Advanced Transaction Processing in Multilevel Secure File Stores

Elisa Bertino[a,1], Sushil Jajodia[b,2] Luigi Mancini[c,3], and Indrajit Ray[b,4]

[a]Dipartimento di Scienze dell'informazione
Università di Milano, Milano, Italy

[b]Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444, U.S.A.

[c]Dipartimento di Informatica e Scienze dell'informazione
Università di Genova, Genova, Italy

Abstract

The concurrency control requirements for transaction processing in a multilevel secure file system are different from those in conventional transaction processing systems; in particular, there is the need to coordinate transactions at different security levels avoiding both potential timing covert channels and the starvation of transactions at high security levels. Suppose a transaction at a low security level attempts to write a data item that is being read by a transaction at a higher security level. On the one hand, a timing covert channel arises if the transaction at the low security level is either delayed or aborted by the scheduler. On the other hand, the transaction at the high security level may be subjected to an indefinite delay if it is forced to abort repeatedly.

This paper extends the two-phase locking mechanism that is suitable for conventional transaction processing systems, to multilevel secure data management systems. The scheme presented here avoids the abort of higher level transactions, nonetheless guaranteeing serializability. The system programmer is provided with a powerful set of linguistic constructs which supports exception handling, partial rollback and forward recovery. The proper use of these constructs can prevent the indefinite delay in completion of a high level transaction, and allows the system programmer to trade off starvation with transaction isolation.

Index Terms—Data management system, file system management, transaction processing, security kernel, concurrency, mandatory access control, two-phase locking, exception handling

Contact Author: Professor Elisa Bertino, Mail Stop 4A4, George Mason University, Fairfax, VA 22030-4444. Telephone: 703–993–1653/1640 (dept.), Fax: 703–993-1638, Internet: ebertino@isse.gmu.edu

[1] Partially supported by Italian M.U.R.S.T. and by Nato Collaborative Research grant number 930888

[2] Partially supported by National Science Foundation under grants IRI–9303416 and INT–9412507 and by National Security Agency under grant MDA904–94–C–6118

[3] Partially supported by Italian M.U.R.S.T.

[4] Partially supported by National Science Foundation under grants IRI–9303416

Advanced Transaction Processing in Multilevel Secure File Stores

Abstract

The concurrency control requirements for transaction processing in a multilevel secure file system are different from those in conventional transaction processing systems; in particular, there is the need to coordinate transactions at different security levels avoiding both potential timing covert channels and the starvation of transactions at high security levels. Suppose a transaction at a low security level attempts to write a data item that is being read by a transaction at a higher security level. On the one hand, a timing covert channel arises if the transaction at the low security level is either delayed or aborted by the scheduler. On the other hand, the transaction at the high security level may be subjected to an indefinite delay if it is forced to abort repeatedly.

This paper extends the two-phase locking mechanism that is suitable for conventional transaction processing systems, to multilevel secure data management systems. The scheme presented here avoids the abort of higher level transactions, nonetheless guaranteeing serializability. The system programmer is provided with a powerful set of linguistic constructs which supports exception handling, partial rollback and forward recovery. The proper use of these constructs can prevent the indefinite delay in completion of a high level transaction, and allows the system programmer to trade off starvation with transaction isolation.

Index Terms—Data management system, file system management, transaction processing, security kernel, concurrency, mandatory access control, two-phase locking, exception handling

1 Introduction

Transactions represent an important functionality that any data management system (either a file system or a database management system) must provide. Conventional and advanced DBMSs provide a transaction management subsystem in charge of transaction synchronization and recovery. Because of the relevance of the transaction paradigm in application development, transaction facilities are now being offered as part of advanced file systems as well. Indeed, a large class of applications exists for which the full power of a DBMS may not be required, but for which reliable, concurrent access to shared data is a crucial requirement. As a result, transaction facilities are now being supported in many file systems such as Camelot [EMS91] and Arjuna [SM94].

While transaction management techniques and algorithms are fairly well understood for data management systems, this is not the case for multilevel secure systems. In such systems, the data and user processes are classified into different security levels, and access to a data item by a process is governed by the following mandatory access rules: A process P can write to a data item x only if x is at the same security level as that of P, and can read x only if x is at a security level lower than or equal to that of P (cf. [BL76]).

