------------*/

/* is p points to an internal node ? */
#define IS_INTERNAL(p)    ( ( ( p != NULL ) && ( p->left != NULL ) &&  \
                            ( p->right != NULL ) )  ? ( 1 )  :  ( 0 ) )

/* is p points to an external node ? */
#define IS_EXTERNAL(p)    ( ( ( p != NULL ) && ( p->left == NULL ) &&  \
                            ( p->right == NULL ) )  ? ( 1 )  :  ( 0 ) )

/* is p points to a red node ? */
#define IS_RED(p)         ( ( ( p != NULL ) && ( p->color == RED ) ) \
                            ? ( 1 )  :  ( 0 ) )

/* is p points to a black node ? */
#define IS_BLK(p)         ( ( ( p != NULL ) && ( p->color == BLACK ) ) \
                            ? ( 1 )  :  ( 0 ) )

/* is p points to a node that has 1 or more red sons ? */
#define HAS_1_RED_SON(p) ( ( ( IS_RED( p->left ) ) || ( IS_RED( p->right ) ) )\
                            ? ( 1 )  :  ( 0 ) )

/* is p points to a node that has 2 black sons ? */
#define HAS_2_BLK_SON(p) ( ( ( IS_BLK( p->left ) ) && ( IS_BLK( p->right ) ) )\
                            ? ( 1 )  :  ( 0 ) )

/* is p points to a node that uses its auxiliary pointer ? */
#define AUX_USED(p)       ( ( ( p != NULL )  &&  (p->aux_ptr != NULL ) ) \
                            ? ( 1 )  :  ( 0 ) )

/* is p points to a node that does not use its auxiliary pointer ? */
#define AUX_FREE(p)       ( ( ( p != NULL )  &&  ( p->aux_ptr == NULL ) ) \
                            ? ( 1 )  :  ( 0 ) )

/* are these two given keys different ? */
#define KEYDIF(p,nkey)    ( ( p->key != nkey )  ? ( 1 )  : ( 0 ) )
```

```
/*-------------------------------------------------------------------------*/
/* pstrec.h - declarations of record structures for node, header, and system */
/*           table.                                                        */
/*-------------------------------------------------------------------------*/

typedef int            KEY_TYPE;
typedef unsigned short LOCK_TYPE;
typedef unsigned short COLOR_TYPE;
typedef short          TIME_TYPE;

struct node{                /* record structure for tree node */
    KEY_TYPE      key;      /* search key or offset to next free slot */
                            /*   ( for 1st node only ) */
    COLOR_TYPE    color;    /* color also indicates the usage of a node */
    LOCK_TYPE     Wlock;    /* write-exclusion, read, and exclusive locks */
    LOCK_TYPE     RXlock;
    struct node   *left;    /* left and right pointers */
    struct node   *right;
    TIME_TYPE     time;     /* node creation time */
    TIME_TYPE     aux_time; /* extra pointer and time stamp */
    struct node   *aux_ptr;
} ;

typedef struct node    NODE_TYPE;
typedef NODE_TYPE      *NODE_PTR;


typedef struct {            /* record structure for tree header */
    LOCK_TYPE     status;   /* the state USED or FREE */
    LOCK_TYPE     Whlock;   /* write-exclusion lock */
    LOCK_TYPE     RXhlock;  /* read and exclusive locks */
    NODE_PTR      root;     /* pointer to a root */
} HDR_TYPE, *HDR_PTR;


typedef struct {               /* record structure for system table */
    LOCK_TYPE     timelock;    /* lock to protect current_time */
    TIME_TYPE     current_time; /* time unit relative to base_time */
    long          base_time;   /* time at the first time() call */
    LOCK_TYPE     userlock;    /* lock to protect user field */
    LOCK_TYPE     user;        /* number of users at current time */
} SYSINFO_TYPE, *SYSINFO_PTR;
```

```
/*-------------------------------------------------------------------------*/
/* rbc()                                                                   */
/*      a driver for processing transactions from two concurrently executed */
/*      tasks.  This driver first creates a shared global memory           */
/*      containing "MAX_NODE" number of nodes, a shared header table       */
/*      containing "MAX_HEADER" number of entries, and a system table      */
/*      containing base time, current time, and a few other things.  Then  */
/*      it builds an initial red-black tree with time stamp = 0 from an    */
/*      input file called "infile".  Next, it invokes fork() system call   */
/*      to create a child process, and from this point onward each task goes*/
/*      on to execute its transactions independently.                      */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"

HDR_PTR           header;       /* pointer to a header */
TIME_TYPE         ttime;        /* transaction time */
unsigned short    task, flag;   /* task id. and a flag */

main()
{
   FILE       *fp0,              /* input file containing numerical keys for */
                                 /*   the initial tree */
              *fp1, *fp2,        /* transaction files */
              *ofp0, *ofp1, *ofp2,/* output files */
              *fclose(), *fopen();/* standard file functions */
   TIME_TYPE  gettime();         /* fun. to find the current time stamp */

   /* create a shared memory, header, and system table */
   if ( ( creatshm() ) == FAIL  ||  ( creathdr() ) == FAIL  ||
      ( creatsysinfo() ) == FAIL )
   {
      printf("program stop !\n");
      exit( 1 );
   }

   if ( ( fp0 = fopen( "infile", "r" ) ) != NULL  &&
      ( ofp0 = fopen( "outfile", "w" ) ) != NULL )
   {
      task = 1;
      buildtree( fp0, ofp0 );     /* build an initial red-black tree */
      fclose( fp0 );
      fclose( ofp0 );
   }
   else
   {
      printf("file opening error -- program stop !\n");
      exit(1);
   }
```

```
   settime( 0 );            /* reset current time stamp to 0 */
   while ( gettime() < 1 )  /* do not process transactions current time = 1 */
      sleep( 1 );

   /* use fork() system call to spawn a child process */
   if ( fork() == 0 )            /* fork() returns an zero pid to the child, */
   {                             /*   so the child continues from here */
      if ( ( fp2 = fopen( "trans2", "r" ) ) != NULL   &&
         ( ofp2 = fopen( "out2", "w" ) ) != NULL )
      {
         task = 2;
         transact( fp2, ofp2 );   /* process transactions by task #2 */
         fclose( fp2 );
         fclose( ofp2 );
      }
      else
      {
         printf("Can't open file 'trans2' -- program stop !\n");
         exit( 1 );
      }
   }
   else                  /* fork() returns a nonzero pid to calling process, */
   {                     /*   so the parent (main()) continues from here */
      if ( ( fp1 = fopen( "trans1", "r" ) ) != NULL &&
         ( ofp1 = fopen( "out1", "w" ) ) != NULL )
      {
         task = 1;
         transact( fp1, ofp1 );   /* process transactions by task #1 */
         fclose( fp1 );
         fclose( ofp1 );
      }
      else
      {
         printf("Can't open file 'trans1' -- program stop !\n");
         exit( 1 );
      }
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* transact()                                                              */
/*      function to process transactions (search, insertion, and deletion).*/
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstext.h"

transact( fp, ofp )
FILE     *fp, *ofp;          /* input and output files */
{
   HDR_PTR   gethdr();       /* fun. to find a header */
   TIME_TYPE gettime();      /* fun. to get the current time */
   int       tcode, tkey, stime,/* transaction code, key, and time */
             ctime;          /* current time stamp */

   /* process transactions ... */
   while ( ( fscanf( fp, "%d%d%d", &tcode, &tkey, &stime ) ) != EOF )
   {
       fprintf( ofp, "msg: tra(c=%d,k=%d,t=%d)\n",tcode,tkey,stime );
       switch ( tcode )
       {
           case 0   :          /* search */
                while ( gettime() < stime )
                   sleep( 1 );
                header = gethdr( ttime = stime, task );
                conc_search( tkey, ttime, ofp );
                subuser( task );       /* adduser() is performed in gethdr() */
                break;
           case 1   :          /* insertion */
                header = gethdr( ttime=gettime(), task );
                conc_insert( tkey, ofp );
                subuser( task );
                break;
           case 2   :          /* deletion */
                header = gethdr( ttime=gettime(), task );
                conc_delete( tkey, ofp );
                subuser( task );
                break;
           case 7   :
                while ( ( ctime=gettime() )  <  stime )
                   sleep( 1 );
                fprintf( ofp,"msg: current time = %d\n\n",ctime );
                break;
           default  :
                fprintf(ofp,"error  : invalid transaction code (%d)\n\n",tcode);
       }
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* buildtree()                                                             */
/*      function to build an initial persistent red-black tree with time   */
/*      stamp = 0.                                                         */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

buildtree( fp, ofp )
FILE      *fp, *ofp;                    /* input and output files */
{
   HDR_PTR   gethdr();                  /* fun. to find a header */
   int       newkey;                    /* input key */

   header = gethdr( ttime=0, task );    /* create a header with time stamp = 0 */

   while ( ( fscanf( fp, "%d", &newkey ) ) != EOF )
   {
      conc_insert( newkey, ofp );       /* build init. persistent RB tree */
   }
   subuser( task );
}
```

