ORANGE LOCKING: CHANNEL-FREE DATABASE CONCURRENCY CONTROL VIA LOCKING

John McDermott [a] and Sushil Jajodia [b]

[a]Code 5540, Naval Research Laboratory, Washington, DC 20375, USA

[b]Department of Information Systems and Systems Engineering, George Mason University, Fairfax, VA 22030, USA

Keyword Codes: D.1.3;K.6.5;H.2.4

Keywords: Concurrent Programming; Security and Protection; Database Management Systems

Abstract

The concurrency control lock (e.g. file lock, table lock) has long been used as a canonical example of a covert channel in a database system. Locking is a fundamental concurrency control technique used in many kinds of computer systems besides database systems.Locking is generally considered to be interfering and hence unsuitable for multilevel systems. In this paper we show how such locks can be used for concurrency control, without introducing covert channels.

1. Introduction

A database system is a software system that provides a collection of predefined operations with three features: 1) efficient management of large amounts of persistent data (the database), 2) transaction management for transactions composed of those operations on the data (concurrency control, atomicity, and recovery from failure), and 3) a data model that provides a simple abstraction for understanding how the predefined operations and data interact. Our concern is with the second of these features, transaction management.

Transaction management for conventional centralized database systems is fairly well understood and much progress is being made for distributed and federated database systems [18]. Our concern is with transaction management in what we call *multilevel database systems* (which may also be centralized, distributed, federated, etc.). Multilevel database systems assign their data to security classes and restrict database operations based on those classes [7]. The security classes are partially ordered; the data and operations are considered to be in various levels, hence the term multilevel.

A database system that just provides security classes and restrictions on operations is not multilevel. An additional feature of multilevel database systems is their ability to enforce the classes and restrictions in the face of nontrivial attempts to bypass or tamper with the enforcement mechanisms. One of the most difficult challenges for multilevel databases is the *covert channel* problem. A database system user with "low" privileges can obtain information from "higher" security classes by having it leaked into his or her security class from the higher classes, by a Trojan horse or virus, via a covert channel. A covert channel is a means of unauthorized interprocess communication that uses a mechanism not intended for interprocess communication.

The concurrency control lock (e.g. file lock, table lock) has long been used as a canonical example of a covert channel in a database system. Locking is a fundamental concurrency control technique used in many kinds of computer systems besides database systems. Locking is generally considered to be interfering, in the sense of [6,16], and hence unsuitable for multilevel systems.

In this paper we show how locks can be used for concurrency control, without introducing covert channels. We have developed three locking algorithms that do not introduce signalling channels [1] yet they produce serializable transaction histories. Early work on concurrency control for multilevel-secure database systems was done by Hinke and Schaefer, where quasi-synchronization [8] was used to solve the secure readers and writers problem, without channels. Quasi-synchronized histories are not necessarily serializable (not surprisingly, since Hinke and Schaefer's work was coeval with the development of serializability theory [5].) Reed and Kanodia later developed a general mechanism [17] for solving problems similar to the secure readers and writers problem, but it too is unable to guarantee serializability. Previous algorithms that do produce serializable histories for concurrency control, without channels, have relied on timestamping [1,10,11] or are based on subtle properties of a particular database system architecture [3,9,14]. Our orange locking algorithms do not use timestamps and they do not depend on the underlying architecture of the database system. The rest of this paper is organized as follows. First, we discuss transactions, conventional locking, and covert channels. Then we present three locking algorithms: conservative orange locking, reset orange locking, and optimistic orange locking. We show these algorithms to be correct (serializable) and secure (noninterfering), and discuss their deadlock properties. Finally, because of our own interest in the replicated architecture, we show how orange locking can be used to implement immediate-write algorithms for the replicated architecture.

2. Transaction Management in Multilevel Databases

A transaction is an abstract unit of concurrent computation that executes atomically. The effects of a transaction do not interfere with other transactions that access the same data. Also, a transaction either happens with all of its effects made permanent or it doesn't happen and none of its effects are permanent. A useful model of a transaction must show how these properties can be achieved by composing smaller units of computation, when those smaller units are not necessarily guaranteed to compose into an atomic transaction. Thus the model must

1. We distinguish implementation invariant (i.e. inherent in the algorithm itself) covert channels as signalling channels. An implementation of a signalling-channel free algorithm may still have covert channels.

be concerned with showing potential conflicts between operations and with showing arbitrary orderings. Since we are managing transactions for secure database systems, our model must also reflect the security policy enforced by the DBS [13].

In this report we model transactions as sequences of abstract *read*, *read_lock, read_unlock, write*, *write_lock, write_unlock, commit*, and *abort* operations, denoted *r[x], rl[x], ru[x], w[x], wl[x], wu[x]*, *c*, and *a*, respectively. The sequence models the order in which database operations are sent to the transaction management algorithms, without modeling the control structure of transactions themselves.Modeling transactions as sequences is desirable because the sequences can be used in a noninterference [6,16] or restrictiveness [12] model to reason about the security of our algorithms.