The development of a multilevel secure data management system requires a careful revisitation of the architectural components, techniques and algorithms used in a conventional non-secure system. If a system intended to be secure is not designed properly, it may have *covert channels* [Lam73, Den82] which can be exploited by sophisticated intruders to gain illegal access to data.

Secure transaction processing is not easily achieved by the conventional techniques, as these have grave security implications. To illustrate, consider the following example. Let T_i be a high level transaction which is reading a low level data item x, and T_j be a low level transaction which is trying to write to x.[5] Assume that the concurrency control mechanism uses *two-phase locking* (2PL) and, therefore, the transaction T_j will have to wait to acquire a lock on x until such time as T_i releases its lock on x. Suppose T_j can measure the time quantum, q, it has to wait to acquire the lock on x; a quantum of waiting time greater than a certain amount is considered to be a 1 and below that is considered to be a 0. This knowledge can be exploited by T_i to send one bit of high level information to T_j, and by repeating this protocol any arbitrary information can be sent. This indirect way of sending information, against the security policy of the system, is an example of a *timing covert channel.* In order to prevent the class of timing covert channel, that arises from the use of a lock-based protocol for transaction synchronization, a low level transaction cannot be delayed or aborted, because of a lock conflict with a high level transaction.

To date, no research effort has been reported dealing with the problem of secure concurrency control in the context of secure file systems, although this problem has been investigated within the framework of database systems. In [KTS90], Keefe, Tsai and Srivastava present a formal framework for secure concurrency control in multilevel databases. Lamport [Lam77], Reed and Kanodia [RK79] and Schaefer [Sch74] offer solutions to the secure readers/writers problem. While these solutions are secure, they do not yield serializable schedules when applied to transactions (since transactions contain read and write sets that are often related). Moreover, they suffer from the problem of starvation, i.e., transactions that are reading lower level data items may be subject to indefinite delays. Ammann and Jajodia [AJ92] give two timestamp based algorithms that yield serializable schedules; however, both suffer from starvation.

Recently, secure versions of commercial DBMSs are providing concurrency control that is free of timing covert channel; however, the solutions adopted in these systems are not completely satisfactory. For example, the concurrency control algorithm of Trusted Oracle DBMS [Ora92] is based on a combination of 2PL, multiversioning and timestamp ordering. As shown in [JA92], the histories generated by this concurrency control algorithm are not always one-copy serializable. As an additional example, Informix-OnLine/Secure DBMS [Inf93a, Inf93b] uses an approach in which a low level transaction can acquire a write lock on a low data item, even if a high level transaction holds a read lock on this data item. Thus, a low level transaction is never delayed by a high level transaction. The high level transaction simply receives a warning that a lock on a low data item has been 'broken.' However, despite the broken lock, the high level transaction is still committed, unless the programmer has explicitly added some code to perform rollback of the transaction. A major drawback of the Informix approach is that no information is provided to the high level transaction specifying which lock has been broken if the transaction has acquired locks on several low data items. Thus only a very primitive handling of broken lock exceptions

[5] Throughout this paper, we use the terms *high* and *low* to refer to two security levels such that the former is strictly higher than the latter in the partial order.

is possible at the application program level.

The goals of this work are:

1. to provide a multi-level secure file system architecture that supports secure transaction processing, and to isolate the trusted components[6] in such a system;

2. to provide a set of linguistic constructs to support flexible yet powerful systems programming. In particular features for exception handling, partial rollback, and forward recovery are incorporated within the transaction paradigm;

3. to extend strict 2PL on single version data items for concurrency control in secure transaction processing;

4. to avoid the abort and thereby starvation of the high level transaction when a low level transaction acquires a conflicting lock;

5. to guarantee serializable execution of concurrent transactions;

6. to close those timing covert channels that arise due to lock-based transaction synchronization; and

7. to provide a secure hierarchical locking facility in order to reduce the number of conflicts between transactions.

In addition to the usual read and write locks of strict 2PL, our protocol uses an additional lock mode called the *signal lock*, which is acquired by a high level transaction T_i to read a lower data item x. Unlike a read lock, the signal lock does not conflict with a write lock, and hence a low level transaction T_j requesting a write lock on x is never delayed or aborted because of the read by T_i. The system notifies T_i of the existence of the new value for x sometime after the write operation by T_j, by sending a *signal* to T_i. When T_i receives the signal, it decides how to *handle* the signal. T_i can ignore the signal, do a rollback, re-read the data item x or take some other action. Our scheme provides for *delayed signals*, i.e., the system sends all signals generated by T_j together, after the last write operation of T_j, instead of sending a signal immediately after each write operation. In particular, if T_i reads x before T_j writes to it, and commits before the system can notify T_i, no signal is handled (or *serviced*) by T_i. The use of delayed signals not only allows the notions of exception handling and forward recovery, as defined in [Cri82], but also increases concurrency among transactions while preserving serializability, as will be shown subsequently.

