Copyright © 1984, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A NEW LOCKING PROTOCOL THAT ACHIEVES ALL SERIALIZABLE EXECUTIONS

by

S. Lafortune, and E. Wong

Memorandum No. UCB/ERL M84/77

25 September 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A NEW LOCKING PROTOCOL THAT ACHIEVES ALL SERIALIZABLE EXECUTIONS*

Stéphane Lafortune

Eugene Wong

Department of Electrical Engineering and Computer Sciences

and Electronics Research Laboratory

University of California, Berkeley, CA 94720

ABSTRACT

This paper considers the concurrency control problem for database management systems. Its purpose is to propose a new locking protocol that allows for maximum concurrency in the sense that it reaches all serial equivalent executions. In addition to this, the protocol permits early detection of deadlocks. The main features of this protocol are the introduction of a third type of locking action (in addition to "lock" and "unlock"), and of a graph that summarizes the "must precede constraints" associated with an execution.

* Research sponsored by the U.S. Army Research Office contract DAAG29-82-K-0091, the National Science Foundation grant ECS-8300463, and in the case of the first author, a scholarship from the Natural Sciences and Engineering Research Council of Canada.

1. INTRODUCTION

It is widely accepted that the appropriate criterion for concurrency control is serializability, i.e., equivalence to some serial execution [1-5]. Of the various ways to achieve serializability that have been proposed, none surpasses the two-phase locking protocol [2] in the degree of concurrency that it affords. Yet, not all serializable executions can be achieved by two-phase locking. The purpose of this paper is to propose a new locking protocol that succeeds in achieving all serializable executions. In addition, this algorithm also provides for early detection of deadlocks.

This paper is organized as follows. In sections 2 to 5, we define the terminology, review two-phase locking and introduce a new locking action ("declare") and a new type of graph that detects violations of serializability along with deadlocks ("must-precede graph"). In sections 6 through 8, we present the "Declare-before-unlock Protocol" and study its properties. We treat extension to "read" and "write" locks in section 9, and deadlock resolution in appendices A and B. Appendix C contains some examples.

2. PRELIMINARIES

We begin with a simple model of concurrency control. Suppose that $a, b, c, \ldots$ denote atomic units of data that we call *objects*. A *transaction* consists of a finite sequence of actions, each action touching a single object. We denote a transaction T_i by

$$T_i = \tau_i(o_1)\tau_i(o_2) \cdots \tau_i(o_{n_i})$$

where $\tau_i(o_j)$ means T_i acts on o_j. For now, we do not distinguish between actions of different types, e.g., reading and writing. Nor do we assume that the objects for the same transaction are distinct.

An *execution* (of a set of transactions) is an interleaved sequence of all the actions from these transactions. For example, let T_1 and T_2 be two transactions:

$$T_1 = \tau_1(a)\tau_1(b) \qquad T_2 = \tau_2(b)\tau_2(c).$$

Then,

$$E = \tau_1(a)\tau_2(b)\tau_2(c)\tau_1(b)$$

is an execution of T_1 and T_2. An execution is said to be *serial* if there is no interleaving. Two executions E_1 and E_2 (of the same transactions) are said to be *equivalent* if for every object o, the subsequence of E_1 touching o is the same as the corresponding subsequence from E_2 [2]. Equivalence is illustrated by the last example as follows. The execution

$$E' = \tau_2(b)\tau_2(c)\tau_1(a)\tau_1(b)$$

is serial, and E' is equivalent to E since for both executions the subsequences touching the individual objects are

$$a : \tau_1(a) \quad b : \tau_2(b)\tau_1(b) \quad c : \tau_2(c).$$

An execution is said to be *serializable* if it is equivalent to some serial execution. It is clear that for our example, any execution of T_1 and T_2 is serializable, since all executions that contain the subsequence $\tau_i(b)\tau_j(b)$ will be equivalent to the serial execution $T_i T_j$, where $i \neq j$, $i, j = 1,2$.

We say that transaction T_i *precedes* T_j in E and denote the condition by $T_i \; p \; T_j$ if there exist some object b and actions $\tau_i(b)$ and $\tau_j(b)$ such that $\tau_i(b)$ comes before $\tau_j(b)$ in E. In such a case, $(\tau_i(b), \tau_j(b))$ will be called a *conflicting pair*. It is well known that an execution E is serializable if and only if "precedes in E" is a partial ordering, i.e. it satisfies

transitivity : $T_i \; p \; T_j$ and $T_j \; p \; T_k \rightarrow T_i \; p \; T_k$
asymmetry : $T_i \; p \; T_j$ and $T_j \; p \; T_i \rightarrow T_i = T_j$.

We can depict the situation by a "precedence graph" PG(E) as follows. Let the nodes of PG(E) represent the transactions. For a given execution E, add a directed arc from T_i to T_j in PG(E) if there is some object b such that T_i acts on b immediately before T_j does so. The execution E is serializable if and only if PG(E) is cycle free. For future reference, we restate this result in the form of a theorem.

Theorem 2.1 [2] : An execution is serializable if and only if its conflicting pairs are consistently ordered. □

Example 2.1 : Consider the following transactions:

$$T_1 = \tau_1(a)\tau_1(b) \quad T_2 = \tau_2(b)\tau_2(c) \quad T_3 = \tau_3(a)\tau_3(c).$$

The execution

$$E = \tau_1(a)\tau_3(a)\tau_1(b)\tau_2(b)\tau_3(c)\tau_2(c)$$

is serializable because its precedence graph PG(E), shown in Figure 2.1, has no cycles. Indeed, it is equivalent to $T_1T_3T_2$. □

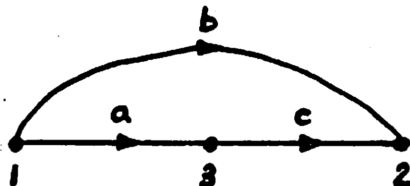


Figure 2.1 - PG(E) of Example 2.1

3. CONTROL BY LOCKING

Locking provides an effective means for achieving serializable executions [3]. In addition to the two basic locking actions "lock" and "unlock," we introduce a third one that we call "declare." For each object in a transaction, the order of the locking actions will be declare-lock-unlock, and each upgrading will void the previous state of lock. We call a transaction with locking actions added an *augmented transaction* while any interleaving of augmented transactions will be termed an *augmented execution*. We represent the locking actions in an augmented sequence by the symbols $d_i(b)$, $l_i(b)$, $u_i(b)$ to denote T_i declares, locks and unlocks object b, respectively. For example, for the execution

$$E = \tau_1(a)\tau_2(a)$$

an augmented execution is

$$E_l = d_1(a)l_1(a)\tau_1(a)u_1(a)d_2(a)l_2(a)\tau_2(a)u_2(a).$$

We now state two sets of *basic locking constraints* which will be referred to frequently in the paper.