Definition 1. A *database* $\boldsymbol{D}$ is a finite set of pairwise disjoint data items that can be operated on by a single atomic database operation. Each data item x in $\boldsymbol{D}$ has a countable domain *dom(x)*. A *database state* is an element of the Cartesian product of the domains of elements of $\boldsymbol{D}$, that is, a state associates a value with each data item in the database. The integrity constraints on a database specify a subset of the Cartesian product, that is, the consistent database states. A *transaction state* of a transaction is an element of the Cartesian product of the domains of the transaction's basis set. A transaction state associates a value with each data item that is read or written by the transaction. A database system state, or *DBS state*, on a history H is a tuple containing a database state as its first element and a transaction state for every transaction in history H. Each database operation maps a DBS state to a DBS state. ❑

Definition 2. A *transaction* T is a finite sequence of database operations from some finite set $\boldsymbol{O}$. Usually $\boldsymbol{O} = \{ r[\bullet], rl[\bullet], ru[\bullet], w[\bullet], wl[\bullet], wu[\bullet], c, a \}$. We will take this read-write model as our definition unless we say otherwise. We denote single operations as one-element sequences thus $r[x]$. Concatenation of sequences is denoted by juxtaposition. If transaction T reads x, then writes x and commits we can denote transaction T as the sequence $r[x]w[x]c_T$. We can also show a transaction as a concatenation of unspecified sequences of database operations, like $T = \alpha r[x]\beta$. We will always use lower case Greek letters to indicate subsequences of database operations.We use λ to denote the empty sequence.

If a sequence of database operations is a transaction, we further require that if the transaction $T = \alpha p$, that is, the sequence of operations α followed the singleton sequence p, then p must be either c_T or a_T and also that neither c_T nor a_T be in α. ❑

Definition 3. Let S_1 be a sequence that contains only distinct elements and let S_2 be a sequence that contains only distinct elements. Say that these two sequences are *compatible* if they do not contain inconsistent orderings of elements common to S_1 and S_2. For example, if $a, b \in S_1$ and $a, b \in S_2$ and if S_1 orders a and b as $a<b$ and S_2 orders a and b as $b<a$, then S_1 and S_2 are not compatible sequences. Let *image* S_1 denote the set of elements in sequence S_1. Define the *shuffle* of two compatible sequences S_1 and S_2, denoted $S_1 * S_2$, to be the set of all sequences that contain just the elements of $(image\ S_1) \cup (image\ S_2)$ and contain S_1 and S_2 as subsequences. The extension to the shuffle $S_1 * S_2 * \ldots * S_k$ of more than two compat-

ible sequences is straightforward. ❑

Definition 4. A *history* H over a set of transactions $\boldsymbol{T}=\{T_1, T_2, \ldots, T_k\}$ is an element of the shuffle of $\boldsymbol{T}$, that is H is a sequence in $T_1{*}T_2{*}\ldots{*}T_k$. A *serial history* has every operation of transaction T_i before every operation of transaction T_j (or vice versa), for every pair of transactions (T_i, T_j) in H. ❑

Definition 5. We define two operations $p[x]$ and $q[x]$ to *conflict* if one of them is a write operation. Intuitively conflicting operations do not commute; we get different results if conflicting operations p and q are done in different orders. We say that two transactions conflict if they contain conflicting operations. ❑

We can now define *conflict equivalence* using the notion of conflicting operations.

Definition 6. Two histories H_1 and H_2 are (conflict) equivalent if

1. they are defined over the same set of transactions and operations,
2. for any pair of conflicting operations $p_i[x]$, $q_j[x]$, (i not necessarily distinct from j) such that a_i, a_j are not in H_1, we have $H_1=\alpha_1 p_i[x]\beta_1 q_j[x]\gamma_1$ iff $H_2=\alpha_2 p_i[x]\beta_2 q_j[x]\gamma_2$. ❑

In this discussion we take the view that histories are correct if they are serializable, that is, equivalent to some serial history. Because aborted transactions have no permanent effect on the database state we do not include them in our equivalent serial histories. Because active transactions (i.e. those that have not committed or aborted yet) may abort, we do not include them either. To accommodate this in our equivalence we define the *committed projection* $C(H)$ of a history H to be the history obtained from H by removing operations that belong to uncommitted transactions.

Definition 7. Formally, we say that history H is *serializable* if its committed projection $C(H)$ is equivalent to some serial history. ❑

Definition 8. A *serialization order of a history H* is the order that transactions appear in a serial history that is equivalent to the committed projection of history H. A serial history equivalent to the committed projection of H is not necessarily unique so H may have several serialization orders. ❑

Definition 9. To model the security policy enforced by our database systems we introduce a finite set of *subjects* $\boldsymbol{S}$, and a finite lattice (SC, $\leq$) of *security classes*. The data items in $\boldsymbol{D}$ of our transaction model will be the passive entities of our security model, that is, abstract units of protected computer resources. A subject is an abstract unit of secure computation. We relate subjects to transactions by defining a subject to be a sequence of database operations. A subject may or may not be a transaction. Every transaction will be a sequence of one or more subjects and every subject in our security model will be in one and only one transaction. We may use the terms *higher* and *lower* to refer to a relation between two or more security classes. By higher, we mean strictly greater than and by lower we mean strictly less than. We use a mapping $\lambda : \boldsymbol{D} \cup \boldsymbol{S} \to SC$ to give the security class of every data element and every transaction.