```
/*-------------------------------------------------------------------------*/
/* conc_delete()                                                           */
/*      function to delete an item from  a  shared persistent red-black tree  */
/*      in a concurrent environment.                                       */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

conc_delete( dkey, ofp )
int         dkey;                       /* item to be deleted */
FILE        *ofp;                       /* output file */
{
   NODE_PTR  stack[STACK_SIZE],         /* stack to store an access path */
             sibptr, dptr, parent, repptr,/* node pointers */
             d_search(),                /* fun. to find dkey */
             sibling();                 /* fun. to find a sibling */
   int       top = -1, bottom=0;        /* indexes to the stack */

   initstack( stack, STACK_SIZE );      /* initialize stack */

   /* search for the item to be deleted */
   dptr = d_search( dkey, stack, &top, &bottom, ofp );
   if ( dptr == NULL )                  /* item not found */
   {
      unWlockall( stack, top, bottom, ofp );  /* release all Wlocks */
      fprintf(ofp,"msg: item not found at %d -- deletion failed\n",dkey,ttime);
      return;
   }

   sibptr = sibling( repptr = stack[top], dptr, ttime );
   if ( top > 0 )                    /* tree with more than 1 internal node */
   {
      parent = stack[top-1];
      if ( AUX_FREE( parent ) )
         update_parent( parent, sibptr, ttime );
      else                           /* copy one or more nodes */
      {
         top--;
         morecopy( sibptr, stack, &top, &bottom );
         top++;
      }
      if ( IS_RED( sibptr ) )
         sibptr->color = BLACK;
      else if ( IS_BLK( repptr ) )
      {                               /* rebalance persistent RB tree */
         unWlocknode( repptr, ofp );
         stack[top] = sibptr;
         d_transform( stack, &top, &bottom, ofp );
```

```
      }
   }
   else                                /* tree with only 1 internal node */
      header->root = sibptr;           /* it becomes an empty tree */
   unWlockall( stack, top, bottom, ofp );
   fprintf(ofp,"msg: item %d deleted at %d\n",dkey,ttime);
}
```

```
/*-------------------------------------------------------------------------*/
/* conc_insert()                                                           */
/*      function to add an item to  a  shared persistent red-black tree in a  */
/*      concurrent environment.                                            */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

conc_insert( insert_key, ofp )
int      insert_key;                /* insertion key */
FILE     *ofp;                      /* output file */
{
   NODE_PTR  stack[STACK_SIZE],      /* stack to store the access path */
             extptr, parent, newnode, /* node pointers */
             i_search(),             /* fun. to locate the place of Ins. */
             attachnode();           /* fun. to create two new nodes */
   int       top = -1, bottom=0;     /* indexes to the stack */

   initstack( stack, STACK_SIZE );   /* initialize stack */

   /* search for a proper location for the new node */
   if ( ( extptr = i_search( insert_key,stack,&top,&bottom,ofp ) ) == NULL )
   {                                 /* new key is already in the tree */
      unWlockall( stack, top, bottom, ofp );
      fprintf(ofp,"msg: item %d already in the tree = %d\n",insert_key,ttime);
      return;
   }
                                     /* insert new item */
   /* attach an internal and an external nodes to the tree */
   newnode = attachnode( insert_key, extptr, ofp );
   if ( top > -1 )
   {                                 /* not an empty tree */
      if ( AUX_FREE( stack[top] ) )  /* parent of new node has a free slot */
         update_parent( stack[top], newnode );
      else                           /* copy 1 or more nodes */
         morecopy( newnode, stack, &top, &bottom );
      if ( IS_RED( stack[top] ) )
      {                              /* rebalance the persistent RB tree */
         Wlocknode ( newnode, ofp );
         push( stack, &top, &bottom, newnode, ofp );
         i_transform( stack, &top, &bottom, ofp );
      }
   }
   else                              /* empty tree */
      header->root = newnode;
   unWlockall( stack, top, bottom, ofp );
   fprintf(ofp,"msg: item %d inserted at %d\n",insert_key,ttime);
}
```

```
/*-------------------------------------------------------------------------*/
/* conc_search()                                                           */
/*      function to find an item in a shared persistent red-black tree in a*/
/*      concurrent environment.                                            */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

conc_search( skey, stime, ofp )
int         skey, stime;        /* search key and time */
FILE        *ofp;               /* output file */
{
   NODE_PTR  cur, son,          /* current and next pointers */
             nextson();         /* fun. to find the desired next ptr */

   Rlockhdr( header, ofp );     /* put a read lock on the header */
   cur = header->root;
   if ( IS_EXTERNAL( cur ) )    /* empty tree */
   {
      unRlockhdr( header, ofp );
      fprintf( ofp, "msg : search(%d) : not found at %d\n\n",skey,stime);
   }
   else                         /* search through the tree */
   {
      Rlocknode( cur, ofp );
      unRlockhdr( header, ofp );
      son = nextson( cur, skey, stime );
      while ( IS_INTERNAL( son ) )
      {
         Rlocknode( son, ofp );
         unRlocknode( cur, ofp );
         cur = son;
         son = nextson( cur, skey, stime );
      }
      if ( son->key == skey )
         fprintf( ofp, "msg : search(%d) : found at %d\n\n",skey,stime);
      else
         fprintf( ofp, "msg : search(%d) : not found at %d\n\n",skey,stime);
      unRlocknode( cur, ofp );
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* copyfun.c - module containing node copying functions : attachnode()     */
/*                                                       copynode()        */
/*                                                       morecopy()        */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

/*-------------------------------------------------------------------------*/
/* attachnode()                                                            */
/*      function to create an external node containing the new item and an */
/*      internal node containing the minimum key of its two children.      */
/*-------------------------------------------------------------------------*/

NODE_PTR  attachnode( i_key, e_ptr, ofp )
int       i_key;              /* new item */
NODE_PTR  e_ptr;              /* pointer to a child of the new int. node */
FILE      *ofp;               /* output file */
{
   NODE_PTR  getnode(),       /* fun. to allocate a new node */
             in, ex;          /* pointers to new int. and ext. nodes */

   if ( ( ex = getnode( ofp ) ) != FAIL  &&  /* create a new ext. node */
      ( in = getnode( ofp ) ) != FAIL )      /* create a new int. node */
   {                                         /* fill in the components */
      ex->color = BLACK;
      ex->key = i_key;
      ex->time = in->time = ttime;
      in->color = RED;
      if ( i_key < e_ptr->key )
      {
         in->key = i_key;
         in->left = ex;
         in->right = e_ptr;
      }
      else
      {
         in->key = e_ptr->key;
         in->left = e_ptr;
         in->right = ex;
      }
      return( in );
   }
   return( FAIL );
}
```

```
/*-------------------------------------------------------------------------*/
/* copynode()                                                              */
/*      function to create a new node from some internal node.  The new node */
/*      contains the latest left and right pointers of the original node, and */
/*      the write-exclusion lock on the original node is released.         */
/*-------------------------------------------------------------------------*/

NODE_PTR copynode( origin, son)
NODE_PTR  origin, son;            /* original node and its child which */
                                  /*    lies on access path */
{
   NODE_PTR  n, getnode(),        /* fun. to allocate a new node */
             leftson(), rightson();/* fun. to find left and right children */

   if ( ( n = getnode() ) != FAIL )       /* get a new node */
   {
      n->key = origin->key;
      n->color = origin->color;
      n->Wlock = origin->Wlock;
      n->time = ttime;
      n->left = leftson( origin, ttime );
      n->right = rightson( origin, ttime );
      if ( son != NULL )
      {
         if ( son->key <= origin->key )
            n->left = son;
         else
            n->right = son;
      }
      origin->Wlock = UNLOCK;
      return( n );
   }
   return( FAIL );
}
```