An augmented transaction T is said to satisfy the basic locking constraints if:

- every object used by T is declared before it is locked,
- every object is locked before it is used,

- every object is unlocked before the end of the transaction.

- each of these locking actions occurs only once for each object.

An augmented execution E is said to satisfy the basic locking constraints if:

- every transaction in E satisfies the basic locking constraints,

- there is never more than one lock on an object (but there can be more than one transaction holding "declare" on that object).

At this point, we wish to comment further on the new action "declare." A declare must precede a lock, but not necessarily immediately. Once obtained, a declare is kept until it is promoted to a lock, unless the transaction aborts. Upon being promoted, it becomes void. So when we say "all the transactions currently holding a declare on a," we mean all the transactions that have declared but not yet locked object a. In contrast to locks, "declares" do not conflict with each other, neither do they conflict with a lock. This means that any number of transactions can simultaneously hold "declares" on the same object, even if it is locked by another transaction.

An augmented execution will be called a *locking execution* if it satisfies the basic locking constraints. A *serializable augmented execution* is a locking execution such that the subsequence obtained by deleting the locking actions is serializable.

A *partial* (augmented) execution is the truncation of an (augmented) execution. A partial augmented execution is said to satisfy the basic locking constraints if it agrees with the truncation of some locking execution.

Lemma 3.1 : Every serializable execution has an augmentation that satisfies the basic locking constraints, i.e. that is a locking execution.

Proof: Starting from the serializable sequence, we give a procedure for its augmentation and show that the resulting sequence satisfies the basic locking constraints.

Standard augmentation procedure

Let E be an execution: $E = e_1 e_2 \cdots e_n$. We begin by augmenting e_1 in E with the

addition of locking actions to produce E_1. Then, we augment e_2 in E_1 to produce E_2, and so forth. At the beginning of the k-th step, we have e_k in the form of $\tau_T(a)$ where T is a transaction and a is an object. In E_{k-1}, one of the following situations prevails.

(a) At the time of e_k, T holds the lock on a; in this case, we set $E_k = E_{k-1}$.

(b) At the time of e_k, a is unlocked; in this case, we replace $e_k = \tau_T(a)$ in E_{k-1} by $d_T(a)l_T(a)\tau_T(a)$ to produce E_k.

(c) At the time of e_k, some transaction $S \neq T$ holds the lock on a. In this case, we determine the set of all objects acted on by S in the subsequence $e_{k+1}e_{k+2} \cdots e_n$. Say this set is $\{b_1, b_2, \ldots, b_m\}$. Then, we replace $e_k = \tau_T(a)$ in E_{k-1} by

$$d_S(b_1)d_S(b_2) \cdots d_S(b_m)u_S(a)d_T(a)l_T(a)\tau_T(a)$$

to produce E_k.

After step n, we remove all redundant "declares" in E_n (retaining only the first declare on each object for each transaction) to get E_n'. Finally, we add all needed unlocks at the end of E_n' to get E_l, the "standard augmentation" of execution E. □

Remark: The purpose of declaring immediately all remaining objects in case (c) will become apparent in section 6.

Now, we return to the proof of Lemma 3.1. First observe that since the sequence is serializable, all its conflicting pairs are consistently ordered. Therefore, only one lock per transaction per object is necessary, since if T needs object a after S, then necessarily S does not need it anymore after T first uses it. So whenever a transaction needs an object already locked by another transaction, it is always "safe" to unlock it ("safe" meaning the transaction will not need to lock it again later on).

With this in mind, it is easy to see that the augmentation obtained by the above procedure always satisfies the basic locking constraints. □

Example 3.1 : Here is an example of the construction of the "standard augmented execution" for the following (serializable) execution:

$$E = r_2(a)r_3(a)r_1(b)r_2(b).$$

At each step, we obtain the following sequences:

$$E_1 = d_2(a)l_2(a)r_2(a)r_3(a)r_1(b)r_2(b)$$
$$E_2 = d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)r_1(b)r_2(b)$$
$$E_3 = d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)d_1(b)l_1(b)r_1(b)r_2(b)$$
$$E_4 = d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)$$
$$d_1(b)l_1(b)r_1(b)u_1(b)d_2(b)l_2(b)r_2(b).$$

Finally, removing the redundant declare and adding the missing unlocks we get the locking execution:

$$E_l = d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)$$
$$d_1(b)l_1(b)r_1(b)u_1(b)l_2(b)r_2(b)u_3(a)u_2(b). \quad \square$$

4. TWO-PHASE LOCKING

In this section, we consider only two types of locking actions: lock and unlock. An augmented execution is said to be two-phased if the lock-unlock subsequence for every transaction satisfies the condition: "the first unlock occurs after the last lock." The celebrated two-phase lock theorem [2] can be stated in our framework as follows:

if E has a two-phased augmentation, then it is serializable.

In practice, this means that if each transaction observes the two-phase locking protocol: "release no lock until all needed locks have been obtained," then serializability is guaranteed provided the augmented execution is a locking execution and no deadlock occurs. However, deadlocks are possible. In particular, if the actions already taken preclude serializability, then deadlock must occur. For example, consider two transactions

$$T_1 = r_1(a)r_1(b) \qquad T_2 = r_2(b)r_2(a)$$

and a partial augmented execution

$$E = l_1(a)r_1(a)l_2(b)r_2(b)$$

Since the subsequence $r_1(a)r_2(b)$ precludes serializability, deadlock must occur. Indeed, we see that the two-phase condition requires that neither T_1 nor T_2 releases the lock it holds until it obtains a lock held by the other transaction.

Two-phase locking cannot achieve all serializable executions. That is, a serializable E may have no augmentation that is both a locking execution and two-phased. For example, consider the execution of Example 3.1:

$$E = \tau_2(a)\tau_3(a)\tau_1(b)\tau_2(b)$$

which is equivalent to the serial order $T_1T_2T_3$. In any augmentation of E, $u_2(a)$ must precede $\tau_3(a)$, which precedes $\tau_1(b)$ which in turn must precede $l_2(b)$. Hence, $u_2(a)$ must precede $l_2(b)$ and the augmented execution cannot be two-phased.

The simplicity of this example suggests that the degree of concurrency achieved by two-phase locking is significantly less than that required to maintain serializability.

5. GRAPH REPRESENTATION OF CONCURRENT TRANSACTIONS

In this section, we define and compare three different directed graphs for the representation of a set of concurrent transactions that is being executed on-line according to some locking protocol. Here, we consider the three locking actions described in section 3, i.e. declare, lock and unlock. We assume the corresponding (possibly partial) execution satisfies the basic locking constraints.

In these graphs, to each transaction will correspond a node. Arcs will be added according to the specifications given below.