The algorithms we present here apply to database systems that enforce the following security policy:

1. All transactions are single-level. That is, every subject in a transaction has the same security class and we can meaningfully apply our level function to transactions, thus $\lambda(T)$.
2. Subject S is not allowed to read data element $x \in \boldsymbol{D}$ unless $\lambda(S) \geq \lambda(x)$.
3. Subject S is not allowed to write into data element $x \in \boldsymbol{D}$ unless $\lambda(S) = \lambda(x)$.
4. $\lambda(S)$ and $\lambda(x)$ do not change. ❑

Definition 10. If two transactions T_i and T_j have security classes $\lambda(T_i)$, $\lambda(T_j)$ such that $\lambda(T_i) < \lambda(T_j)$, then T_i and T_j are *low* and *high* transactions with respect to each other. We introduce this definition simply for convenience of exposition. ❑

Definition 11. A transaction T_i *reads x from transaction* T_j in history H if

$$H = \alpha\, w_j[x] \beta\, r_i[x] \gamma$$

and $a_j \notin \beta$ and if $w_k[x] \in \beta$ then $a_k \in \beta$. ❑

Definition 12. A *read-down* is a read operation $r[x]$ of a transaction T_i such that $\lambda(T_i) > \lambda(x)$. Data item x is a *read-down data item*. If transaction T_i reads x from transaction T_j and x is a read-down data item, then T_i *reads-x-down from* T_j, and if for some data item x, T_i reads-x-down from T_j, then T_i *reads-down from* T_j. ❑

3. Locking and Channels

Concurrency control via conventional locking is based on the following principle: 1) each operation that is to be scheduled includes a (possibly implicit) lock request, and 2) if a transaction requests a lock $pl_i[x]$ that conflicts with a lock $ql_j[y]$ that is already set then the requesting transaction is delayed. Two locks $pl_i[x]$ and $ql_j[y]$ conflict if their corresponding operations p and q conflict, $x = y$, and $i \neq j$. Locks can be implemented as a lock table inside the scheduler. Our abstract read lock $rl_i[x]$ is implemented as a lock table entry $\langle x, read, i \rangle$. Transactions that are delayed can be placed on a queue associated with the entry; the mechanism for effecting the delay depends on the underlying operating system. The setting and releasing of locks and the scheduling of operations is done by the scheduler. Transactions request operations and the scheduler returns the results when they are available.

Intuitively, locking should be sufficient by itself to ensure correct database system operation. Unfortunately, it is not. Locking intended to achieve serializability must also be *two-phase*, in the following sense. Transactions that use two-phase locking have a *growing phase* wherein all of a transaction's locks are set and a *shrinking phase* wherein all of its locks are released. A transaction's locks are not necessarily set or released all at the same time, but no lock may be set after a lock has been released. Formally, we say that for any data items x and y, and any transaction T_i, it is always the case that $pl_i[x]$ precedes $qu_i[y]$.

To make recovery from failures tractable, two-phase locking algorithms are often designed to be *strict*. A transaction scheduled by a strict two-phase locking algorithm holds all of its write locks until the end of the transaction, and then releases them together.

Conventional locking introduces a signalling channel. If a virus or Trojan horse in transaction T_i wishes to signal information to a less privileged transaction T_j (i.e. T_j runs in a lower security class) it can do so by reading down, from some predetermined data item x such that the security class of x is the same as T_j's security class. Transaction T_i's read request will set a read lock $rl_i[x]$. If transaction T_i now tries to write into data item x, transaction T_j will be delayed by the read lock $rl_i[x]$. By selectively read locking and read unlocking data item x, transaction T_i can leak information to T_j. It is this well-known scenario we wish to prevent.

4. Optimistic Orange Locking (OOL)

Now we show how to use locks for concurrency control, in a way that does not introduce signalling channels. Our first algorithm is optimistic, that is, operations are never delayed by the scheduling algorithm. Instead, when a transaction is ready to commit, the scheduling algorithm checks the schedule to see if it is correct. If not, then some transaction is aborted (to be rerun later) to make the schedule correct.

In our first approach we simply let the high transaction T_j set read locks on low data items as in a conventional, untrusted database system. If a low transaction T_i then tries to set a write lock on one of the same data items, we immediately grant T_i's write lock and change T_j's read-down lock to an *orange lock*, indicating the possibility of an incorrect read.

Low transactions will not be interfered with by high transactions following this approach. However, we have to decide what to do with high transactions that read data via orange locks instead of read-down locks. If we simply inform the transactions of the orange locks but let them read anyway, the transactions will probably be incorrect. The read-down operations will have been invalidated by the conflicting write that was performed in a nonserializable fashion.

We can obtain serializable schedules by simply aborting a transaction whenever its first read down is orange locked. If most transactions only read down on a few data items and transactions are easy to restart, this approach will allow us to correctly schedule them in a simple manner. This approach begins to have problems when the number of read-downs increases or the cost of restarting a transaction is high. We can do better, at the expense of an increase in complexity, by reducing the number of aborts and making restarts easier.

First, we add a *local workspace* for each transaction. The local workspace contains storage for all the values a transaction will read down. We begin each transaction by having it perform all of its read-downs before beginning any processing. After all of the data items in the local workspace are read, the transaction proceeds as a conventional transaction, reading from and writing to the database directly, within the transaction's security class. Any read-downs during processing are performed from the transaction's local workspace.

Definition 13. A transaction T_i has a *home-free point* that it must reach before completing its processing. A transaction T_i has reached its home-free point when all data items x to be read down by T_i are either read locked and read into T_i's local workspace or orange locked and read into T_i's local workspace. ❑

If a high transaction T_i reaches its home-free point without any orange locks it is allowed to proceed. If T_i has an orange lock set before it reaches its home free point, it is aborted. This abort can be made a *lightweight abort*, that is, we do not need to resubmit the transaction to the scheduler. Instead we can abort by releasing all read-down locks, resetting the local workspace, and moving the transaction's program counter back to the beginning of the transaction. Thus we achieve the effect of a full abort with less overhead. Because we do all our read-downs together, we reduce the length of time we are likely to be interfered with by a low transaction.

Our simple optimistic approach can be shown to be correct because its histories are identical to histories produced by conventional two-phase locking. Transactions that do not conform to the conventional two-phase model are aborted and do not appear in the committed prefix that defines the current stable database state. Our workspace-based improvement is also correct; we will show how later in this paper.