```
/*-------------------------------------------------------------------------*/
/* morecopy()                                                              */
/*      function to copy node(s) starting from the top of stack if necessary. */
/*-------------------------------------------------------------------------*/

morecopy( c, stack, top, bottom )
NODE_PTR  c, stack[];     /* node pointer and stack containing an access path */
int       *top, *bottom;  /* indexes to the stack */
{
   int       i, done = NO;

   for ( i = *top; i >= *bottom && !done; i-- )
   {
      if ( AUX_USED( stack[i] ) &&
         ( stack[i]->aux_time < ttime || stack[i]->aux_ptr->key != c->key ) )
         stack[i] = c = copynode( stack[i], c);
      else
      {
         update_parent( stack[i], c );
         done = YES;
      }
   }
   if ( !done )
      header->root = c;
}
```

```
/*-------------------------------------------------------------------------*/
/* d_search()                                                              */
/*      function to locate the item to be deleted.  Nodes encountered on the  */
/*      search path are placed with write-exclusion locks.  A NULL value is   */
/*      returned if the item is not found.                                 */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

NODE_PTR d_search( dkey, stack, top, bottom, ofp )
NODE_PTR stack[];               /* stack to store the access path */
int      dkey, *top, *bottom;   /* deletion key and indexes to the stack */
FILE     *ofp;                  /* output file */
{
   NODE_PTR  ls, rs, cur, son,      /* node pointers */
             leftson(), rightson(); /* fun. to find the latest children */
   int       i, j;

   Wlockhdr( header, ofp );         /* place a write-exclusion lock on header */
   flag = 1;
   if ( IS_EXTERNAL( header->root ) ) /* empty tree */
      return( NULL );

   Wlocknode( cur = header->root, ofp, *top );  /* put a Wlock on the root */
   push( stack, top, bottom, cur, ofp );
   ls = leftson( cur, ttime );
   rs = rightson( cur, ttime );
   if ( IS_BLK( cur ) && HAS_2_BLK_SON( cur ) )
      cur->color = RED;
   son = ( ( dkey <= cur->key )  ? ( ls )  : ( rs ) );

   while ( IS_INTERNAL( son ) )             /* search through the tree */
   {
      Wlocknode( son, ofp, *top );
      push( stack, top, bottom, son, ofp );  /* save the access path */
      ls = leftson( son, ttime );
      rs = rightson( son, ttime );
      if ( IS_RED( son ) || HAS_1_RED_SON( son ) )
      {
         for ( j=0,i = *bottom; i < *top; i++ )
         {     /* a node has a free slot, or its aux_ptr is pointed to a node */
               /*    on the search path at the current time */
            if ( AUX_FREE( stack[i] )  ||  ( stack[i]->aux_time == ttime
                 && stack[i]->aux_ptr == stack[i+1] ) )
            {
               if ( flag )
               {
                  unWlockhdr( header, ofp );
```

```
                flag = 0;
              }
              j = i;
            }
          }
          while ( *bottom < j ) /* release Wlocks from the old starred node to */
            unWlocknode( stack[*bottom++] );/* the parent of new starred node */
        }
        cur = son;
        son = ( ( dkey <= cur->key )  ? ( ls )  : ( rs ) );
      }
      if ( son->key == dkey )
        return( son );
      else
        return( NULL );
    }
```

```
/*-------------------------------------------------------------------------*/
/* d_transform()                                                           */
/*      function to rebalance a persistent red-black tree after a deletion.*/
/*      The balance properties of the red-black tree are maintained by color*/
/*      changes or a combination of color and pointer changes.             */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

d_transform( stack, top, bottom, ofp )
NODE_PTR  stack[];          /* stack containing the access path */
int       *top, *bottom;    /* indexes to the stack */
FILE      *ofp;             /* output file */
{
   NODE_PTR    cptr, sptr, gptr, nptr, shortptr, saveptr, p,
               pop(),               /* fun. to remove an item from the stack */
               leftson(), rightson();/* fun. to find left and right children  */
   COLOR_TYPE  savecolor;
   int         done = NO;

   gptr = nptr = shortptr = saveptr = p = NULL;
   shortptr = pop( stack, top, bottom );
   cptr = pop( stack, top, bottom );

   while ( !done  &&  IS_BLK( shortptr )  &&  cptr != NULL  )
   {
      savecolor = cptr->color;         /* save color for ambiguous node */
      if ( shortptr->key <= cptr->key )
      {
         sptr = rightson( cptr, ttime );
         Wlocknode( sptr , ofp );
         if ( IS_RED( sptr ) )         /* case b -- nonterminating case */
         {                             /*           single left rotation */
            p = NULL;
            pre_rotation( &cptr,&sptr,&p,stack,top,bottom,ofp );
            single_L( cptr, sptr, stack, top, bottom, ofp );
            push( stack, top, bottom, sptr, ofp );
         }
         else
         {
            gptr = rightson( sptr, ttime );
            if ( IS_RED( gptr ) )        /* case d -- single left rotation */
            {
               p = NULL;
               pre_rotation( &cptr,&sptr,&p,stack,top,bottom,ofp );
               single_L( cptr, sptr, stack, top, bottom, ofp );
               sptr->color = savecolor;
               cptr->color = gptr->color = BLACK;
```

```
            done = unWlockx( shortptr, cptr, sptr, NULL, ofp );
         }
         else
         {
            gptr = leftson( sptr, ttime );
            if ( ( Wlocknode( gptr, ofp ) ) == NULL  ||  IS_BLK( gptr ) )
            {
               if ( IS_RED( cptr ) )  /* case c -- change color only */
               {
                  cptr->color = BLACK;
                  done = unWlockx( shortptr, cptr, sptr, gptr, ofp );
               }
               else                   /* case a -- nonterminating case */
               {
                  unWlockx( shortptr, NULL, sptr, gptr, ofp );
                  shortptr = cptr;
                  cptr = pop( stack, top, bottom );
               }
               sptr->color = RED;
            }
            else                      /* case e -- double left rotation */
            {
               pre_rotation( &cptr,&sptr,&gptr,stack,top,bottom, ofp );
               double_L( cptr,sptr,gptr,stack,top,bottom, ofp );
               gptr->color = savecolor;
               cptr->color = sptr->color = BLACK;
               done = unWlockx( shortptr, cptr, sptr, gptr, ofp );
            }
         }
      }
   }
   else  /* mirror images of the above cases */
   {
      sptr = leftson( cptr, ttime );
      Wlocknode( sptr, ofp );
      if ( IS_RED( sptr ) )          /* case b -- nonterminating case */
      {                              /*           single right rotaion */
         p = NULL;
         pre_rotation( &cptr,&sptr,&p,stack,top,bottom, ofp );
         single_R( cptr, sptr, stack, top, bottom, ofp );
         push( stack, top, bottom, sptr, ofp );
      }
      else
      {
         gptr = leftson( sptr, ttime );
         if ( IS_RED( gptr ) )       /* case d -- single right rotation */
         {
            p = NULL;
            pre_rotation( &cptr,&sptr,&p,stack,top,bottom, ofp );
            single_R( cptr, sptr, stack, top, bottom, ofp );
            sptr->color = savecolor;
            cptr->color = gptr->color = BLACK;
```

```
                    done = unWlockx( shortptr, cptr, sptr, NULL, ofp );
                }
                else
                {
                    gptr = rightson( sptr, ttime );
                    if ( Wlocknode( gptr, ofp ) == NULL  ||  IS_BLK( gptr ) )
                    {
                        if ( IS_RED( cptr ) )  /* case c -- change color only */
                        {
                            cptr->color = BLACK;
                            done = unWlockx( shortptr, cptr, sptr, gptr, ofp );
                        }
                        else                   /* case a -- nonterminating case */
                        {
                            unWlockx( shortptr, NULL, sptr, gptr, ofp );
                            shortptr = cptr;
                            cptr = pop( stack, top, bottom );
                        }
                        sptr->color = RED;
                    }
                    else                       /* case e -- double right rotation */
                    {
                        pre_rotation( &cptr,&sptr,&gptr,stack,top,bottom, ofp );
                        double_R( cptr,sptr,gptr,stack,top,bottom,ofp );
                        gptr->color = savecolor;
                        cptr->color = sptr->color = BLACK;
                        done = unWlockx( shortptr, cptr, sptr, gptr, ofp );
                    }
                }
            }
        }
    }
}
```