Precedence graph (PG)

This graph was defined in section 2. In fact, it corresponds to the (consistent or not) partial ordering of the transactions obtained from their conflicting pairs. In terms of locking actions, this graph can be characterized as follows:

when transaction T locks object a, draw arc $S \rightarrow T$ where S is the last lock-owner of a. (In such a case, we say "T reads a from S.")

Remark: In the following, when an object is not locked, most recent lock-owner will mean the last transaction that held a lock on that object, if there is any.

Must-precede graph (MPG)

For this graph, arcs are added in the following way:

(a) when transaction T declares object a, draw arc $P \rightarrow T$ where P is the most recent lock-owner of a;

(b) when transaction T locks object a, draw arc $T \rightarrow F$ for each F that is currently holding a declare on a.

(Recall that a declare becomes void once the transaction obtains a lock on the object.) In this graph, arcs are labeled by the objects that gave rise to them. There can be many arcs between two nodes, each arc due to a different object.

If there is an arc from S to T in the MPG, then S is called an *immediate predecessor* of T and T an *immediate follower* of S. If there is a path from S to T in the MPG, then S is called a *predecessor* of T and T a *follower* of S.

Wait-for graph (WFG)

This graph is constructed as follows:

(a) when T requests a lock on a (assuming T has already declared a) and a is currently locked by S, draw arc $S \rightarrow T$ ("T is waiting for S");

(b) update graph at each new lock or unlock.

Theorem 5.1 : The precedence graph is a sub-graph of the must-precede graph.

Proof: There is an arc from S to T in the PG only if there exists some object c such that T reads c from S. Necessarily, the augmented sequence must look like:

$$l_S(c) \cdots u_S(c) \cdots l_T(c)$$

with no other transaction locking c between S and T. Because of the basic locking constraints, $d_T(c)$ occurs before $l_T(c)$. If $d_T(c)$ occurs after $l_S(c)$, then the arc $S \rightarrow T$ is added to the MPG at $d_T(c)$; if $d_T(c)$ occurs before $l_S(c)$, then the arc $S \rightarrow T$ is added to the MPG at $l_S(c)$ (recall the way the MPG is constructed). This proves the result. □

Theorem 5.2 : For a serializable augmented execution, the must-precede graph is a sub-graph of the transitive closure of the precedence graph.

Proof: The statement means: $S \rightarrow T$ in MPG implies $S \; p \; T$. According to the method of construction of the MPG, the arc was added only if either one of the following cases occurred:

(a) T declared object c for which S was the most recent lock-owner. The corresponding locking actions must be of one of the two following forms:

$$l_S(c) \cdots d_T(c) \cdots u_S(c) \cdots l_T(c) \quad \text{or} \quad l_S(c) \cdots u_S(c) \cdots d_T(c) \cdots l_T(c),$$

with no $l_{T'}(c)$ between $u_S(c)$ and $d_T(c)$ in the second case, for some $T' \neq S, T' \neq T$. Both cases imply $S \; p \; T$. If intermediate transactions lock c between S and T, $S \; p \; T$ follows by transitivity of the partial order.

(b) S locked object c for which T is holding a declare. Here, the locking actions are of the form:

$$d_T(c) \cdots l_S(c) \cdots l_T(c) \quad \rightarrow \quad S \; p \; T.$$

Again, if intermediate transactions lock c between S and T, the result follows by transitivity. □

Corollary 5.1 : Consider a serializable augmented execution and its associated must-precede graph. If there is a path from S to T in the must-precede graph, then $S \; p \; T$.

Proof: Use Theorem 5.2 and transitivity of the partial order p. □

The next result characterizes serializability in terms of the MPG.

Theorem 5.3 : The must-precede graph of a locking execution is cycle free if and only if this execution is serializable.

Proof:

($\rightarrow$) By Theorem 5.1 the PG is also cycle free and so the partial order p is consistent. Therefore, the execution is serializable.

($\leftarrow$) By Theorem 5.2, the MPG is a sub-graph of the transitive closure of the PG and so is

necessarily cycle free. □

Corollary 5.2 : If there is a cycle in the must-precede graph and the execution is allowed to continue, then there will be a cycle in the precedence graph.

Proof: Follows from Theorems 2.1, 5.1 and 5.3. □

Theorem 5.4 : The wait-for graph is a sub-graph of the must-precede graph.

Proof: If we have the arc $S \rightarrow T$ in the WFG, i.e. T is waiting for S, then necessarily the same arc was added before to the MPG, either when:

i) T declared an object which was already locked by S, or

ii) S locked an object for which T was holding a declare at the time. □

Corollary 5.3 : Cycles in the wait-for graph and in the precedence graph are always anticipated by cycles in the must-precede graph.

Proof: (WFG) From Theorem 5.4 and the fact that an arc is always added first to the MPG because it is only added to the WFG when the lock is actually requested.
(PG) Because, as the proof of Theorem 5.1 shows, an arc is added to the MPG before it is added to the PG. □

6. A NEW LOCKING PROTOCOL

Two new concepts are critical to an understanding of this new locking protocol. They are: first, a new locking action that we called "declare," and second, the must-precede graph introduced in the previous section. We define the "Declare-before-unlock Protocol" by a set of rules the augmented execution must satisfy.

Declare-before-unlock Protocol

(A) The augmented execution must satisfy the basic locking constraints (listed in section 3).

(B) "Declares" and "locks" are granted only if the must-precede graph remains cycle free; specifically:

1) no declare on an object is granted to a transaction that is a predecessor of the most recent lock-owner of the object.

2) no lock on an object is granted to a transaction that has a predecessor currently holding a declare on the object.

(C) A transaction must declare all the objects it needs before it can unlock any object. □

In order to completely specify the protocol, we need to say what actions are appropriate when a request for "declare" or "lock" is rejected. This will be analyzed in the next section. For the moment, we mention that in the case of a rejected "lock," waiting will almost certainly solve the problem, whereas in the case of a rejected "declare," deadlock is inevitable and a procedure for resolution must be undertaken.

We note that condition (C) is similar to the two-phase condition. It is far weaker than two-phase since "declares" conflict with neither "locks" nor each other. They are merely means for exchanging information among the transactions.

Note also that the Declare-before-unlock Protocol does not specify where the declares have to be placed, except that a declare on an object must come before the lock (locking constraint), and that condition (C) has to be satisfied. This affords considerable freedom in requesting declares.

We now prove two main properties of the protocol: (a) correctness - that it generates only serializable augmented executions (Theorem 6.1), and (b) maximum concurrency- that it reaches all serializable executions (Theorem 6.2).

Theorem 6.1 : If no deadlock occurs, the Declare-before-unlock Protocol always produces a serializable augmented execution.