The rest of the paper is organized as follows. Section 2 summarizes the security model on which our work is based. Section 3 provides an overview of the approach. In particular, Section 3.1 presents the system architecture. It identifies the trusted and untrusted components of the system as well as their various functions. Section 3.2 defines the three different types of locking operations and also provides the lock compatibility matrix used for

[6] A component is *trusted* if its software operates flawlessly for the system to enforce security policy correctly. This requires developing a formal specification of the component software and a demonstration that the implemented system is consistent with the specification. Since it is impractical to perform the latter task rigorously for large programs, it is necessary that the number of trusted components be kept as small as possible.

locking. Section 4 includes the description of the basic transaction model, with section 4.1 introducing the four new system calls for transaction processing, section 4.2 describing the Trusted Lock Manager and section 4.3 providing two examples of transactions in this model. Section 5 formalizes the notions of a well-formed transaction and histories, and proves that histories consisting of well-formed transactions are serializable. Section 6 describes an optimization of the basic locking protocol with the introduction of multigranularity locking in a hierarchical mode. Section 7 concludes the paper.

2 The Security Model

The multilevel secure system consists of a set D of data items, a set T of transactions (subjects) which manipulate these data items and a lattice S of security levels, called the *security lattice*, whose elements are ordered by the dominance relation $\preceq$. If two security levels s_i and s_j are ordered in the lattice such that $s_i \preceq s_j$, then s_j *dominates* s_i. A security level s_i is said to be strictly dominated by a security level s_j, denoted as $s_i \prec s_j$, if $s_i \preceq s_j$ and $i \neq j$. Each data item from the set D and every transaction from the set T is assigned a fixed security level.

In order for a transaction T_i to access a data item x, the following two *necessary* conditions must be satisfied:

1. T_i is allowed a read access to data item x only if $L(x) \preceq L(T_i)$. In other words the transaction must be at a security level that dominates the security level of the data item x in order for T_i to read x.

2. T_i is allowed a write access to the data item x only if $L(x) = L(T_i)$. That is, the transaction must be at the same security level as the data item x, if it has to write to x.

Note that the second constraint is the restricted version of the $\star$–property which allows transactions to write to higher levels [BL76, Den82]; the constrained version is desirable for integrity reasons.

In addition to these two restrictions, a secure system must guard against illegal information flows through covert channels. The security model addresses only mandatory access control and no discretionary access controls.

3 Overview of the Approach

3.1 System Architecture

The architecture of our multilevel secure transaction processing system can be divided into a trusted, an untrusted, and a passive component as shown in Figure 1. The trusted component consists of two active elements: Trusted Lock Manager and Trusted File Manager. The untrusted component is a collection of Transaction Managers (TMs), one for each security level. The passive component is the File Store which may be physically partitioned according to the security levels.