```
/*-------------------------------------------------------------------------*/
/* err_handler()                                                           */
/*      function to handle a few error conditions.                         */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"

err_handler( err_msg, stack, top, bottom, ofp )
int      err_msg, top, bottom;  /* error message and indexes to the stack */
NODE_PTR stack[];               /* stack containing the access path */
FILE     *ofp;                  /* output file */
{
   switch ( err_msg )
   {
      case OUT_HDR  :
           fprintf( ofp, "Error  : Out of Header Nodes !\n" );
           break;
      case OUT_NODE :
           fprintf( ofp, "Error  : Out of Shared Memory !\n" );
           break;
       case STK_OVFL :
           fprintf( ofp, "Error  : Stack Overflow !\n" );
           break;
       default       :
           fprintf( ofp, "Error  : Nothing Wrong !\n" );
   }
   unWlockall( stack, top, bottom );  /* release all locks on the access path */
   fprintf( ofp, "Program Stopped Abnormally !\n");
   exit( 1 );
}
```

```
/*-------------------------------------------------------------------------*/
/* hdrfun.c - module contains header processing functions : creathdr()     */
/*                                                          gethdr()       */
/*-------------------------------------------------------------------------*/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"

int       sid2;   /* shared memory id */
HDR_PTR  hdrtab;  /* pointer to header table */

/*-------------------------------------------------------------------------*/
/* creathdr()                                                              */
/*      create a shared memory with MAX_HEADER number of entries by shmget()  */
/*      Unix system call.                                                  */
/*-------------------------------------------------------------------------*/

creathdr()
{
   extern  char  *shmat(); /* attach the shared header to its virtual memory */
   HDR_PTR       haddr;    /* header pointer */
   int           i, j;

   /* create a shared header table */
   sid2 = shmget( SHMKEY2, MAX_HEADER * sizeof(HDR_TYPE), 0777|IPC_CREAT);
   if ( sid2 != ERROR )
   {
      haddr = hdrtab = ( HDR_PTR ) shmat( sid2, 0, 0 );
      for ( i=0; i < MAX_HEADER; i++ )        /* initialize table */
      {
         haddr->status = FREE;
         haddr->RXhlock = haddr->Whlock = UNLOCK;
         haddr->root = NULL;
         haddr++;
      }
      printf("\nA Header Table of %d entry has been created :\n",MAX_HEADER );
      printf("Sid2 = %d\thdrtab = %x\tlastaddr = %x\n",sid2,hdrtab,--haddr );
      return( OK );
   }
   printf("\nError  : Cannot create shared header table !\n");
   return( ERROR );
}
```

```
/*-------------------------------------------------------------------------*/
/* gethdr()                                                                */
/*      function to find a header associated with a given time stamp.  If the */
/*      given time stamp is greater than the current time stamp, one or more  */
/*      new headers are created.                                           */
/*-------------------------------------------------------------------------*/

HDR_PTR gethdr( ttime, task, ofp )
TIME_TYPE       ttime;      /* transaction time stamp */
unsigned short  task;       /* current task id. */
FILE            *ofp;       /* output file */
{
   extern SYSINFO_PTR  sysinfo;    /* pointer to system table */
   NODE_PTR            p,          /* node pointer */
                       getnode();  /* fun. to allocate a new node */
   int                 i, t;

   if ( ttime > ( t = sysinfo->current_time ) )
   {
      if ( ttime < MAX_HEADER )
      {
         locktime();   /* enter critical sect. -- allow 1 task to modify time */
         if ( tas( &hdrtab[ttime].status ) == 0 )
         {
            for ( i=t+1; i < ttime; i++ )    /* create headers from the last */
            {                                /*   current_time to ttime */
               if ( tas( &hdrtab[i].status ) == 0 )
                  hdrtab[i].root = hdrtab[i-1].root;
            }
            if ( ttime == 0 )
            {                           /* create a dummy node for init. tree */
               p = hdrtab[ttime].root = getnode( ofp );
               p->time = 0;
               p->key = MAX_KEY;
               p->color = BLACK;
            }
            else
            {
               hdrtab[ttime].root = hdrtab[t].root;
               while ( sysinfo->user > 0 )/* wait for tasks in current time */
                  sleep( 1 );             /*   stamp to complete their jobs */
            }
            sysinfo->current_time = ttime; /* set the new current time stamp */
         }
         unlocktime();   /* exit from the critical section */
      }
      else
         err_handler( OUT_HDR );     /* out of header nodes -- PROGRAM STOP ! */
   }
   adduser( task );                  /* increment user count */
   return( &hdrtab[ttime] );
}
```

```
/*-------------------------------------------------------------------------*/
/* i_search()                                                              */
/*      function to search for a proper location for the new item.  Write- */
/*      exclusion locks are placed on nodes encountered on the access path.*/
/*      A NULL value is returned if the new item is already in the tree.   */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

NODE_PTR i_search( newkey, stack, top, bottom, ofp )
NODE_PTR stack[];               /* stack to store the access path */
int      newkey, *top, *bottom; /* new item and indexes to the stack */
FILE     *ofp;                  /* output file */
{
   NODE_PTR  ls, rs, cur, son,
             leftson(), rightson(); /* fun. to find left and right children */
   int       i, j;

   Wlockhdr( header, ofp );                    /* Wlock header */
   flag = 1;
   if ( IS_EXTERNAL( header->root ) )          /* empty tree */
      return( header->root );

   Wlocknode( cur = header->root, ofp, *top ); /* Wlock root */
   push( stack, top, bottom, cur, ofp );
   ls = leftson( cur, ttime );
   rs = rightson( cur, ttime );
   if ( IS_RED( cur ) )
      cur->color = BLACK;
   else if ( IS_RED( ls )  &&  IS_RED( rs ) )
      ls->color = rs->color = BLACK;
   son = ( ( newkey <= cur->key )  ? ( ls )  : ( rs ) );

   while ( IS_INTERNAL( son ) )
   {
      Wlocknode( son, ofp, *top );
      push( stack, top, bottom, son, ofp );
      ls = leftson( son, ttime );
      rs = rightson( son, ttime );
      if ( IS_BLK( son ) )
      {
         if ( IS_BLK( ls )  ||  IS_BLK( rs )  ||  IS_BLK( cur ) )
         {
            for ( j=0,i = *bottom; i < *top; i++ )
            {
               if ( AUX_FREE( stack[i] )  ||  ( stack[i]->aux_time == ttime
                    && stack[i]->aux_ptr == stack[i+1] ) )
               {
```

```
                if ( flag )
                {
                   unWlockhdr( header, ofp );
                   flag = 0;
                }
                j = i;
             }
          }
          while ( *bottom < j )   /* remove Wlocks from the old starred node */
             unWlocknode( stack[*bottom++], ofp ); /*   to the parent of new */
       }                                          /*   starred node */
    }
    cur = son;
    son = ( ( newkey <= cur->key )  ? ( ls )  : ( rs ) );
  }
  if ( son->key != newkey )
    return( son );
  else
    return( NULL );
}
```

```
/*-------------------------------------------------------------------------*/
/* i_transform()                                                           */
/*      function to rebalance a persistent red-black tree after an insertion. */
/*      The balance properties of the red-balck tree are maintained  by    */
/*      color changes or a combination of color and pointer changes.  When a  */
/*      rotation is necessary, some nodes may be copied, and a few write-  */
/*      exclusion locks are converted to exclusive locks.                  */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

i_transform( stack, top, bottom, ofp )
NODE_PTR   stack[];       /* stack containing access path */
int        *top, *bottom; /* indexes to the stack */
FILE       *ofp;          /* output file */
{
   NODE_PTR  gptr, sptr, cptr,ls1,ls2, rs1, rs2,          /* node pointers */
             pop(),               /* fun. to remove an item from a stack  */
             leftson(), rightson(); /* fun. to find left and right children */
   int       done = NO;

   gptr = pop( stack, top, bottom );
   sptr = pop( stack, top, bottom );
   cptr = pop( stack, top, bottom );

   while ( cptr != NULL  &&  !done )
   {
      if ( IS_BLK( cptr )  &&  IS_RED( sptr )  &&  IS_RED( gptr ) )
      {
         if ( ( ls1 = leftson( cptr, ttime ) )  ==  sptr )
         {
            rs1 = rightson( cptr, ttime );
            if ( IS_BLK( rs1 ) )
            {
               if ( ( rs2 = rightson( sptr, ttime ) )  ==  gptr )
               {                          /* case d -- double right rotation */
                  pre_rotation( &cptr,&sptr,&gptr,stack,top,bottom,ofp);
                  double_R( cptr,sptr,gptr,stack,top,bottom,ofp);
               }
               else
               {                          /* case c -- single right rotation */
                  rs2 = NULL;
                  pre_rotation( &cptr,&sptr,&rs2,stack,top,bottom,ofp );
                  single_R( cptr,sptr,stack,top,bottom,ofp);
               }
               done = YES;
            }
            else                          /* case a -- nonterminating case   */
```