Proof: If an execution is completed and the protocol was observed, then the final MPG must be cycle free (condition (B)) and the augmented execution is a locking execution (condition (A)). By Theorem 5.3, this execution is serializable. □

Lemma 6.1 : Every augmentation of a serializable execution satisfying (A) also satisfies (B).

Proof: By Lemma 3.1, there exists at least one augmentation that yields a locking execution, i.e. that satisfies the basic locking constraints (the "standard augmentation procedure" given in the proof of Lemma 3.1 satisfies (A)). Hence, the lemma is not void. But, by Theorem 5.3, the MPG of any locking execution is cycle free when the given execution is serializable. Hence, (B) is satisfied. □

Theorem 6.2 : Every serializable execution has an augmentation that satisfies the Declare-before-unlock Protocol.

Proof: The standard augmentation procedure given in the proof of Lemma 3.1 satisfies (A) (by Lemma 3.1), (C) (by construction), and (B) by Lemma 6.1. □

Remark: Because of Theorem 6.2 and the fact that not every serializable sequence can be implemented by two-phase locking (as we saw in section 4), the Declare-before-unlock Protocol allows for more concurrency than two-phase locking. Here is an example.

Example 6.1 : In section 4 we showed that the sequence

$$E = r_2(a)r_3(a)r_1(b)r_2(b)$$

had no augmentation satisfying the two-phase locking protocol. But it has many different augmentations satisfying the Declare-before-unlock Protocol, e.g.

$$d_2(a)l_2(a)r_2(a)d_2(b)u_2(a)d_3(a)l_3(a)r_3(a)$$
$$d_1(b)l_1(b)r_1(b)u_1(b)l_2(b)r_2(b)u_3(a)u_2(b)$$

the "standard augmentation" that was obtained in Example 3.1, or

$$d_2(a)d_2(b)l_2(a)r_2(a)u_2(a)d_3(a)l_3(a)r_3(a)$$
$$d_1(b)l_1(b)r_1(b)u_1(b)l_2(b)r_2(b)u_3(a)u_2(b)$$

where for each transaction all the declares precede the first lock. For both cases, the MPG is $1 \rightarrow 2 \rightarrow 3$. □

Remark: Roles of the "declare" action and of the "declare-before-unlock" condition.

The detailed effects of "declare" and of the "declare-before-unlock" (*DBU*) condition are considered in [12], where the concurrency control problem is approached as a control problem for a dynamical system, and where a new state model is proposed for it. This model is then applied to a detailed analysis of the method of control by locking.

We emphasize here the following points. First, the "declare" locking action is a means for predicting future conflicting actions. It allows one to construct the MPG which contains more information than the PG or the WFG. Intuitively, better performance is possible when the transactions declare early, because then a larger fraction of the cycles in the MPG are created at "lock" than at "declare," and these "lock cycles" *can be resolved by waiting*. In the WFG and the PG, all cycles indicate impossibility to continue the execution without violating serializability. None of them can be solved by waiting, in contrast to "lock cycles" in the MPG.

Roughly speaking, the *DBU* condition is a simple condition that achieves the goal of early declaration by delaying unlocking until a transaction has finished declaring its objects. And since "declare" is a non-conflicting locking action, the *DBU* condition is not restrictive in terms of achievable serializable executions, in contrast to the two-phase condition. □

Remark: Cycle verification in the MPG

Observe that when a cycle is created at "declare" in the MPG, the violation of serializability has not yet occured since the PG is still cycle free. It will inevitably occur, but not before all the transactions involved in the cycle have unlocked at least one object. For this reason, when the *DBU* condition is in force, it is not necessary to check for cycles in the MPG at each "declare" request, but only at the first "unlock" by each transaction. □

In implementing the Declare-before-unlock Protocol, one can safely proceed as long as no cycle is detected in the MPG. Any completed execution is serializable. As new transactions arrive, corresponding nodes are added to the MPG. We now analyse when arcs and nodes can be deleted from this graph.

Theorem 6.3 : Cycles in the must-precede graph are unaffected by deleting from the graph a node and all arcs connected to it once the corresponding transaction and all its predecessors have obtained all their locks.

Proof : Consider a transaction T along with its predecessors and followers in the MPG. In this graph, any predecessor or follower is in a path of the form

$$P_n \rightarrow \cdots \rightarrow P_1 \rightarrow T \rightarrow F_1 \rightarrow \cdots \rightarrow F_m .$$

Once T and all its predecessors have obtained all their locks, there can be no arc coming into the corresponding nodes. This is because an arc can only be drawn into a node when the corresponding transaction is holding "declare" on the object under consideration. Hence, T can never be in the path of a cycle. Therefore, the node and the arcs connected to it can safely be removed from the graph. □

Remarks on Theorem 6.3:

1) Observe that once the condition in the theorem is satisfied, the transaction under consideration cannot get any new predecessors.

2) The condition in the theorem may be hard to verify. A stronger but more easily verifiable condition is the following:

> a node and all the arcs connected to it can be deleted from the MPG when the corresponding transaction and all its predecessors are committed.

In such a case, it is clear that no other transaction can become a predecessor of T and so no cycle involving T can occur in the future.

7. DEALING WITH REJECTED REQUESTS

A request for either "declare" or "lock" can be rejected because either one may cause a cycle to be created in the MPG, thus violating serializability. The two situations need to be handled differently.

First, suppose a request by T for "declare" on object a is denied. This happens if and only if in the MPG T is a predecessor of the most recent lock-owner of a, say T'. T must

precede T' according to the MPG, and T' must precede T because of the previous locking action on object a by T'. Deadlock is inevitable in the sense that the execution cannot be continued without violating serializability. Observe that the precedence graph does not yet have a cycle, and if the same deadlock were to occur with two-phase locking, it would be detected much later in the WFG. Thus, when a request for "declare" is denied, some procedure for resolving deadlocks must be invoked. This is discussed in appendices A and B.

Observe that this "early detection" of deadlocks, a consequence of Corollary 5.3, is a highly desirable feature of our locking protocol. In the next section, we shall present a modified version of our protocol that completely eliminates deadlocks provided it is known beforehand which objects each transaction will need.

In the remainder of this section, we deal with the case where a request by a transaction T to lock an object a is denied. This happens if and only if either one of the following cases occur:

case 1: a is locked, but no predecessor of T is currently holding a declare on it;

case 2: a is unlocked, but one or more predecessors of T in the MPG are holding declares on it;

case 3: a is locked, and one or more predecessors of T in the MPG are holding declares on it.

N.B. In cases 2 and 3, it does not matter whether some predecessors have requested or not the lock because "declare" becomes void only when "lock" is granted.

Defining also:

case 4: a is unlocked and no predecessor of T is holding a declare on it;

the diagram representing the permissible transitions among these 4 states is depicted in Figure 7.1.