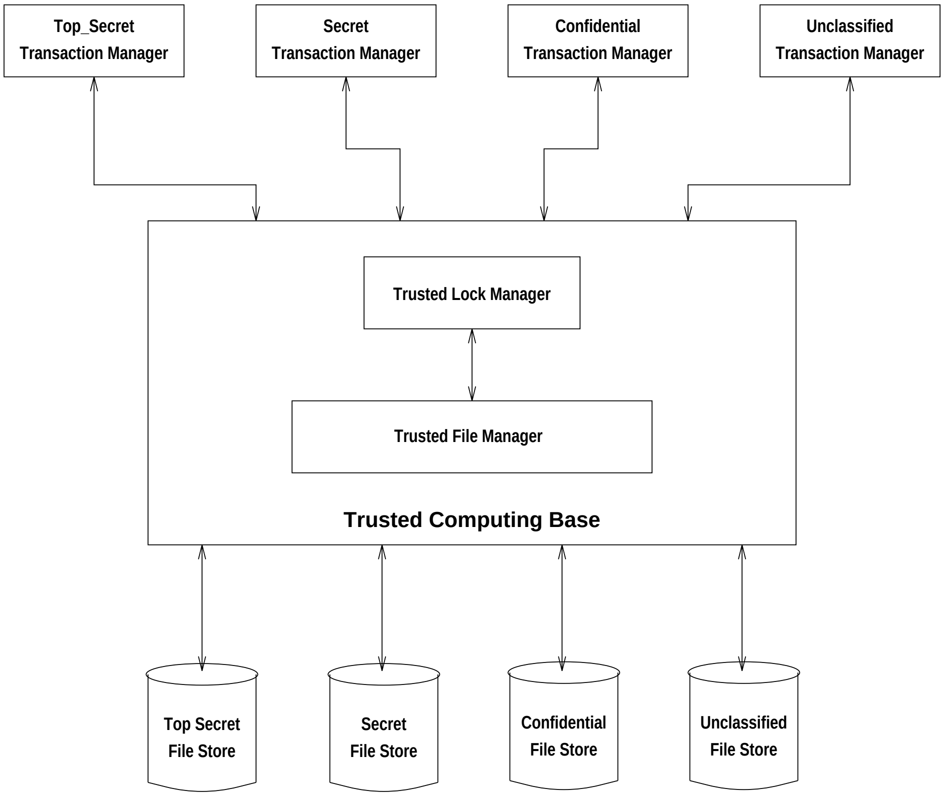


Figure 1: System architecture with four security levels

Note that the assumption about the Lock Manager being trusted can be relaxed by providing one Lock Manager for each security level. A Trojan Horse inside some untrusted Lock Manager can compromise the correct execution of concurrent transactions, but cannot violate security. However, as the whole body of a standard Lock Manager, written with all the requisite defensive programming, exception handlers, optimizations, deadlock detectors, etc. comes to about a thousand lines of actual code [GR93], it is easily verifiable. Thus the assumption of a Trusted Lock Manager, which jeopardizes neither security nor integrity, is justified.

Note also that although we use the term Trusted File Manager, this component need not be fully trusted. We refer the interested reader to [SW88, IAT90, Irv95] for implementation of File Managers that require a small Trusted Computing Base as their only trusted component.

The function of the Trusted Lock Manager is to provide the basic operations of locking and unlocking data items. The functionalities of the Trusted Lock Manager should be designed and implemented so that they cannot be exploited as timing covert channels. Specifically, if a TM_i acquires a read lock on a data item at level s_j, where $s_j \prec s_i$, the execution of such operation should be transparent to all subjects at any level s_k with $s_k \prec s_i$.

Transaction processing is implemented by a cooperation between the Transaction Managers, the Trusted Lock Manager and the Trusted File Manager. The services of the Trusted Lock Manager and the Trusted File Manager can be invoked by each of the untrusted TMs. A TM, on receiving an operation from a transaction, sends a request for the appropriate lock mode to the Trusted Lock Manager. When the Trusted Lock Manager acknowledges that the lock is set, the TM sends the operation to the Trusted File Manager which then is responsible for the actual access to the data item. Our locking protocol closes the timing covert channel that arises from using a lock-based concurrency control algorithm. This is due to the fact that a lower level transaction is never required to wait for a write lock on some data item x even if a high level transaction is reading x.

A transaction T_k at the security level s_i can read information stored at level s_j only if $s_j \preceq s_i$, and can write information only at level s_i. The corresponding TM_i at the level s_i controls the concurrent execution and the recovery of those (and only those) transactions that are at level s_i, and hence TM_i can be an untrusted component of the architecture. In particular, a TM_i that does not set or release locks on data items properly, can cause integrity violations at all the security levels s_j such that $s_i \preceq s_j$. However, this does not lead to a security violation because no information can be passed on to the lower levels from the higher levels through the modulation of the lock/unlock operations. This is a direct consequence of the fact that the Trusted Lock Manager provides the locking operations, transparently.