The advantage of this optimistic approach is that we have a relatively simple algorithm, even with our workspace version. We do not have to change our conventional two-phase locking implementation very much. Unfortunately, we get poor performance if lock contention is heavy and we can get also get starvation as a high transaction's read-down locks are repeatedly set to orange, forcing the high transaction to restart.

Remark. The potential for infinite overtaking suggests a possibility for denial of service. While this is theoretically true, it is of no practical concern. A more effective denial of service attack can be mounted with crude techniques such as resource exhaustion.

5. Conservative Orange Locking (COL)

If aborts and restarts are too expensive, but we still want to maintain correctness for our transactions, we can do so by using a conservative approach. Our approach is not conservative in the usual sense because it can still deadlock. Our approach is conservative in the sense that it does not need to abort any transactions for concurrency control reasons and also because it tries to avoid any possible missteps in its approach to scheduling.

In the OOL scheduler, we lock and schedule operations as we normally would, except we cannot delay low write operations to ensure correct read-down operations. In OOL we give up on the high transaction as soon as we detect a conflict with a low write operation. We can do better than this by trying to save the high transaction instead of aborting it. In the conservative orange locking approach, we will use the orange locks to identify a current low transaction that we can safely read from, thus we do not have to give up if a low transaction has a conflict with a read-down. To do this, we may have to resubmit some read-down operations that were invalidated by low transactions. In fact, we can sometimes do even better than rereading invalidated read-downs. If we override a read-down lock into an orange lock *before* we schedule the associated read-down operation, we can delay that read operation until we have identified the proper low transaction to read from. We will avoid performing an invalid read in the first place.

We continue with some data structure definitions. We will unavoidably use some terms before they are defined; we ask the reader to trust that all meanings will be resolved as quickly as possible.

5.1 Local Workspace

Each transaction has a local workspace that is used to hold the values of data items the transaction needs to read-down. The local workspace is used in the same way as in optimistic orange locking; any read-downs during processing are performed from the transaction's local workspace. In conservative orange locking, each read-down data item in the local workspace can be marked *read* or *unread*. These markers are used to determine when a transaction has reached its home-free point. Since a high transaction does not give up when it finds one of its read-down operations has been invalidated, the transaction must know which data items to reread or delay on in order to get a valid view of the database.

5.2 Read-Down Queue

The scheduler associates every transaction with a transaction-specific queue Q_i, called a *read-down queue*. Whenever a high transaction T_j must repeat or defer one of its reads, it does so in order to read from a currently active low transaction T_i. To do this, the scheduler places transaction T_j on the low transaction T_i's read-down queue to wait for T_i to write the necessary value. Management of the read-down queues is done by the scheduler. A low transaction T_i is not even aware of the existence of its corresponding read-down queue Q_i.

Along with low transaction T_i's read-down queue, the scheduler keeps a list W_i of values written by the corresponding transaction. For efficiency, this list W_i may be incorporated into the database system's cache and recovery log, depending on their implementation. When low transaction T_i commits, the scheduler services the reads requested by any high transaction T_j that was placed on T_i's read-down queue. The values returned are taken from list W_i. (To preserve recoverability, cascadelessness, and strictness, the scheduler should not make orange locked data items in W_i available for reading via Q_i until transaction T_i has committed.)

5.3 Conservative Orange Locks: Overriding Read Locks for Read-Downs

The heart of our conservative approach is the way we use orange locks. Instead of passively marking data items, our orange locks actively affect the individual scheduling of reads and writes. Whenever a low transaction T_i needs to obtain a write lock on a data item x that is being read by a high transaction T_j, the scheduler tries[1] to *override* the high transaction T_j's read-down of x. On behalf of transaction T_j, the scheduler converts T_j's read lock to an orange lock on data item x. Whenever a data item x is orange locked on behalf of transaction T_j, data item x in T_j's local workspace is also marked *unread* by the scheduler. Even if data item x had previously been read it is still marked *unread*. A high transaction T_j that has its read lock converted to an orange lock is placed on the appropriate read-down queue to wait for the overriding low transaction T_i to complete. At the same time, all of the read-down data items in T_i's write set are also orange locked and thus marked *unread*. At this point we say that transaction T_j is *orange locked into transaction* T_i. If T_i commits then T_j will read-down from transaction T_i every data item in the write set of transaction T_i that T_j reads. If this happens then the override is considered to have occurred. If instead transaction T_i aborts, then all of the orange locks that were associated with it must be reset to read locks (the affected data items will all still be unread) and transaction T_j must continue to try to reach its read-down point. If another low transaction T_k tries to write lock data item x and high transaction T_j already holds an orange lock on x then the original orange lock is retained but the low transaction T_k gets its write lock and continues. We state this formally as the *orange locking rule*.

Definition 14. We denote the read set and write set of a transaction T_i as $\boldsymbol{R}_i$ and $\boldsymbol{W}_i$ respectively. We also define the *read-down set of transaction* T_i as the set $\boldsymbol{E}_i$ of all T_i's read-down data items and we also define the *orange-locked set* $\boldsymbol{O}_{ij}$ as the set of all read-down data items that T_i reads down from transaction T_j via an orange lock. If transaction T_i reads x down and transaction T_j converts T_i's read lock on x to an orange lock then $\boldsymbol{O}_{ij}=\boldsymbol{E}_i \cap \boldsymbol{W}_j$. If transaction T_i reads x down and its read lock is not converted but x is in $\boldsymbol{W}_j$ then $\boldsymbol{O}_{ij}$ is empty. We will refer to this condition as the *conservative orange locking rule*, that is if data item x is in $\boldsymbol{E}_i \cap \boldsymbol{W}_j$. then $\boldsymbol{O}_{ij}=\boldsymbol{E}_i \cap \boldsymbol{W}_j$ or $\boldsymbol{O}_{ij}=\varnothing$.