```
            {
               sptr->color = rs1->color = BLACK;
               cptr->color = RED;
            }
         }
         else                              /* mirror images of the above cases */
         {
            if ( IS_BLK( ls1 ) )
            {
               if ( ( ls2 = leftson( sptr, ttime ) )  ==  gptr )
               {                             /* case d -- double left rotation  */
                 pre_rotation(&cptr,&sptr,&gptr,stack,top,bottom,ofp);
                 double_L( cptr,sptr,gptr,stack,top,bottom,ofp);
               }
               else                          /* case c -- single left rotation  */
               {
                  ls2 = NULL;
                  pre_rotation(&cptr,&sptr,&rs2,stack,top,bottom,ofp);
                  single_L( cptr,sptr,stack,top,bottom,ofp);
               }
               done = YES;
            }
            else                              /* case a -- nonterminating case  */
            {
               sptr->color = ls1->color = BLACK;
               cptr->color = RED;
            }
         }
      }
      if ( !done )
      {
         unWlocknode( gptr, ofp ); /* release Wlock on the lowest node of the */
         gptr = sptr;              /*    three and move up the access path    */
         sptr = cptr;
         cptr = pop( stack, top, bottom );
      }
   }
   if ( !done  &&  IS_RED( gptr )  &&  IS_RED( sptr ) ) /* case b */
      sptr->color = BLACK;
   unWlocknode( gptr, ofp );        /* release Wlocks on nodes that have been */
   unWlocknode( sptr, ofp );        /*    removed from the stack */
   unWlocknode( cptr, ofp );
}
```

```
/*-------------------------------------------------------------------------*/
/* lockfun.c - module containing 'lock' and 'unlock' functions: Wlockhdr()  */
/*                                                             Wlocknode()  */
/*                                                             Rlockhdr()   */
/*                                                             Rlocknode()  */
/*                                                             Xlockhdr()   */
/*                                                             Xlocknode()  */
/*                                                             locktime()   */
/*                                                             lockuser()   */
/*                                                             unWlockx()   */
/*                                                             unWlockall() */
/*                                                             unWlockhdr() */
/*                                                             unWlocknode()*/
/*                                                             unRlockhdr() */
/*                                                             unRlocknode()*/
/*                                                             unXlockhdr() */
/*                                                             unXlocknode()*/
/*                                                             unlocktime() */
/*                                                             unlockuser() */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"

extern   SYSINFO_PTR  sysinfo;  /* pointer to system table */
unsigned short   mask[]   = { 0x0000, 0x0001, 0x0002, 0x0004, 0x0008 };
unsigned short   unmask[] = { 0x7fff, 0xfffe, 0xfffd, 0xfffb, 0xfff7 };

/*-------------------------------------------------------------------------*/
/* Wlockhdr()                                                              */
/*      function to place a write-exclusion lock on a header by setting the*/
/*      most significant bit of Whlock field to 1.  If the MSB is already set*/
/*      by another task, the caller  has to try again later.  The setting of*/
/*      the MSB is done by the indivisible test-and-set instruction.       */
/*-------------------------------------------------------------------------*/

Wlockhdr( header )
HDR_PTR   header;   /* header pointer */
{
   while ( tas( &header->Whlock ) )  /* tas() returns 0 if Wlock is granted, */
      sleep( 1 );                    /*   1 if Wlock is held by another task */
}
```

```
/*-------------------------------------------------------------------------*/
/* Wlocknode()                                                             */
/*      function to place a write-exclusion lock on a node by setting the  */
/*      most significant bit of Wlock field to 1.  If the MSB is already set */
/*      by another task, the caller  has to try again later.  The setting of */
/*      the MSB is done by indivisible test-and-set instruction.           */
/*-------------------------------------------------------------------------*/

Wlocknode( nodeptr )
NODE_PTR  nodeptr;    /* node pointer */
{
   if ( nodeptr != NULL )
   {
      while ( tas( &nodeptr->Wlock ) )/* tas() returns 0 if Wlock is granted, */
         sleep( 1 );                  /*   1 if Wlock is held by another task */
      return( OK );
   }
   return( NULL );
}




/*-------------------------------------------------------------------------*/
/* Rlockhdr()                                                              */
/*      function to place a read lock on a header by marking the taskth bit */
/*      of RXhlock.                                                        */
/*-------------------------------------------------------------------------*/

Rlockhdr( header )
HDR_PTR   header;         /* header pointer */
{
   while ( tas( &header->RXhlock ) )       /* try to enter a critical section */
      sleep( 1 );                          /* busy -- sleep for 1 sec.        */
   header->RXhlock |= mask[task];          /* mark taskth bit of RXhlock      */
   header->RXhlock &= unmask[0];           /* exit from critical section      */
}
```

```
/*-------------------------------------------------------------------------*/
/* Rlocknode()                                                             */
/*      function to put a read lock on a node by marking the taskth bit of */
/*      RXlock field.                                                      */
/*-------------------------------------------------------------------------*/

Rlocknode( nodeptr )
NODE_PTR   nodeptr;       /* node pointer */
{
   while ( tas( &nodeptr->RXlock ) )       /* try to enter a critical section */
      sleep( 1 );                          /* busy -- sleep for 1 sec.        */
   nodeptr->RXlock |= mask[task];          /* mark taskth bit of RXlock       */
   nodeptr->RXlock &= unmask[0];           /* exit from critical section      */
}
```

```
/*-------------------------------------------------------------------------*/
/* Xlockhdr()                                                              */
/*      function to place an exclusive lock on a header by setting the MSB */
/*      of RXhlock to 1.                                                   */
/*-------------------------------------------------------------------------*/

Xlockhdr( header, ofp )
HDR_PTR   header;
{
   int     done = NO;

   while ( !done )
   {
      while ( tas( &header->RXhlock ) )    /* try to enter a critical section */
         sleep(1);                         /* busy -- wait for 1 sec.         */
      if ( header->RXhlock == XLOCK )
         done = YES;                       /* succeed !           */
      else
         header->RXhlock &= unmask[0];     /* fail -- try again ! */
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* Xlocknode()                                                             */
/*      function to place an exclusive lock on a node by setting the most  */
/*      significant bit of RXlock to 1.                                    */
/*-------------------------------------------------------------------------*/

Xlocknode( nodeptr )
NODE_PTR   nodeptr;       /* node pointer */
{
   int     done = NO;

   while ( !done )
   {
      while ( tas( &nodeptr->RXlock ) )    /* try to entry a critical section */
         sleep(1);                         /* busy -- wait for 1 sec.         */
      if ( nodeptr->RXlock == XLOCK )
         done = YES;                       /* succeed !           */
      else
         nodeptr->RXlock &= unmask[0];     /* fail -- try again ! */
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* locktime()                                                              */
/*      function to place an exclusive lock on the current_time field of   */
/*      system information block.                                          */
/*-------------------------------------------------------------------------*/

locktime()
{
   while ( tas( &sysinfo->timelock ) )
      sleep( 1 );
}
```

```
/*-------------------------------------------------------------------------*/
/* lockuser()                                                              */
/*      function to place an exclusive lock on the user field of system    */
/*      information block.                                                 */
/*-------------------------------------------------------------------------*/

lockuser()
{
   while ( tas( &sysinfo->userlock ) )
      sleep ( 1 );
}
```

```
/*-------------------------------------------------------------------------*/
/* unWlockx()                                                              */
/*      function to release a number of write-exclusion locks.             */
/*-------------------------------------------------------------------------*/

unWlockx( shptr, cptr, sptr, gptr )
NODE_PTR  shptr, cptr, sptr, gptr;  /* node pointers */
{
   unWlocknode( shptr );
   unWlocknode( cptr );
   unWlocknode( sptr );
   unWlocknode( gptr );
   return( OK );
}