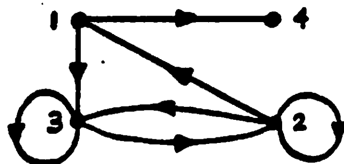


Figure 7.1 - Possible transitions when waiting for a lock

When in states 1, 2 or 3, the simplest solution is to have T to wait until there is a transition to state 4. Then the lock on a can be granted to T without causing a cycle in the MPG. If we are in state 1 and if no predecessor of T declares a before the lock is released, then T will get the lock, i.e. we will have a transition $1 \rightarrow 4$. Otherwise, we have the transitions $1 \rightarrow 3 \rightarrow 2$.

However, states 2 and 3 could imply potential trouble. For instance, it could happen that the number of predecessors of T declaring a increases or that some of these predecessors are themselves waiting for other objects, and so on, so that we have no guarantee that the waiting time for T will be finite. We have modeled the system of Figure 7.1 as a continuous-time Markov chain and found that the system it represents is inherently stable (i.e. returns to 4 in finite time) when the product: (number of concurrent transactions × fraction of the database used on average by a transaction) is less than one.

We have also found a sufficient condition that guarantees T's waiting time will be finite. Before we state it, we need to introduce an extra assumption. From now on, assume a first-come-first-serve policy in granting permissible lock requests is in force, in complement to the "predecessor-first" policy of the Declare-before-unlock Protocol. More precisely,

> when an object is unlocked and one or more transactions have requested a lock on it (call this set R_l), then the lock is granted to the transaction in R_l that has no predecessor in R_l, and if there is more than one such transaction, to the one among these that first requested the lock.

Then the following result holds.

Theorem 7.1 : If, at any time t after transaction T requests "lock" on object a, all the concurrent transactions that act on a and all their current predecessors have

requested all locks that they will ever need, then T will eventually obtain "lock" on a.

Proof: Define the two sets:

$N_a(t)$ = {all the concurrent transactions at time t that act on a}.

$PN_a(t)$ = {all the transactions in $N_a(t)$ and all their predecessors in the MPG at time t}.

and observe that:

(1) The condition in the theorem implies that, after time t, no transaction not in $PN_a(t)$ can become a predecessor of a transaction in $PN_a(t)$. This would occur if and only if that transaction would lock before a transaction in $PN_a(t)$ an object needed by both transactions. But by assumption, all the transactions in $PN_a(t)$ have requested all their locks and we are using a first-come-first-serve policy in granting permissible lock requests. Also, there is always a transaction in $PN_a(t)$ that can be granted a lock when an object becomes available, namely the one in that set which currently has no uncommitted predecessor still using this object.

(2) In particular, (1) implies that the set of transactions that will get a lock on a before T is bounded above by $N_a(t)$ since after t, no new transaction needing a can become a predecessor of any transaction in $N_a(t)$.

The result follows by noting that (1) implies that $PN_a(t)$ is a "closed" set that will clear up in finite time since no deadlock can occur (all the declares have been obtained; see section 8) and there is always at least one transaction in that set that can be executed with no waiting, namely the one at the summit of the sub-graph of the MPG for the transactions in $PN_a(t)$. □

Observe that under the Declare-before-unlock Protocol, the "system" (by this we mean the task of executing transactions) is "stable" in the sense that when transaction initiation is stopped, all transactions will terminate in finite time (if no deadlock occurs). This means a long denied request for lock on an object can always be cleared by suspending initiation of new transactions. In fact, such an action is stronger than necessary and

Theorem 7.1 suggests the following procedure:

Let $O_{PN_a}(t)$ be the set of all objects acted on by the transactions in $PN_a(t)$. At time t, force all the transactions in $PN_a(t)$ to request locks on all the objects they act on. At the same time, do not initiate any new transaction that will act on any object in $O_{PN_a}(t)$ until all the above requests have been made.

Clearly, once all the requests have been made, the sufficient condition of Theorem 7.1 is satisfied and we will eventually have a transition to case 4 for object a.

8. A PROTOCOL FOR DEADLOCK AVOIDANCE

The Declare-before-unlock Protocol affords flexibility in placing the locking actions. As we mentioned in section 6, the occurrence of deadlocks critically depends on where the declares are placed in the augmented execution. Consider the following modified version of the Declare-before-unlock Protocol:

Prior Declaration Protocol

(A) and (B) Same as Declare-before-unlock Protocol.

(C') A transaction must declare all the objects it will act on before its first lock. □

This protocol is just a special case of the Declare-before-unlock Protocol because condition (C') implies condition (C) of the latter. It requires information that is not always available, because the set of objects a transaction acts on might depend on the execution itself. However, it has the following important properties.

Theorem 8.1 : The Prior Declaration Protocol precludes the occurence of deadlock and ensures serializability of the resulting augmented execution.

Proof: That the execution is serializable is due to the fact that the Prior Declaration Protocol is just a special case of the Declare-before-unlock Protocol. To see that deadlocks never occur, observe that when each transaction is in the "declaring phase" (before its first lock), only arcs coming into it are drawn in the MPG. Therefore, the declares will always be

granted because the arcs hereby added will never cause cycles. Cycles will only occur when locks are placed, and we saw in the last section that these cycles do not cause deadlock. □

Theorem 8.2 : Every serializable execution has an augmentation that satisfies the Prior Declaration Protocol.

Proof: Use the standard augmentation procedure of Lemma 3.1 and move the declare actions to the left until (C') is satisfied. Clearly, both Lemmas 3.1 and 6.1 are still true, and thus the result is proved since (A), (B) and (C') are satisfied. □

The authors believe that Theorem 8.1 is a new result, at least in the present context. It is more general than the following well-known deadlock-free scheme: "each transaction must request all needed objects at once and cannot proceed until all have been granted" [5-6], because here the objects need only be declared, not locked, and declares are not exclusive. However, in general, the idea of "declaring" all needed resources (or an upper bound on them) in order to prevent deadlock is not new. For instance, this strategy is used in [7] for the prevention of deadlocks in operating systems.

In some sense, the Prior Declaration Protocol is similar to such concurrency control methods as transaction scheduling or timestamping. By declaring all needed objects before the first lock, a transaction orders itself with respect to some of the other concurrent transactions by identifying those that have to precede it because they have already locked an object it will be acting on. This imposes a partial ordering at the very beginning of a transaction, and no other transaction will then be allowed to precede it (by a lock) if this causes a deadlock, so that the partial ordering will remain consistent. The advantage of this protocol is that in general the ordering is not total. In fact, it is the minimum consistent partial ordering of the concurrent transactions, thus allowing for maximum concurrency.

9. READ AND WRITE LOCKS

In this section, we consider two separate "read" and "write" actions on the objects. We revise the definition of conflicting pair (section 2) by requiring that at least one of the two actions be a write. In other words, two reads by different transactions on an object do not constitute a conflicting pair.