5.4 The Conservative Orange Locking Algorithm

Now that we have a clear definition of the override operation, it is possible to talk about how orange locking is used in the algorithm. We give the steps to be followed by a transaction T_i and by the scheduler in serializing T_i's operations.

(1) Transaction T_i declares its read-down set $\boldsymbol{E}_i$ and its write set $\boldsymbol{W}_i$.

(2) The scheduler marks all of T_i's local workspace *unread* and sets Q_i, its read-down queue, to empty.

(3) While some read-down data item in its local workspace is still marked *unread*, transaction T_i submits read-down operations for those unread data items. If the read-down data item is read locked it is read from the database and marked *read* in the local workspace. Otherwise the data item must be orange locked and the transaction reads, via the scheduler's list W_j, from the committed transaction T_j whose write operation required conversion of T_i's read lock into an orange lock. When this step completes, transaction T_i has reached its home-free point.

(4) Transaction T_i now releases the locks on its read-down data items. The read-down locks can be released together in a single operation. Alternatively, if read-

1. The read lock may be released before it is overridden.

down locks are not released together and some low transaction T_j requests a write lock after T_i has reached its home-free point but before the scheduler has released all of T_i's read-down locks, the scheduler simply grants the write lock and schedules the write before releasing the rest of T_i's locks.

(5) Transaction T_i now performs the rest of its processing using conventional strict two-phase locking on data items within its own security class. If transaction T_i needs to perform a write operation on data item x at the same time another transaction T_j needs to read-down x, then T_i will override T_j's read-down by converting T_j's read-lock to an orange lock.

(6) When transaction T_i commits, all of the high transactions that are waiting for T_i on the scheduler's queue Q_i are allowed to read from T_i, via the scheduler's list W_i. At this point transaction T_i will have succeeded in overriding the reads of those higher transactions, thus requiring them to read from list W_i. ❑

A COL scheduler avoids starvation[1] because it selects a specific active low transaction for a high transaction to read from, or it schedules the high transaction to read from the database itself via valid read-downs. It achieves this at the expense of complexity of mechanism. Note that by waiting until the selected low transaction completes, we incur less delay than our intuition would suggest, since we would have had to wait almost as long for the selected low transaction if it had already held the lock.

The serialization order established by a COL scheduler is determined by the home-free points between security classes and by the lock points within the same security class. The home-free point of a transaction must come either before or after the lock point of every conflicting transaction. By holding its read-down locks until its home-free point, conservative orange locking becomes a four-phase protocol. There is a growing and a shrinking phase for read-downs and then a growing and a shrinking phase for intra-class reading and writing.

6. Reset Orange Locking (ROL)

Intuitively, the conservative orange locking rule may seem to be too strong. We would like to do something less than orange lock the entire intersection of a transaction's read-down set and the corresponding update transaction's write set. Fortunately, we can do better than the conservative orange locking rule, if we are willing to return to the possibility of infinite overtaking or starvation. We can do this while still avoiding the need to abort any high transactions that have had read-downs invalidated by low write operations.

In the reset orange locking algorithm, we use the same definitions. Again the local workspace only holds the values the transaction needs to read down. In the ROL algorithm, values to be written are not held in a list W_i by the scheduler and there is no read-down queue Q_i for a transaction. Instead, we can let transactions read down directly from the database.

---

1. An exception to this is the pathological case of infinite overtaking by transactions that abort and restart with no other interleaved transactions committing on the same write set.

6.1 Reset Orange Locks: Resetting Read Locks for Read-Downs

In reset orange locking, just as in COL, low transactions override the read down locks of high transactions. Whenever a low transaction T_i needs to obtain a write lock on a data item x that is being read by a high transaction T_j, the scheduler tries to override the high transaction T_j's read-down of x.

In reset orange locking, the effect of an override is different from COL. First, the scheduler sets low transaction T_i's write lock on data item x and schedules transaction T_i's write operation. Then the scheduler marks data item x in transaction T_j's local workspace as unread. Next, the scheduler releases high transaction T_j's read lock. Eventually transaction T_j requests the scheduler to set it again, by asking for the corresponding read operation. The result of this attempt is that high transaction T_j's read request is queued waiting for a chance to read according to the normal rules of two-phase locking (e.g. it may have to wait for other writes besides T_i's). In the case of reset orange locking, if another low transaction T_k tries to write lock data item x and high transaction T_j has once more obtained its read lock on data item x, the new low transaction *does* override high transaction T_j. This repeated overriding can cause starvation and transaction T_j may never reach its home-free point.

Because a transaction holds all its read-down locks until it reaches its home-free point it is sure to detect (via resetting) any writes that could potentially invalidate a previous or pending read operation.

6.2 The Reset Orange Locking Algorithm

We give the steps followed by a transaction T_i scheduled by ROL and by the ROL scheduler . We follow the same style of exposition to allow comparison with COL.

(1) The scheduler marks all of T_i's local workspace *unread* .

(2) While some read-down data item in its local workspace is still marked *unread* , transaction T_i submits read-down operations for those unread data items. If the read-down data item is read locked it is read from the database and marked *read* in the local workspace. Otherwise transaction T_i's read request is queued waiting for a chance to read according to the normal rules of two-phase locking. When this step completes, transaction T_i has reached its home-free point.