/*-------------------------------------------------------------------------*/
/* unWlockall()                                                            */
/*      function to release write-exclusion locks on nodes in the stack and,*/
/*      if necessary, on the current header.                               */
/*-------------------------------------------------------------------------*/

unWlockall( stack, top, bottom )
NODE_PTR    stack[];      /* stack containing access path */
int         top, bottom;  /* indexes to the stack         */
{
   int  i;

   if ( flag )
   {
      unWlockhdr( header );
      flag = 0;
   }
   for ( i=bottom;  i <= top;  i++ )
      unWlocknode( stack[i] );
}
```

```
/*-------------------------------------------------------------------------*/
/* unWlockhdr()                                                            */
/*      function to release the write-exclusion lock on a header.          */
/*-------------------------------------------------------------------------*/

unWlockhdr( header )
HDR_PTR     header;       /* header pointer */
{
   if ( header != NULL )
      header->Whlock = UNLOCK;
}




/*-------------------------------------------------------------------------*/
/* unWlocknode()                                                           */
/*      function to release the write-exclusion lock on a node.            */
/*-------------------------------------------------------------------------*/

unWlocknode( nodeptr )
NODE_PTR     nodeptr;   /* node pointer */
{
   if ( nodeptr != NULL )
      nodeptr->Wlock = UNLOCK;
}




/*-------------------------------------------------------------------------*/
/* unRlockhdr()                                                            */
/*      function to release a read lock on a header by unmarking the taskth*/
/*      bit of RXhlock field.                                              */
/*-------------------------------------------------------------------------*/

unRlockhdr( header )
HDR_PTR     header;       /* header pointer */
{
   if ( header != NULL )
   {
      while ( tas( &header->RXhlock ) )    /* try to enter a critical section */
         sleep( 1 );                       /* busy -- sleep for 1 sec.        */
      header->RXhlock &= unmask[task];     /* unmark taskth bit of RXhlock    */
      header->RXhlock &= unmask[0];        /* exit from critical section      */
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* unRlocknode()                                                           */
/*      function to release a read lock on a node by unmarking the taskth bit */
/*      of RXlock field.                                                   */
/*-------------------------------------------------------------------------*/

unRlocknode( nodeptr )
NODE_PTR     nodeptr;     /* node pointer */
{
   if ( nodeptr != NULL )
   {
      while ( tas( &nodeptr->RXlock ) )    /* try to enter a critical section */
         sleep( 1 );                       /* busy -- sleep for 1 sec.        */
      nodeptr->RXlock &= unmask[task];     /* reset taskth bit of RXlock      */
      nodeptr->RXlock &= unmask[0];        /* exit from critical section      */
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* unXlockhdr()                                                            */
/*      function to release an exclusive lock on a header by resetting the    */
/*      MSB of RXhlock.                                                    */
/*-------------------------------------------------------------------------*/

unXlockhdr( header )
HDR_PTR     header;       /* header pointer */
{
   if ( header != NULL )
      header->RXhlock = UNLOCK;
}
```

```
/*-------------------------------------------------------------------------*/
/* unXlocknode()                                                           */
/*      function to release an exclusive lock on a node by resetting the MSB  */
/*      of RXlock.                                                         */
/*-------------------------------------------------------------------------*/

unXlocknode( nodeptr )
NODE_PTR     nodeptr;     /* node pointer */
{
   if ( nodeptr != NULL )
      nodeptr->RXlock = UNLOCK;
}
```

```
/*-------------------------------------------------------------------------*/
/* unlocktime()                                                            */
/*      function to release an exclusive lock on the current_time field of */
/*      system information block.                                          */
/*-------------------------------------------------------------------------*/

unlocktime()
{
   sysinfo->timelock = UNLOCK;
}




/*-------------------------------------------------------------------------*/
/* unlockuser()                                                            */
/*      function to release an exclusive lock on the user field of system  */
/*      information block.                                                 */
/*-------------------------------------------------------------------------*/

unlockuser()
{
   sysinfo->userlock = UNLOCK;
}
```

```
/*-------------------------------------------------------------------------*/
/* nextfun.c - module contains next pointer functions: leftson()           */
/*                                                     rightson()          */
/*                                                     nextson()           */
/*                                                     sibling()           */
/*-------------------------------------------------------------------------*/

#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"

/*-------------------------------------------------------------------------*/
/* leftson()                                                               */
/*      function to find a left child pointer with the biggest time stamp  */
/*      that is less than or equal to search time.                         */
/*-------------------------------------------------------------------------*/

NODE_PTR leftson( nodeptr, stime )
NODE_PTR          nodeptr;      /* node pointer */
TIME_TYPE         stime;        /* search time */
{
   if ( AUX_USED( nodeptr )  &&  nodeptr->aux_time <= stime  &&
      (nodeptr->aux_ptr)->key <= nodeptr->key )
      return( nodeptr->aux_ptr );     /* return aux. pointer */
   return( nodeptr->left );
}



/*-------------------------------------------------------------------------*/
/* rightson()                                                              */
/*      function to find a right child pointer with the biggest time stamp */
/*      that is less than or equal to search time.                         */
/*-------------------------------------------------------------------------*/

NODE_PTR rightson( nodeptr, stime )
NODE_PTR           nodeptr;     /* node pointer */
TIME_TYPE          stime;       /* search time */
{
   if ( AUX_USED( nodeptr )  &&  nodeptr->aux_time <= stime  &&
      (nodeptr->aux_ptr)->key > nodeptr->key )
      return( nodeptr->aux_ptr );     /* return aux. pointer */
   return( nodeptr->right );
}
```

```
/*-------------------------------------------------------------------------*/
/* nextson()                                                               */
/*      function to find a next pointer on the access path.                */
/*-------------------------------------------------------------------------*/

NODE_PTR nextson( pptr, skey, stime )
NODE_PTR          pptr;         /* node pointer */
TIME_TYPE         skey, stime;  /* search key and time */
{
   NODE_PTR  leftson(), rightson();

   if ( skey <= pptr->key )
      return( leftson( pptr, stime ) );
   else
      return( rightson( pptr, stime ) );
}
```

```
/*-------------------------------------------------------------------------*/
/* sibling()                                                               */
/*      function to find the sibling of a node.                            */
/*-------------------------------------------------------------------------*/

NODE_PTR sibling( pptr, ptr, stime )
NODE_PTR          pptr, ptr;    /* ptr is a child pointer of pptr */
TIME_TYPE         stime;        /* search time */
{
   NODE_PTR  leftson(), rightson();

   if ( ptr->key <= pptr->key )
      return( rightson( pptr, stime ) );
   else
      return( leftson( pptr, stime ) );
}
```

```
/*-------------------------------------------------------------------------*/
/* nodefun.c - module contains node processing functions: creatshm()       */
/*                                                        getnode()        */
/*                                                        freenode()       */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "pstdef.h"
#include "pstrec.h"

int         sid1;     /* shared memory id */
NODE_PTR    shmaddr;  /* pointer to shared memory -- don't modify */

/*-------------------------------------------------------------------------*/
/* creatshm()                                                              */
/*      function to create and initialize a shared memory with MAX_NODE number*/
/*      of nodes.  This function is called only once at the beginning of the  */
/*      driver program.                                                    */
/*-------------------------------------------------------------------------*/

creatshm()
{
   extern char *shmat();  /* fun. to attach a shared memo. to its vir. memo. */
   NODE_PTR    saddr;     /* pointer to share memory */
   int         i;