There are at least two reasons for distinguishing reads from writes. The first one is that more concurrency is possible because the class of serializable executions is bigger. In order to allow for this extra concurrency, it is necessary to have two types of locks: write or exclusive locks (xl) and read or share locks (sl). As usual, sl does not conflict with sl, but the three other pairs are conflicting.

The second reason has to do with deadlock, transaction abort and crash recovery. In these instances, for obvious reasons, one would like to avoid rolling-back committed transactions (cascade roll-back). One way of ensuring this if there is only one type of lock is for a transaction to keep all its locks until it is committed (see Appendix B). Our protocol then would have no advantage because its main feature is to allow early unlocking. However, with separate read and write locks, only write locks need be held till commit time. Thus, our protocol enjoys a real advantage even with support for avoiding cascaded roll-back by permitting early release of read locks.

A read lock has no effect on the MPG only if the other transaction involved also wants a read lock. But this information is not available when the other transaction has only declared the object, which is when we need to update the MPG. For this reason, we require that there be two types of "declares," as for the locks:

(i) exclusive declare or xd, to be acquired before a xl; and

(ii) share declare or sd, to be acquired before a sl.

In the MPG, arcs will be added for all pairs:

(sl,xd), (sd,xl), (xl,sd), (xd,sl), (xl,xd), (xd,xl),

but no arcs are added for: (sl,sd), (sd,sl) (corresponding to non-conflicting pair

read/read).

For a share lock to be upgraded to an exclusive lock, it is necessary for the transaction to obtain an exclusive declare on the object. However, downgrading of a lock from xl to sl might lead to an undetected violation of serializability if the declares are placed in a certain manner and if only the last state of lock is considered. Consider the following example.

Example 9.1 : Consider the following partial augmented execution

$$xd_1(b)xl_1(b)xd_2(b)xd_2(c)xl_2(c)xd_3(c)xd_3(a)xl_3(a)$$
$$w_3(a)sl_3(a)r_3(a)u_3(a)sd_1(a)sl_1(a)r_1(a).$$

This execution is not serializable because $1\ p\ 3$ (from $1\ p\ 2$ (object b) and $2\ p\ 3$ (object c)) and $3\ p\ 1$ (object a). However no cycle is detected in the MPG if at $sd_1(a)$ only $sl_3(a)$ is taken into account and not $xl_3(a)$ because by assumption (sd,sl) is not a conflicting pair. □

This example shows that the strongest state of lock held by the most recent lock-owner must be considered when arcs are added to the MPG. As in the basic locking constraints of section 3, a transaction is not allowed to acquire any type of lock on an object after it unlocks it.

With these new locking constraints and this slight change in the construction of the MPG, the Declare-before-unlock Protocol is the same as in section 6. In step (C), "declares" now include both exclusive declares *and* share declares since both can cause cycles to appear in the MPG. It is not hard to verify that the results of Theorems 6.1 and 6.2 are still valid in this new context.

10. CONCLUSION

In this paper, we have presented a new locking protocol for concurrency control that can achieve all serializable executions. Our protocol entails the use of a "must-precede graph" which is in effect an augmented version of the "wait-for graph" used to detect deadlock in conventional locking schemes.

We have shown that by keeping information about "must precede constraints" instead of just "waiting for constraints." we are able to achieve all serializable executions. Even though deadlock can still occur, it is detected earlier in the MPG in comparison with when cycles appear in the PG (when serializability is violated). Also, all cycles in the WFG are anticipated long before by cycles in the MPG. The "must precede constraints" are the consequence of a new locking action we introduced, which we called "declare."

If for deadlock resolution or recovery purposes it is required that all exclusive or write locks done by a transaction be kept until it is committed, then our protocol is still advantageous with respect to the existing ones by permitting early release of read locks.

Deadlock occurence is directly related to how early the transactions declare. We have shown that if it is known at the beginning which objects each transaction will act on, then not only do we still get maximum concurrency, but no deadlock will ever occur. The "declare-before-unlock" condition is a simple and implementable condition that we propose for achieving the objective of early declaration.

An interesting direction for future work on this subject would be the formulation of a probabilistic model for "control by locking" in order to study the performance of the protocols discussed in this paper. For instance, deadlock occurrence and waiting time for denied lock request are of special interest. Some work has been done along those lines. References [8-9] contain recent work on the modeling of locking in the special case where all locks are requested at the beginning and released simultaneously at the end. Finally, some modeling and simulation results for two-phase locking are available in [10-11].

APPENDIX A : A PROCEDURE FOR ROLL-BACK

In this appendix, we assume the protocol in use is the Declare-before-unlock Protocol and we want to analyse how we can get out of a deadlock. For simplicity, we will consider only one type of lock. Extension to share and exclusive locks is discussed at the end.

As mentioned before, a deadlock occurs when $d_T(b)$ is not granted because a follower of transaction T has locked object b earlier. In fact, more than one follower of T can have done so; but they are all on the same path:

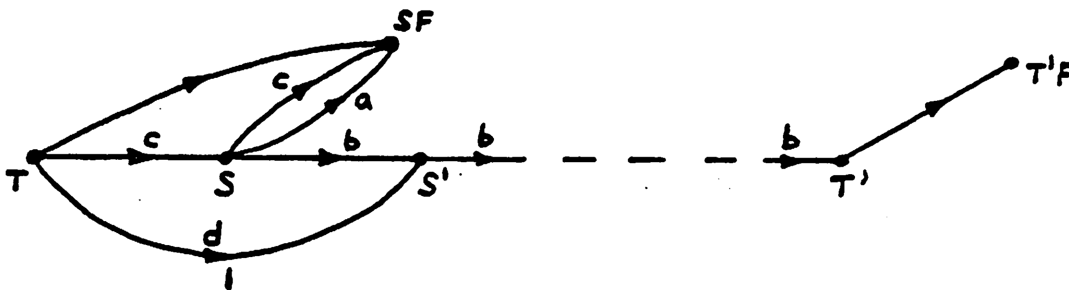


Figure A.1

A deadlock is detected at $d_T(b)$, with T' the current lock-owner of b. S became a follower of T by

$$l_T(c) \cdots d_S(c) \quad or \quad d_S(c) \cdots l_T(c)$$

with no intermediate locks on c.

Observe that T has not unlocked any object yet for it is required to declare all the objects it needs before it can unlock. This also implies that S is not committed when deadlock occurs because it is still waiting for object c which is currently locked by T.

One possible way out of the deadlock would be to make S a predecessor of T so that $d_T(b)$ could be granted. Specifically, all the transactions on the path between T and T' that are immediate followers of T (this could include T') would have to become predecessors of T. This means many modifications to the MPG and is a "dangerous game." For example, arcs like 1 above would also have to be reversed. Since a reasonable objective is to make as few modifications as possible to the MPG, this is not a good policy. Moreover.

we want to give priority to predecessors which is not what this does.