(3) Transaction T_i now releases the locks on its read-down data items. The read-down locks can be released together in a single operation. Alternatively , if read-down locks are not released together and some low transaction T_j requests a write lock after T_i has reached its home-free point but before the scheduler has released all of T_i's read-down locks, the scheduler simply grants the write lock and schedules the write before releasing the rest of T_i's locks.

(4) Transaction T_i now performs the rest of its processing using conventional strict two-phase locking on data items within its own security class. If transaction T_i needs to perform a write operation on data item x at the same time another transaction T_j needs to read-down x, then T_i will override (via the scheduler) T_j's read-down by converting T_j's read-lock to a queued read-lock request. T rans-

action T_i commits according to the rules of conventional strict two-phase locking. ❑

The reset orange locking algorithm is simpler than conservative orange locking. It also does not require declaration of read-down and write sets. In return for this decrease in complexity, we now have the possibility of delays due to multiple overrides.

7. Correctness

Our proofs depend on the following important definition:

Definition 15. We define the home-free point HFP_i of transaction T_i to be the first unlock operation $ru_i[x]$ performed on a data item x that is read down by T_i. We define the lock point LP_i of transaction T_i to be the first unlock operation $qu_i[y]$ performed on a data item y in the same security class as transaction T_i. Intuitively, the lock point of T_i is the conventional lock point associated with strict two-phase locking, as we use it within a security class. For a transaction that does not read down we consider the home-free point to be the lock point. ❑

We will now show the correctness of ROL. Instead of the usual graph theoretic proof, we argue directly towards the definition of conflict serializability.

Given any history H produced by an ROL scheduler, construct from H a serial history H_s as follows: take the committed projection of H and for every pair of transactions (T_i, T_j) in $C(H)$, if

1. the security class of transaction T_i is the same as the security class of transaction T_j, that is, $\lambda(T_i) = \lambda(T_j)$, then put the transactions into H_s in the order that their lock points appear in $C(H)$, or
2. $\lambda(T_i) < \lambda(T_j)$ or vice versa, then put the transactions in H_s in the order that the home-free point of the high transaction appears with respect to the low transaction's lock point in $C(H)$, or
3. some transaction is pairwise incomparable to every other transaction in $C(H)$; put each such transaction at the end of H_s.

The serial history H_s is defined over the same set of transactions and has the same set of operations as $C(H)$. We show that the committed projection $C(H)$ orders pairs of conflicting operations the same as serial history H_s by constructing a chain of equivalent histories starting from $C(H)$ and ending with H_s.

By the definition of conflicting operation and conflict equivalence, we can swap two adjacent nonconflicting operations in a history and the result will be equivalent to the original history. Thus, if the conflicting operations in $C(H)$ and H_s are already in the same order we can transform $C(H)$ into H_s via a finite number of equivalence preserving swaps.

To show that all pairs of conflicting operations are already in the same order, we consider three cases:

Case $\lambda(T_i) = \lambda(T_j)$: Only operations on data items at the same security class conflict, all other operations must be read downs. By the strict two-phase locking of step (4) we know that, for any pair of conflicting operations $p_i[x]$, $q_j[x]$, either

(1) $C(H) = \alpha_1 \, p_i[x] \alpha_2 LP_i \alpha_3 \, q_j[x] \alpha_4$ and

(2) $H_s = \alpha_5 \, p_i[x] \alpha_6 \, LP_i \alpha_7 \, q_j[x] \alpha_8$

or vice versa, depending on which lock point comes first in $C(H)$. Thus all pairs of conflicting operations from transactions in the same security class are ordered the same in $C(H)$ and H_s. ❑

Case $\lambda(T_i) < \lambda(T_j)$: Definition 9 tells us that we can only have conflicting pairs of the form $w_i[x]$, $r_j[x]$, in either order. Suppose we have committed projection

(3) $C(H) = \alpha \, r_j[x] \beta \, w_i[x] \gamma$

for some pair of operations $w_i[x]$, $r_j[x]$. By steps (2), (3), and (4) of the algorithm, the committed projection must be

(4) $C(H) = \alpha \, r_j[x] \beta_1 \, HFP_j \beta_2 w_i[x] \gamma_1 LP_i \gamma_2$

We know that every other pair of operations $w_i[y]$, $r_j[y]$ in $C(H)$ must also be shuffled such that

1. $r_j[y] \in \alpha$ or $r_j[y] \in \delta$, by the definition of home-free point, and
2. $w_i[x] \notin \alpha$ and $w_i[x] \notin \delta$, since transaction T_j will be in its step 2 or step 3 and transaction T_i will be in its step 4.

Thus if one pair of conflicting operations $w_i[x]$, $r_j[x]$ is ordered according to equation (3) then all pairs of conflicting operations are also ordered the same way, which corresponds to the application of serial history construction rule 2: place transaction T_j before transaction T_i in H_s because T_j's home-free point HFP_j is before T_i's lock point LP_i in $C(H)$.

Suppose that the committed projection has some pair of operations $w_i[x]$, $r_j[x]$ in the other order, that is

(5) $C(H) = \alpha \, w_i[x] \beta \, r_j[x] \gamma$

By steps (2), (3), and (4) of the algorithm, the committed projection must be

(6) $C(H) = \alpha \, w_i[x] \beta_1 \, LP_i \beta_2 r_j[x] \gamma_1 HFP_j \gamma_2$

We know that every other pair of operations $w_i[y]$, $r_j[y]$ in $C(H)$ must also be shuffled such that

1. $r_j[y] \notin \alpha$ and $r_j[y] \notin \delta$, since transaction T_j will be in its step 2 or step 3 and transaction T_i will be in its step 4, and
2. $w_i[x] \in \alpha$ or $w_i[x] \in \delta$, by the definition of two-phase locking.