   /* create a shared memory with "MAX_NODE" number of nodes */
   sid1 = shmget( SHMKEY1, ( MAX_NODE+1 ) * sizeof(NODE_TYPE), 0777|IPC_CREAT );
   if ( sid1 != ERROR )
   {
      saddr = shmaddr = ( NODE_PTR  ) shmat( sid1, 0, 0 );
      for( i=0; i < MAX_NODE+1; i++ )  /* init. nodes */
      {
         saddr->key = NULLKEY;
         saddr->color = FREE;
         saddr->RXlock = saddr->Wlock = UNLOCK;
         saddr->time = saddr->aux_time = NULLTIME;
         saddr->left = saddr->right = saddr->aux_ptr = NULL;
         saddr++;
      }
      printf("\nA Shared Memory with %d nodes has been created :\n",MAX_NODE);
      printf("sid1 = %d\tshmaddr = %x\tlastaddr = %x\n",sid1,shmaddr,--saddr);
      shmaddr->key = 1;  /* NOTE : first "key" field contains the offset to */
      return( OK );      /*        the next free node */
   }
   printf("\nCannot create shared memory !\n");
   return( FAIL );
}
```

```
/*-------------------------------------------------------------------------*/
/* getnode()                                                               */
/*      function to allocate a new node from the shared memory.            */
/*-------------------------------------------------------------------------*/

NODE_PTR getnode()
{
   NODE_PTR    naddr;          /* node pointer */
   int         i, offset;

   offset = shmaddr->key;      /* get the offset to the next free node */
   for ( i=0; i < MAX_NODE; i++ )
   {
      naddr = shmaddr + offset++;
      if ( offset > MAX_NODE+1 )
         offset = 1;
      if ( tas( &naddr->color ) == 0 )    /* check if this node is free ? */
      {                                   /* yes */
         Wlocknode( shmaddr );            /* enter critical section */
         shmaddr->key = offset;           /* update offset to next free node */
         unWlocknode( shmaddr );          /* exit from critical section */
         return( naddr );                 /* return new node pointer */
      }
   }
   return( FAIL );
}
```

```
/*-------------------------------------------------------------------------*/
/* freenode()                                                              */
/*      function to reclaim a deleted node and put it on the free list.    */
/*-------------------------------------------------------------------------*/

freenode( nodeptr )
NODE_PTR  nodeptr;               /* pointer to a deleted node */
{
   if ( nodeptr != NULL  &&  nodeptr != shmaddr )
   {
      nodeptr->color = FREE;      /* color also indicates the usage of a node */
      nodeptr->key = NULLKEY;
      nodeptr->Wlock = nodeptr->RXlock = UNLOCK;
      nodeptr->time = nodeptr->aux_time = NULLTIME;
      nodeptr->left = nodeptr->right = nodeptr->aux_ptr = NULL;
   }
}
```

```
/*-------------------------------------------------------------------------*/
/* rotatefun.c - module contains nodes rotating functions: pre_rotation()  */
/*                                                         single_L()      */
/*                                                         single_R()      */
/*                                                         double_L()      */
/*                                                         double_R()      */
/*                                                         update_pptr()   */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"
#include "pstmac.h"
#include "pstext.h"


/*-------------------------------------------------------------------------*/
/* pre_rotation()                                                          */
/*      function to prepare nodes for a single or double rotation by first */
/*      making sure that certain nodes have enough free slots for the new  */
/*      pointers to be added to them.  Next, convert two or three write-   */
/*      exclusive locks to exclusive locks depending on the type of rotation. */
/*-------------------------------------------------------------------------*/

pre_rotation( cptr, sptr, gptr, stack, top, bottom )
NODE_PTR  *cptr, *sptr, *gptr,  /* pointers to node pointer */
          stack[];              /* stack containing an access path */
int       *top, *bottom;        /* indexes to the stack */
{
   NODE_PTR  g, s, c, pptr,     /* node pointers */
             copynode();        /* fun. to create a new node */

   g = s = NULL;
   c = *cptr;
   pptr = stack[*top];

   if ( *gptr != NULL  &&  (*gptr)->time != ttime )    /* double rotation */
   {                       /* need 2 free slots in the promoted node -- gptr */
      *gptr = g = copynode( *gptr, NULL );
      if ( ( (*sptr)->aux_ptr == NULL )  ||           /* has a free slot */
         ( (*sptr)->aux_time == ttime && (*sptr)->aux_ptr->key == (*gptr)->key))
         update_parent( *sptr, *gptr );
      else
         *sptr = s = copynode( *sptr, *gptr );        /* create a new node */
   }
   else if ( *gptr == NULL  &&  (*sptr)->time != ttime ) /* single rotation */
      *sptr = s = copynode( *sptr, *gptr );

   if ( ( (*cptr)->aux_ptr == NULL ) ||
      ( (*cptr)->aux_time == ttime && (*cptr)->aux_ptr->key == (*sptr)->key ) )
   {
      if ( s != NULL )
```

```
      update_parent( *cptr, *sptr );
    if ( ( *top > -1  &&  AUX_USED( pptr ) )  &&
      ( pptr->aux_time < ttime || KEYDIF( pptr->aux_ptr,c->key) ) )
      morecopy( c, stack, top, bottom );          /* copy 1 or more nodes */
  }
  else
  {
    *cptr = c = copynode( *cptr, *sptr );
    morecopy( c, stack, top, bottom );
  }

  if ( *top > -1 )        /* convert Wlocks to Xlocks */
    Xlocknode( stack[*top] );
  else
    Xlockhdr( header );
  Xlocknode( *cptr );     /* single rotation needs Xlocks on pptr and cptr */
  if ( *gptr != NULL )    /* double rotation needs additional Xlock on sptr */
    Xlocknode( *sptr );
}



/*-------------------------------------------------------------------------*/
/* single_L()                                                              */
/*     function to perform a single left rotation.                         */
/*-------------------------------------------------------------------------*/

single_L( cptr, sptr, stack, top, bottom )
NODE_PTR  cptr, sptr,         /* node pointers */
          stack[];            /* stack containing an access path */
int       *top, *bottom;      /* indexes to the stack */
{

   if ( cptr->time < ttime )  /* change 2 pointers */
     update_aux( cptr, sptr->left);
   else
     cptr->right = sptr->left;
   sptr->left = cptr;

   cptr->color = RED;         /* change  2 colors */
   sptr->color = BLACK;

   /* link promoted node to its new parent */
   update_pptr( sptr, stack, top, bottom );
   unXlocknode( cptr );       /* release Xlock */
}
```

```
/*-------------------------------------------------------------------------*/
/* single_R()                                                              */
/*      function to perform a single right rotation.                       */
/*-------------------------------------------------------------------------*/

single_R( cptr, sptr, stack, top, bottom )
NODE_PTR  cptr, sptr,          /* node pointers */
          stack[];             /* stack containing an access path */
int       *top, *bottom;       /* indexes to the stack */
{
    if ( cptr->time < ttime )  /* update 2 pointers */
       update_aux( cptr, sptr->right );
    else
       cptr->left = sptr->right;
    sptr->right = cptr;

    cptr->color = RED;         /* change colors */
    sptr->color = BLACK;

    /* link the promoted node to its new parent */
    update_pptr( sptr, stack, top, bottom );
    unXlocknode( cptr );       /* release Xlock */
}
```

```
/*-------------------------------------------------------------------------*/
/* double_L()                                                              */
/*      function to perform a double left rotation.                        */
/*-------------------------------------------------------------------------*/

double_L( cptr, sptr, gptr, stack, top, bottom )
NODE_PTR  cptr, sptr, gptr,    /* node pointers */
          stack[];             /* stack containing an access path */
int       *top, *bottom;       /* indexes to the stack */
{
    if ( cptr->time < ttime )  /* change 4 pointers */
       update_aux( cptr, gptr->left );
    else
       cptr->right = gptr->left;
    if ( sptr->time < ttime )
       update_aux( sptr, gptr->right );
    else
       sptr->left = gptr->right;
    gptr->left = cptr;
    gptr->right = sptr;

    gptr->color = BLACK;        /* change 2 colors */
    cptr->color = RED;

    /* link the promoted node to its new parent */
    update_pptr( gptr, stack, top, bottom );
    unXlocknode( cptr );        /* release Xlocks */
    unXlocknode( sptr );
}
```

```
/*-------------------------------------------------------------------------*/
/* double_R()                                                              */
/*      function to perform a double right rotation.                       */
/*-------------------------------------------------------------------------*/

double_R( cptr, sptr, gptr, stack, top, bottom )
NODE_PTR  cptr, sptr, gptr,     /* node pointers */
          stack[];              /* stack containing an access path */
int       *top, *bottom;        /* indexes to the stack */
{
    if ( cptr->time < ttime )   /* change 4 pointers */
       update_aux( cptr, gptr->right );
    else
       cptr->left = gptr->right;
    if ( sptr->time < ttime )
       update_aux( sptr, gptr->left );
    else
       sptr->right = gptr->left;
    gptr->left = sptr;
    gptr->right = cptr;

    gptr->color = BLACK;        /* change colors */
    cptr->color = RED;