We require that the Trusted File Manager provide memoryless services on the file stores.[7] By memoryless service we mean that whenever a high level transaction requests any operation (such as open, close, or read) on low data items, the Trusted File Manager executes this operation in such a manner that the utilization of any system resource is not observable to the lower level transactions. In the architecture of Figure 1, a security violation can occur if two TMs establish a timing covert channel either through the Trusted Lock Manager or through the Trusted File Manager. If the Trusted File Manager is memoryless and the request from a TM_i is served by the Trusted Lock Manager transparently with respect to the lower level TMs, then these types of covert channels can be eliminated. The description of such a Trusted File Manager is beyond the scope of this paper. We refer to [SW88, IAT90, Irv95] for description of multilevel file systems that provide file services to transactions at a particular level in a manner transparent to all lower level transactions.

In our work, we discuss the design of the concurrency control mechanism that will close the timing covert channel, arising from the use of locking, as described in Section 1.

3.2 Locking Operations

We define three kinds of locks that a transaction may acquire in order to perform an operation on a data item.

Read Lock If a transaction wants to read a data item which is at its own security level, it has to acquire a *read lock* on the data item.

[7] Strictly speaking, this requirement is not necessary for closing the timing covert channel due to lock-based transaction synchronization. It has been added to close a number of other covert channels that can potentially arise.

Granted Requested	None	Read	Write	Signal
Read	Y	Y	N	Y
Write	Y	N	N	Y
Signal	Y	Y	N	Y

Y = Lock is granted

N = Lock is not granted

Figure 2: Lock compatibility matrix

Write Lock If a transaction wants to write to a data item which is at its own security level, it has to acquire a *write lock* on the data item.

Signal Lock If a transaction wants to read an item which is at a security level lower than that of the transaction, it has to acquire a *signal lock* on the data item.

The lock compatibility matrix as used by the lock manager is given in Figure 2.

The compatibility matrix shows that the signal lock and the write lock are compatible in some cases but conflicting in others. Specifically, if a write lock is requested on a data item x, when it has already been locked by a signal lock, the write lock is granted; however if the signal lock is requested at a time when the write lock is already in effect on the item, the signal lock is refused.

In principle, transactions that are reading down on a data item x do conflict with lower level transactions that are writing to x. However, we cannot delay or abort the low level transaction because doing so will open a timing covert channel. On the other hand, if a low level transaction is writing to x, we can delay the high transactions that wish to read x. It is precisely to implement this policy, a requested write lock does not conflict with a granted signal lock, but a requested signal lock does conflict with a granted write lock. If a low level transaction, T_i wants to acquire a write lock on an item x while a high transaction T_j is holding a signal lock on x, T_i will be granted the write lock; however, if T_j wants to acquire a signal lock on x while T_i is holding a write lock on x, T_j will be denied the lock, and has to wait till such time as T_i releases the write lock. A typical example of interleaving of concurrent transactions T_1 and T_2, called *history*, in this model is given in Figure 3, where $r_i[x]$ and $w_i[x]$ denotes the read and write operations on data item x issued by a transaction T_i. Note that this history is not serializable, as the operations of transaction T_2 follows $r_1[x]$ and precedes $r_1[y]$. As we detail the model, it will be evident how the signal lock together with the proposed system calls remove any inconsistent concurrent execution from the history.

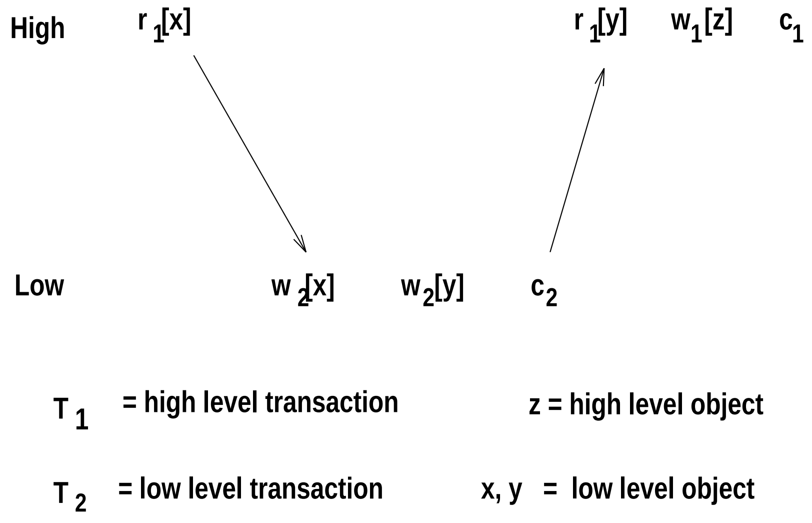


Figure 3: An example history

4 Transaction Processing Model

As mentioned earlier in Section 3.1, transaction processing is achieved by the cooperation between a Transaction Manager, Trusted Lock Manager, and the Trusted File Manager in our model. A transaction consists of a sequence of system primitives, possibly interspersed with some other commands. TM is responsible for the system primitive commands while the run time support of the programming language is responsible for the other commands. TM relies on the Trusted Lock Manager for execution of some of these system primitives.