Thus if one pair of conflicting operations $w_i[x]$, $r_j[x]$ is ordered according to equation (5) then all pairs of conflicting operations are also ordered the same way, which corresponds to the application of serial history construction rule 2: place transaction T_i before transaction T_j in H_s because T_i's lock point LP_i is before T_j's home-free point HFP_j in $C(H)$. ❑

Case $\lambda(T_i) > \lambda(T_j)$: The arguments are symmetric to the preceding case. ❑

The correctness proofs for COL and workspace-based OOL are very similar. For simple OOL we merely note that any potentially nonserializable transactions are missing from $C(H)$ and use the proof for conventional two-phase locking given in [2].

8. Security

We argue informally that orange locking is noninterfering. We need to do this because some parts of the algorithm may be implemented as trusted code. We can restrict our discussion to read-down operations because they are the only parts of the algorithms that have the potential to affect the low state of the database system in a way that is interfering. We assume without discussion that no low state variables are changed explicitly by any of the algorithms, including error messages that might report a data item as being locked. Instead we are concerned with delays; that the value of time as a state variable can be made to change according to high inputs (read down requests) to our algorithms. In all three algorithms, if no write is requested while a read-down lock is set, there is no delay possible.

In COL scheduling we must set the low transaction's write lock immediately, before we invalidate the read-down of the high transaction. Likewise, the scheduling of the write operation must precede the orange locking action. If our mechanism for recording values in the list W_i causes a perceptible delay in returning an acknowledgment for the write, then COL scheduling could have a problem. However, the value of a write is usually recorded in a cache or recovery log or both, as part of the normal write process. If not, we can simply make the write operation always put the value in W_i, thus it becomes a constant time operation. We also need to make the action of placing a high transaction on the read-down queue part of the action of setting a write lock.

In ROL scheduling, we must also set the write lock immediately and schedule the write operation right away so as to make the write a constant time operation. The release and resetting of the high transaction's read lock will not interfere with any low transaction. Also, in ROL we do not have to deal with read-down queues and lists of writes, so it is easier to make ROL secure.

OOL scheduling is trivially noninterfering; the scheduler only has to override read-down locks. Since the orange locked transaction will be aborted, there is no problem of getting correct values for the read that is overridden.

9. Deadlocks

Conventional two-phase locking is subject to deadlocks. Two or more transactions can obtain exclusive locks (i.e. write locks) that the others are waiting for and none of the transactions will be able to proceed. This is because the two-phase nature of the algorithm precludes releasing some lock and resetting it later. Deadlocks are usually resolved by restarting one of the transactions involved in the deadlock.

Orange locking has the same potential for deadlocks, within transactions at the same security class, for the same reason. Read-down operations across security classes cannot cause deadlocks at lower classes because read locks can never delay a lower transaction. Transactions that read down via orange locking can be involved in deadlocks because they may also interact with transactions in their own security class.

10. Application to the Replicated Architecture

While orange locking is applicable to kernelized multilevel database systems we are interested in its potential for use in concurrency and replica control for the replicated architecture [4].

The frontend-backend architecture with full replication has been around as a concept for some time [15]. In its SINTRA project, the Naval Research Laboratory is currently prototyping several frontend-backend architectures with full replication. What will be called in this paper the SINTRA architecture was proposed by Froscher and Meadows.

The SINTRA architecture uses full replication to provide multilevel security. There is an untrusted backend database system for each security class. Data from dominated security classes is replicated in each backend system. Logically, the user is allowed to read down and write at the same class but physically the frontend reads all data at the same class and writes at the same class and up into dominating classes to maintain the replicas. It is important to remember that while the replicated architecture uses distributed database system technology, the replicated approach is a centralized architecture. These techniques may be adapted to distributed database systems but not without careful consideration of additional issues. Figure 1 illustrates the basic replicated architecture, for the partial order of Figure 2. Notice that the low data appears at all backends, left data at the left and high backends, etc.

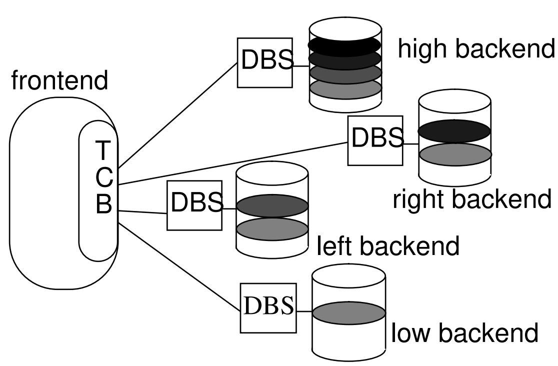

Figure 1. The Replicated Architecture

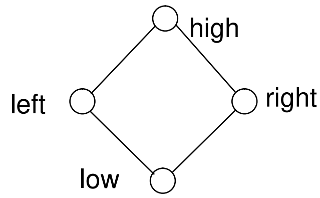

Figure 2. Partial Order for Figure 1

Several *deferred-write* algorithms have been developed for the replicated architecture [3,9,14]. Deferred-write replica control algorithms perform updates on one replica at the time the update is requested and defer the other updates until later. In contrast, *immediate-write* algorithms update all replicas simultaneously.