A better way out of the deadlock would be to force T to lock b before S. In order to implement this policy, we have to:

(a) roll-back S from its current state to the point where it first locked an object needed but not yet declared by T. (We do this in case there is another object e locked by S before and which will need to be declared by T later on, provided of course this information is available. Otherwise, we roll-back until $l_S(b)$.) As we will see, this implies the roll-back of some followers of S as well, in particular those that also locked b.

(b) put all other transactions on waiting and have T declare all remaining objects it has in common with one of its followers (not only b, for precaution). (Some declares might not be granted in which case we have to roll-back within the roll-back.)

(c) resume execution.

Observations

1- If in rolling-back a transaction we also undo the MPG, then the set of followers of T can change (e.g. S might not be a follower anymore at that point), and it is not clear what will happen when the execution is resumed.

2- Immediate followers of S that became so in the period which is being undone because they locked an object which S unlocked in that period also have to be rolled-back. We then get a chain-reaction so that the number of transactions to roll-back will not be limited to those immediate followers of S.

3- Suppose the arc $S \rightarrow SF$ was added in the past because, e.g. $l_S(c) \cdots d_{SF}(c)$ and reoccured later after $l_S(b)$ because of $d_S(a) \cdots l_S(a) \cdots d_{SF}(a)$; then if we undo $d_S(a)$ and $l_S(a)$, and $d_{SF}(a)$ remains, deadlock can reoccur because $l_{SF}(a)$ can be accepted if it comes before $d_S(a)$ and so $d_S(a)$ causes another deadlock.

This suggests not to undo the declares made by S. In this case, $l_{SF}(a)$ is rejected and SF has to wait until S unlocks a.

4- On rolling-back a transaction and some of its followers.

declares : leave them for the above reason.

locks and unlocks : 5 cases to distinguish. Suppose we want to roll-back S until time t. The subsequences below represent all locks and unlocks involving the objects mentioned that occured after t.

1) $\cdots u_S(c) \cdots$
2) $\cdots u_S(c) \cdots l_{SF}(c) \cdots$
3) $\cdots l_S(a) \cdots$
4) $\cdots l_S(a) \cdots u_S(a) \cdots$
5) $\cdots l_S(a) \cdots u_S(a) \cdots l_{SF}(a) \cdots$

(cases 2 and 5 can be generalized to include more than one follower of S locking c or a).

Remark: We will say "U is an immediate predecessor of V through b" if there exists an arc from U to V that is due to object b.

Cases 1 and 2 : When undoing $u_S(c)$, immediate predecessors or followers of S through object c are not affected since they became predecessors when they locked c or when S declared c (followers when they declared c or when S locked c), and these actions remain. In case 2, transaction SF has to be rolled-back to the point where it locked c. Since $d_{SF}(c)$ remains (by policy) and $l_S(c)$ occured before t, then necessarily SF will continue to be an immediate follower of S when the execution resumes.

Cases 3 and 4 : Undoing these actions does not affect the immediate predecessors of S through object a since they became so when they locked a or at $d_S(a)$. However, the followers of S can change and SF, originally a follower of S because of $d_{SF}(a)$ e.g., can become a predecessor of S if $l_{SF}(a)$ occurs before $l_S(a)$ when the execution resumes.

Case 5 : As for cases 3 and 4 but in addition *SF* has to be rolled-back like in case 2.

5- Another problem we want to investigate is the following. Is it possible to fall in a loop deadlock/roll-back/deadlock/... and never be able to terminate the execution? This is where the roll-back policy has critical importance. Considering that deadlock is a matter of declaring an object too late and that each transaction needs only a finite number of objects, it means that after a finite number number of roll-backs all the transactions will have declared all the objects they need and so from then on deadlock will not occur. For this reason, the policy of never undoing a declare seems a safe one.

Deadlock loops might be caused because the MPG undergoes many changes. Recall cases 3 and 4 above. If a follower of S becomes a predecessor when the execution is resumed and if it has not declared all its objects, then another deadlock can occur. This can be avoided if arcs are not removed from the MPG when actions are undone in order to force the same ordering of the transactions when the executon resumes.

In summary, the above observations suggest that a good roll-back policy would be to "force T to lock b before S" and in implementing this as described in (a), (b) and (c) above, one should 1) not undo declares, and 2) not remove arcs from the MPG.

Two types of lock case

Essentially all the above analysis is true for the case where we have both share and exclusive locks. In this case, deadlock occurs when either a share declare or an exclusive declare is denied. The only difference concerns observation 2 and cases 2 and 5 of observation 4, and is precisely why two types of lock are advantageous: as far as S is concerned, it is not necessary to roll-back followers of S that have locked (either sl or xl) objects S unlocked if S only acquired share locks on these objects. This is because S did not alter the state of these objects.

We have to say "as far as S is concerned" because such a follower of S could have to be rolled-back because of the "chain-reaction" we mentioned above. For instance, consider the following portion of a MPG:

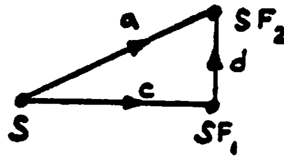


Figure A.2

where object a was only "share locked" by S but object c was "exclusively locked" by S and object d was "exclusively locked" by SF_1. Then roll-back of S can force roll-back of SF_1 if $u_S(c)$ has to be undone and SF_1 acquired any type of lock on c; in turn, roll-back of SF_1 can force roll-back of SF_2 for the same reasons but with object d this time.

APPENDIX B : SIDE-EFFECTS OF ROLL-BACK

The question we want to answer in this appendix is when the roll-back of a transaction has no side-effects, i.e. does not affect other transactions, committed or not, although the emphasis is on the committed ones. Rolling-back a committed transaction (cascade roll-back) is very undesirable and should be avoided. We make no assumptions concerning the cause of the roll-back (e.g. deadlock or transaction abort). For the sake of generality, we consider immediately two types of locks. The one type of lock case is just a special case of this where share locks are never requested.

Lemma B.1 : The roll-back of a transaction T has no side-effects if and only if, at the time the roll-back is started, this transaction has no follower in the must-precede graph that has locked and acted on an object for which T had requested an exclusive lock.

Proof :

(if part) : from the analysis in Appendix A;

(only if part) : suppose the transaction has such a follower in the MPG. This follower is directly affected by the roll-back of T because, after T is undone, either it will have read a non-existent value, or its update of the object will have been lost, or both. Hence, it has to be rolled-back too and so the roll-back of T has a side-effect. We have proved the contrapositive. □

Clearly, if all transactions keep their exclusive locks until commit time, then the condition in Lemma B.1 is satisfied.