In this section we first describe the transaction processing system primitives which are used by the programmer to develop a transaction. This description is accompanied by a pseudo-code for a TM, which gives the algorithm for these primitives. Next, we provide a description and algorithm for the Trusted Lock Manager. Finally we give two examples of typical transactions in our model.

4.1 Transaction Processing System Primitives

Before we give the description of the primitives for secure transaction processing, let us recall two assumptions of our basic transaction model. The first assumption is that the system maintains single version of each data item and uses strict 2PL for concurrency control for transactions at the same security level. The second assumption, important from the point of view of security, is that a low level transaction cannot be made to wait for a lock to be released by a high transaction. This has been shown to expose a timing covert channel by which a security breach can take place.

In addition to the basic transaction processing system calls, namely Begin_Transaction, End_Transaction, Read, Write, Commit and Abort, the TM also supports the following four new system primitives:

sl := SaveWork;

Rollback(sl);

RaiseSignal;

GetSignal $[sl_1 \rightarrow handler_1; \ldots ; sl_n \rightarrow handler_n]$;

Figure 4 shows the algorithm for the transaction manager module at any security level l. (Note that the figure deals with only those portions of the module that are relevant to the four system calls.) The thread TransactionRunTimeSupport contains the code for all four transaction processing primitives and is executed for each active transaction as and when one of the primitives is encountered in the transaction code. The other thread of importance to us is the SignalReceiver thread. Its relevance will become evident as we proceed with the description of the model. For now, it is sufficient to note that this thread keeps signals intended for a transaction T_i in a queue associated with T_i.

4.1.1 Description of the SaveWork and RollBack primitives

The SaveWork call establishes a savepoint, which causes the system to record the current state of processing. On encountering a SaveWork command in a transaction's execution, the TM writes on the transaction's log a savepoint record, while the run-time support of the programming language saves the current values of any local variables on the volatile memory. The SaveWork call returns to the transaction a handle in the form of the identifier sl, called a *signal label*. This signal label can subsequently be used to refer to that savepoint. Typically, this handle is a monotonically increasing number. The transaction can return to any savepoint by invoking the RollBack(sl) primitive and passing to it the signal label, sl, of the savepoint to be restored. The execution of this command first restores the state of the system to the state that existed at the time of the savepoint denoted by the signal label sl, and then reexecutes the operations that follow the savepoint. Depending on the application logic, the system programmer can decide to return to the most recent savepoint or to any other.

We assume that the successful execution of the Begin_Transaction primitive establishes the default savepoint for a transaction.

4.1.2 Description of the RaiseSignal primitive

When a transaction T_j at some security level invokes a RaiseSignal primitive, the Trusted Lock Manager sends a signal to each of those higher level transactions that have signal locks on data items being written by T_j. That is, the Trusted Lock Manager considers all the data items x_j's on which T_j holds write locks, and signals the TMs of those uncommitted higher level transactions that have signal locks on these x_j's (for example if a higher level transaction T_i holds a signal lock on x_j, then the TM for T_i is notified by the Lock Manager). Once these higher level TMs have been notified, the Trusted Lock Manager acknowledges the RaiseSignal call to T_j. T_j blocks on the RaiseSignal call until it is acknowledged by the

```
TransactionManager$_l$()                    /* $l$ : security level */
for each active Transaction $T_i$ at level $l$ do
create a queue $SQ_i$
                /* queue for signals from LockManager; contains */
                /* identities of data items whose signal locks */
                /* have been broken */
create a queue $LW_i$ of all operations being executed by $T_i$
⋮
cobegin
⋮
thread TransactionRunTimeSupport$_i$()
    AppendToQueue($LW_i$, CurrentInstruction)
                /* save the current instruction in the queue $LW_i$ */
    case CurrentInstruction of
        Begin_Transaction:
            performs initialization functions and
            establishes the default savepoint.
        ⋮
        Read(x):
            if L($x$) = L($T_i$) then
                wait(LockManager, $T_i$, lock, $x$, read)
                /* read at the same level; wait till read lock is granted */
            else
                wait(LockManager, $T_i$, lock, $x$, signal)
                /* read at a lower level; wait till signal lock */
                /* is granted */
            execute read[x]
        Write(x):
            wait(LockManager,$T_i$,lock,$x$,write)
                /* wait till write lock is granted */
            execute write[x]
        SaveWork:
            $sl_x$ := system generated unique signal label
            AppendToQueue($LW_i$, $sl_x$)
                /* writes the savepoint record on the queue $LW_i$ */
            return($sl_x$)
        Rollback($sl_y$):
            rollback transaction up to a savepoint denoted by $sl_y$ in $LW_i$
```