    /* link the promoted node to its new parent */
    update_pptr( gptr, stack, top, bottom );
    unXlocknode( cptr );        /* release Xlocks */
    unXlocknode( sptr );
}
```

```
/*-------------------------------------------------------------------------*/
/* update_pptr()                                                           */
/*      function to link the promoted node to its new parent.              */
/*-------------------------------------------------------------------------*/

update_pptr( ptr, stack, top, bottom )
NODE_PTR  ptr, stack[];  /* promoted node and stack containing an access path */
int       *top, *bottom; /* indexes to the stack */
{
    NODE_PTR pptr;              /* the new parent of the promoted node */

    if ( *top < *bottom )
    {
       header->root = ptr;
       unXlockhdr( header );    /* release Xlock */
    }
    else
    {
       pptr = stack[*top];      /* pptr is on top of the stack */
       if ( pptr->time < ttime )/* pptr is created at previous time */
          update_aux( pptr, ptr );
       else                     /* pptr is created at current time  */
       {
          if ( ptr->key > pptr->key )
             pptr->right = ptr;
          else
             pptr->left = ptr;
       }
       unXlocknode( pptr );     /* release Xlock */
    }
}
```

```
/*-------------------------------------------------------------------------*/
/* stackfun.c - module contains stack processing functions: initstack()    */
/*                                                          push()         */
/*                                                          pop()          */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include "pstdef.h"
#include "pstrec.h"

/*-------------------------------------------------------------------------*/
/* initstack()                                                             */
/*      function to initialize a stack.                                    */
/*-------------------------------------------------------------------------*/

initstack( stack, size )
NODE_PTR   stack[];       /* a stack */
int        size;          /* size of the stack */
{
   int  i;
   for ( i=0; i < size; i++)
       stack[i] = NULL;
}




/*-------------------------------------------------------------------------*/
/* push()                                                                  */
/*      function to push a node pointer onto a given stack.                */
/*-------------------------------------------------------------------------*/

push( stack, top, bottom, nodeptr, ofp )
NODE_PTR  stack[], nodeptr;    /* stack and node pointer */
int       *top, *bottom;       /* pointers to the top and bottom indexes */
FILE      *ofp;                /* output file */
{
    if ( *top > STACK_SIZE-2 ) /* full */
       err_handler( STK_OVFL, stack, *top, *bottom, ofp ); /* PGM STOP ! */
    else
    {
       stack[*top += 1] = nodeptr;
       return( *top );
    }
}
```

```
/*-------------------------------------------------------------------------*/
/* pop()                                                                   */
/*      function to remove a node pointer from the top of a given stack.   */
/*-------------------------------------------------------------------------*/

NODE_PTR pop( stack, top, bottom )
NODE_PTR stack[];               /* stack containg the access path */
int      *top, *bottom;         /* pointers to top &bottom indexes */
{
    int  i = *top;

    if ( *top < *bottom )       /* empty */
       return(NULL);            /* return a NULL ptr */
    *top -= 1;
    return(stack[i]);
}
```

```
/*-------------------------------------------------------------------------*/
/* sysfun.c - module contains time manipulation functions: creatsysinfo()  */
/*                                                         settime()       */
/*                                                         gettime()       */
/*                                                         adduser()       */
/*                                                         subuser()       */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "pstdef.h"
#include "pstrec.h"

extern unsigned short  mask[];
extern unsigned short  unmask[];
long           time();   /* a UNIX system call to get the system time */
SYSINFO_PTR    sysinfo;  /* pointer to system table */
int            sid3;     /* shared memory key */

/*-------------------------------------------------------------------------*/
/* creatsysinfo()                                                          */
/*      function to create a systemwide table containing system times.     */
/*-------------------------------------------------------------------------*/

creatsysinfo()
{
   extern char *shmat();

   /* create a systemwide table that contains system times */
   sid3 = shmget( SHMKEY3, sizeof( SYSINFO_PTR ), 0777 | IPC_CREAT );
   if ( sid3 != ERROR )
   {
      sysinfo = ( SYSINFO_PTR ) shmat( sid3, 0, 0 );
      sysinfo->timelock = sysinfo->userlock = UNLOCK;
      sysinfo->current_time = -1;
      sysinfo->base_time = time( 0 ) >> INTERVAL;
      sysinfo->user = 0;
      printf("\nA Sys. Info. Block containing system times has been created\n");
      printf("Current_time = 0\t Base_time = %d\n",sysinfo->base_time);
      return( OK );
   }
   printf("\nError  : Cannot create Sys. Info. Block !\n");
   return( FAIL );
}
```

```
/*-------------------------------------------------------------------------*/
/* settime()                                                               */
/*      function to set system times, namely base and current times.       */
/*-------------------------------------------------------------------------*/

settime( t )
TIME_TYPE     t;
{
   sysinfo->current_time = t;
   sysinfo->base_time = time( 0 ) >> INTERVAL;
}




/*-------------------------------------------------------------------------*/
/* gettime()                                                               */
/*      function to get the current time stamp.                            */
/*-------------------------------------------------------------------------*/

TIME_TYPE gettime()
{
   long       t;

   t = time( 0 ) >> INTERVAL;
   return( ( TIME_TYPE ) ( t - sysinfo->base_time ) );
}




/*-------------------------------------------------------------------------*/
/* adduser()                                                               */
/*      function to add a task to the system in the current time stamp.    */
/*-------------------------------------------------------------------------*/

adduser( task )
unsigned short  task;
{
   lockuser();
   sysinfo->user |= mask[task];
   unlockuser();
}
```

```
/*-------------------------------------------------------------------------*/
/* subuser()                                                               */
/*      function to subtract a task from the system in the current time stamp.*/
/*-------------------------------------------------------------------------*/

subuser( task )
unsigned short  task;
{
   lockuser();
   sysinfo->user &= unmask[task];
   unlockuser();
}
```

```
*-------------------------------------------------------------------------*
* tas()                                                                   *
*      function to provide an interface for high-level routines to access *
*      indivisible test_and_set instruction.  Test_and_set is a special   *
*      hardware instruction that can be used to synchronize multiple tasks*
*      in a concurrent environment.                                       *
*-------------------------------------------------------------------------*
*
.sp     equ     12
.fp     equ     13
.ap     equ     14
        impur
        pure
        align   4
        entry   _tas
_tas    equ     *
.FL1    equ     0
.AL1    equ     00000000
.RL1    equ     40
.FP1    equ     0
.AP1    equ     .FP1+.FL1
.RP1    equ     .AP1+.AL1
.F1     equ     .FL1+.AL1+.RL1
.DAP1   equ     0
        lr      .sp,.fp
        ai      .fp,.F1
*                               * Note : R2, R3 are index Regs.
        l       2,.DAP1+4-4(.ap)* R2 = addr of 1st arg
        ts      0(2)            * testandset MSB of the halfword arg
*
        bm      L13             * L == ? (Is it a locked node ?)
*                               * 0 -- it is not a locked node.  Caller
*                               *   succeeds in locking the node.
        li      0,0             * R0 = 0 (return value stores in R0)
        br      15              * return to caller.
*
L13     equ                     * 1 -- it is a locked node.  Caller
*                               *   fails to lock the node.
        li      0,1             * R0 = 1 (return value stores in R0)
        br      15              * return to caller.
L12     equ     *
        br      15
        impur
        end
```

VITA 2

Hack Hoo New

Candidate for the Degree of

Master of Science

Thesis: CONCURRENT OPERATIONS IN PERSISTENT SEARCH TREES

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Republic of Singapore, May 1, 1960, the son of Yu Lik and Soh Kek; married Yew Lan on December 18, 1986.

Education: Graduated from Chung Cheng High School, Republic of Singapore, in December 1978; received Bachelor of Science Degree in Business Administration from Oklahoma State University in May, 1985; completed requirements for the Master of Science Degree at Oklahoma State University in May, 1988.

Professional Experience: Computer Programmer, Hydrologic, Inc., Stillwater, June, 1986 to February, 1988; Research Assistant, School of Civil Engineering, Oklahoma State University, February, 1987 to February, 1988.