APPENDIX C : EXAMPLES

Example C.1 : Consider the simplest type of deadlock with a cycle involving only two transactions in the wait-for graph: "T waits for S" and "S waits for T."

$$T = \tau_T(c)\tau_T(b) \qquad S = \tau_S(b)\tau_S(c).$$

Suppose the actions arrive in the following order:

$$\tau_T(c)\tau_S(b)\tau_T(b)\tau_S(c);$$

(a) with two-phase locking, we get deadlock:

$l_T(c)\tau_T(c)l_S(b)\tau_S(b) \quad | \quad$ *deadlock.*

(b) with the Prior Declaration Protocol, S's request for a lock on b is rejected because it would cause a cycle in the MPG. S has to wait until T unlocks b.

$d_T(c)d_T(b)l_T(c)\tau_T(c)d_S(b)d_S(c) \quad | \quad l_S(b)$ *rejected*
continuation $\quad l_T(b)\tau_T(b)u_T(b)l_S(b)$ (*now granted*)
$\tau_S(b)u_T(c)l_S(c)\tau_S(c)u_S(b)u_S(c)$.

(c) with the Declare-before-unlock Protocol, we can get a deadlock if the declares are placed later:

$d_T(c)l_T(c)\tau_T(c)d_S(b)l_S(b)\tau_S(b)d_S(c)u_S(b) \quad | \quad d_T(b)$ *rejected : deadlock.* □

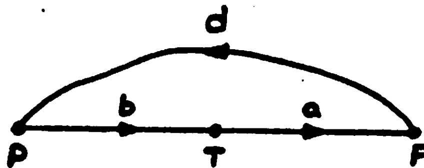
Example C.2 : This example illustrates that a cycle involving T can occur when one of its predecessors has not obtained all its locks. Consider the three transactions

$$P = \tau_P(b)\tau_P(d) \quad T = \tau_T(b)\tau_T(a) \quad F = \tau_F(a)\tau_F(d)$$

and the partial augmented execution

$d_P(b)d_T(b)l_P(b)\tau_P(b)d_P(d)u_P(b)d_F(a)d_T(a)l_T(b)\tau_T(b)l_T(a)$
$\tau_T(a)u_T(b)u_T(a)$ (*Tdone*) $l_F(a)\tau_F(a)d_F(d)l_F(d)$.

This last action causes a cycle to appear in the MPG:



Observe that this cycle can be resolved by having F wait until P is committed before it locks object d. □

Example C.3 : Consider the three transactions

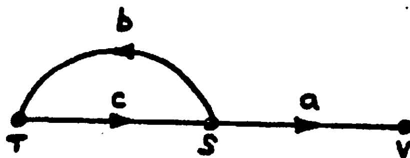
$$T = \tau_T(c)\tau_T(b) \quad S = \tau_S(a)\tau_S(b)\tau_S(c) \quad V = \tau_V(a)$$

and suppose their actions arrive in the following order

$$\tau_S(a)\tau_V(a)\tau_T(c)\tau_S(b)\tau_T(b)\tau_S(c).$$

(a) Declare-before-unlock Protocol with declares placed later:

$$d_S(a)l_S(a)\tau_S(a)d_S(b)d_S(c)u_S(a)d_V(a)l_V(a)\tau_V(a)$$
$$d_T(c)l_T(c)\tau_T(c)l_S(b)\tau_S(b)u_S(b) \quad | \quad d_T(b) \textit{ rejected : deadlock.}$$



(b) two-phase locking. Here we assume the implementation of two-phase locking is such that the request $\tau_V(a)$ will force S to unlock a, and so to request immediately all remaining needed locks. Hence, T's request for a lock on c will be denied until S is done with c:

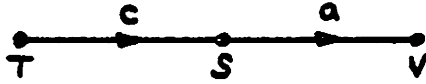
$$l_S(a)\tau_S(a)l_S(b)l_S(c)u_S(a)l_V(a)\tau_V(a) \quad | \quad l_T(c) \textit{ denied}$$
$$\textit{continuation} \quad \tau_S(b)\tau_S(c) \quad | \quad l_T(c) \textit{ can be granted}$$
$$u_S(c)l_T(c)\tau_T(c)u_S(b)l_T(b)\tau_T(b)u_T(c)u_T(b)u_V(a).$$

The fact that S acquired its lock on c early (because it had to unlock a) prevented the deadlock from occuring.

(c) with the Prior Declaration Protocol, S's request for a lock on b is denied and S has to wait until T is done with b:

$$d_S(a)d_S(b)d_S(c)l_S(a)\tau_S(a)u_S(a)d_V(a)l_V(a)\tau_V(a)$$
$$d_T(c)d_T(b)l_T(c)\tau_T(c) \quad | \quad l_S(b) \textit{ denied}$$

continuation $l_T(b)r_T(b)u_T(b) \mid l_S(b)$ *can be granted* $l_S(b)r_S(b)u_T(c)l_S(c)r_S(c)u_S(c)u_S(b)u_V(a)$.□



REFERENCES

[1] H. T. Kung and C. H. Papadimitriou, "An Optimality Theory of Concurrency Control for Databases," *Proceedings of the ACM-SIGMOD 1979 International Conference on Management of Data*, Boston, 1979, pp. 116-126.

[2] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[3] C. J. Date, *An Introduction to Database Systems - Volume II*, Reading, MA: Addison-Wesley, 1983.

[4] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[5] J. Gray, "Notes on Database Operating Systems," in R. Bayer, R. M. Graham and G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, Springer-Verlag, 1978.

[6] E. G. Coffman, jr., M. J. Elphick, A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, Vol. 3, No. 2, June 1971, pp. 67-78.

[7] A. N. Habermann, "Prevention of System Deadlocks," *Communications of the ACM*, Vol. 12, No. 7, July 1969, pp. 373-377, 385.

[8] D. Mitra, P.J. Weinberger, "Probabilistic Models of Database Locking: Solutions, Computational Algorithms and Asymptotics," *Journal of the ACM*, Vol. 31, No. 4, October 1984.

[9] D. Mitra, "Probabilistic Models and Asymptotic Results for Concurrent Processing with Exclusive and Non-Exclusive Locks," to appear in the *SIAM Journal on Computing*.

[10] O. Shmueli, P. Spirakis and N. Goodman, "A Methodology for Concurrency Control Performance Evaluation," Technical Report TR-33-82, Aiken Computation Laboratory, Harvard University, August 1982.

[11] W. K. Lin and J. Nolte, "Performance of Two Phase Locking," *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 16-19, 1982, pp. 131-160.

[12] S. Lafortune and E. Wong, "A State Model for the Concurrency Control Problem in Database Management Systems," Memorandum of the Electronics Research Laboratory, University of California, Berkeley, CA 94720, March 1985.