Figure 4: Transaction manager module (continued on next page)

```
        RaiseSignal:
            Wait(LockManager, $T_i$, RaiseSignal)
                /* sends a request to the LockManager to execute */
                /* a RaiseSignal. Waits until an acknowledgement */
                /* is received from the LockManager */
        GetSignal:
            AppendToQueue($SQ_i$, GetSignalMark)
                /* put a marker in the queue $SQ_i$ to indicate */
                /* that only those signals in the queue before */
                /* and upto the marker will be serviced */
            OldestRead := GetLastElement($LW_i$)
                /* the following while loop selects among all the low */
                /* read operations performed by $T_i$. the earliest */
                /* read whose signal locks have been broken */
                /* At the end of the loop, the variable OldestRead */
                /* contains this earliest read operation */
            while not GetSignalMark in front of queue $SQ_i$ do
                QueueFront := Dequeue($SQ_i$)
                FirstRead := First($LW_i$, read[QueueFront])
                if Precede($LW_i$, FirstRead, OldestRead) then
                    OldestRead := FirstRead
            endwhile
            Dequeue($SQ_i$)
                /* remove the GetSignalMark from $SQ_i$ */
            $sl_x$ := Last($LW_i$, OldestRead, “SaveWork”)
                /* selects the signal label of the savepoint which */
                /* immediately precedes OldestRead in the queue $LW_i$ */
            signal transaction $T_i$ with $sl_x$
    endcase
⋮
thread SignalReceiver()
                /* this thread keeps on listening for a signal */
                /* from the LockManager; when a signal is */
                /* received, it queues the identity of the */
                /* signaled data item in $SQ_i$ */
repeat
    ReceiveSignal(LockManager, $T_i$, $x$)
    AppendToQueue($SQ_i$, $x$)
forever
⋮
coend
⋮
endfor
```

Figure 4: Transaction Manager Module

Dequeue($Queue$)	Return and delete the first element from the queue $Queue$.
Precede($Queue$, $Item1$, $Item2$,)	If $Item1$ occurs before $Item2$ in the queue $Queue$, then return true.
Last($Queue$, Pos, $Item$)	Scan the queue $Queue$ starting backwards from position Pos and stop at the first occurrence of $Item$. Return the element immediately following it in $Queue$.
AppendToQueue($Queue$,$Item$)	Insert $Item$ at the end of the queue $Queue$.
GetLastElement($Queue$)	Return the last element in the queue $Queue$.
First($Queue$, $Item$)	Scan the queue $Queue$ from its front; stop at the first occurrence of $Item$ in $Queue$ and return the position of $Item$.
Wait(TO, T_i, *op*, x, *mode*)	Send a message to TO. The message includes the transaction identifier T_i for the current transaction, the operation *op* to perform (Lock / Unlock), the identifier of the data item x on which to perform the operation and the operation mode $Mode$. Then wait for acknowledgement from TO.
ReceiveSignal(LockManager,TID,x)	Returns the identifier of the signaled data item x, together with the identity, TID, of the transaction that initially had the signal lock on the data item x.

Figure 5: Notation used in Figure 4

Trusted Lock Manager. Eventually, if and when the higher level T_i executes a GetSignal call (which is explained in the next section), its TM will notify T_i about the new value for the low data item x_j. This implementation of the RaiseSignal guarantees that the transaction histories are serializable, as will be shown later in Section 5.