Immediate-write concurrency control algorithms for the replicated architecture require the concurrency control mechanism in general and the lock table in particular to reside on the frontend. Obtaining a lock on data item x must lock all replicas of x, simultaneously, by a global lock for x acquired on the frontend. This is because the write operations must be sent to the backend databases simultaneously. Since orange locking can provide this kind of concurrency control without introducing signalling channels, it is suitable for immediate-write concurrency control in the replicated architecture.

11. Conclusions

Locking is the preferred mechanism for achieving concurrency control in practical systems. Conventional locking introduces signalling channels. We have shown three different ways of provided channel-free locking for concurrency control: conservative orange locking, reset orange locking, and optimistic orange locking.

The structural differences between these three algorithms are significant. The COL approach avoids the possibility of starvation in the theoretical sense. In the practical sense, it also avoids multiple overrides that could reduce the performance of ROL. COL is structurally more complex than the other two approaches, in both algorithm and data structures. The ROL approach is simpler than COL in algorithm and data structure. It can suffer from multiple overrides of its read locks but it does not need to abort transactions to deal with overrides. The OOL approach without the local workspace has the simplest structure of all. In systems where conflicts are few, this simplicity will give it the best performance of the three. As the level of conflict (number of conflicting operations per transaction) and multiprogramming (number of transactions active at the same time) increases, determining best performance among the three approaches becomes problematic.

The need to declare read-down sets and write sets in COL is not as limiting as it first seems. The declarations prohibit correct scheduling of ad-hoc transactions with COL, but not interactive applications. Many interactive DBS applications are supported through forms, which are compatible with COL scheduling.

Some trusted code may be necessary to implement orange locking. The lock tables themselves may be multilevel objects and should only be accessed by trusted code. How much trusted code is required outside the lock table is also problematic. In some systems it may be possible to implement the lock table as a collection of single-level objects and the lock manager as a collection of single-level processes.

Future work should investigate performance issues in greater depth and look into orange locking implementation architectures with minimal trusted code. Extension of four-phase locking to untrusted systems, with an eye to increasing concurrency, is something else we plan to investigate.

Acknowledgments

We would like to thank Ravi Sandhu and the students in his *Advanced Topics in Computer Security* for their animated discussion of this problem, Oliver Costich for naming the home-free point, and Myong Kang, Judy Froscher, and the anonymous referees for their comments.

References

1. P. Amman and S. Jajodia, "A Timestamp Ordering Algorithm for Secure, Single-Version, Multilevel Databases", in *Database Security V: Status and Prospects*, ed. C. E. Landwehr and S. Jajodia, North-Holland, Amsterdam, 1992

2. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987, ISBN 0-201-10715-5.

3. O. Costich, "Transaction Processing Using an Untrusted Scheduler in a Multilevel Database with Replicated Architecture", in *Database Security V: Status and Prospects*, ed. C. E. Landwehr and S. Jajodia, North-Holland, Amsterdam, 1992.

4. J. Froscher and C. Meadows, "Achieving a trusted database management system using parallelism", in *Database Security II: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1989, ISBN 0-444-87483-6, pp. 253-261.

5. J. Gray, R. Lorie, G. Putzulo, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database", IBM Research Report RJ1654, September 1975.

6. J. Gougen and J. Meseguer, "Unwinding and Inference Control", *Proceedings of 1984 IEEE Symposium on Security and Privacy*, Oakland, CA. pp. 75-86.

7. T. Hinke, "DBMS Technology vs. Threats", in *Database Security: Status and Prospects*, ed. C. E. Landwehr, North-Holland, Amsterdam, 1988, pp. 57-87.

8. T. Hinke and M. Schaefer, *Secure Database Management System*, RADC-TR-75-266, Final Technical Report, System Development Corporation, November 1975.

9. S. Jajodia and B. Kogan, "Transaction Processing in Multilevel-Secure Databases Using Replicated Architecture", *Proceedings of 1990 IEEE Symposium on Security and Privacy*, Oakland, CA, pp. 360-368.

10. T. Keefe, W. Tsai, J. Srivastava, "Multilevel Secure Database Concurrency Control", *Proceedings of Sixth International Conference on Data Engineering*, Los Angeles, CA, February 1990, pp. 337-344.

11. W. Maimone and I. Greenberg, "Single-Level Multiversion Schedulers for Multilevel Secure Database Systems", *Proceedings of Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December, 1990, pp. 137-147.

12. .D. McCullough, “Specifications for Multi-Level Security and a Hook-Up Property”, *Proceedings of 1987 IEEE Symposium on Security and Privacy, Oakland, CA*, pp. 161-166.

13. J. McDermott, O.Costich, M. Kang, “A Formal Model of Secure Transaction Management”, *NRL TM 5540-192*, July , 1992.

14. J. McDermott, S. Jajodia, and R. Sandhu, “A Single-level Scheduler for the Replicated Architecture for Multilevel-Secure Databases”, *Proceedings of Seventh Annual Computer Security Applications Conference*, San Antonio, TX, 1991, pp. 2-11.

15. “Multilevel Data Management”, Committee on Multilevel Data Management, Air Force Studies Board, National Research Council, Washington, DC, 1983.

16. I. Moskowitz and O. Costich, “A Classical Automata Approach to Noninterference Type Problems”, Proceedings of the Computer Security Foundations Workshop 5, Franconia, NH, June 1992, pp. 2-8.

17. D. Reed and R. Kanodia, “Synchronization with Event Counts and Sequencers”, *Communications of the ACM*, 22, 2, February 1979, pp. 115-123.

18. A. Sheth and J. Larson, “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, *ACM Computing Surveys*, 22, 3 (Sep.), 1990, pp 183-236.