Note that the lower level transaction T_j never waits for either a GetSignal call from a higher level transaction or a higher level transaction to service (or handle) the RaiseSignal call that T_j has just issued; it is upto the Trusted Lock Manager to correctly inform the TMs at higher levels about the RaiseSignal. T_j only waits till it gets an acknowledgement from the Trusted lock Manager. This protocol is implemented in the Trusted Lock Manager avoiding potential timing covert channels. The RaiseSignal call guarantees that if a higher level transaction T_i has read a data item x, which is then modified by the lower level T_j, and if T_i executes a GetSignal after the RaiseSignal of T_j, then T_i will service the signal of T_j.

An issue that can arise in the reader's mind is what happens if the programmer does not issue a RaiseSignal. This will not cause any security violation, although it may lead to consistency problem because the higher level transactions will not be aware of broken signal locks. Later on we will show that RaiseSignal and GetSignal can be made a part of the End_Transaction primitive, and hence they will be automatically invoked. This transparent invocation of the system commands not only reduces the programming effort, but also guarantees a well-formed transaction and hence serializable histories. An alternative to having the RaiseSignal as a part of the End_Transaction primitive is to implement immediate signals instead of delayed signals. What this means is that the Trusted Lock Manager informs any high level transaction that has a signal lock on a low data item x of the new value for x immediately when x is updated by a lower transaction. Consequently, even if the programmer does not issue a RaiseSignal, correctness of the protocol will not be affected.

4.1.3 Description of the GetSignal primitive

The GetSignal primitive is used by a system programmer to specify how signals from lower level transactions are to be serviced by a high transaction.

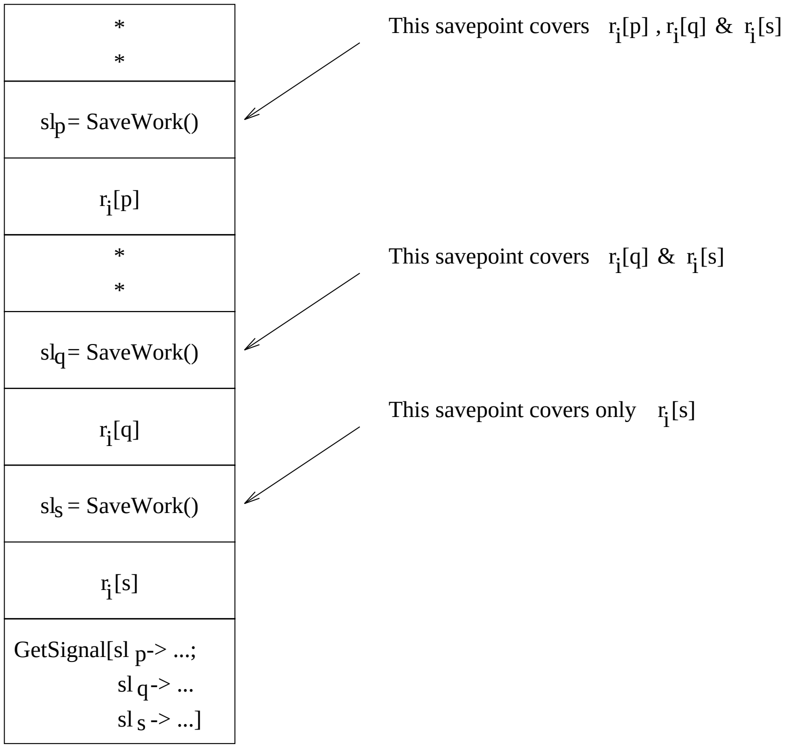
The GetSignal call has two exit points: a standard one which is the next instruction in the transaction body after the GetSignal instruction and an exceptional continuation which is one $sl_i \rightarrow handler_i$ from the expression

$$[sl_1 \rightarrow handler_1; \ldots; sl_n \rightarrow handler_n]$$

Each sl_i represents a signal label and the corresponding $handler_i$ represents a piece of program code to be executed for this signal label. The exceptional continuation is taken when the signal label sl_i is returned by the transaction manager to the GetSignal call.

When a transaction T_i invokes a GetSignal call, its TM considers all the signals that have been received on behalf of T_i, after the last GetSignal invocation by T_i, and selects only one signal to send to T_i. (Recall that the TM for T_i receives these signals from the Trusted Lock Manager.) If the current GetSignal is the first one in T_i, then all signals received since the beginning of execution of T_i are considered.

Three cases may arise at this point: (i) There is no signal to be serviced, (ii) There is only one signal to be serviced or (iii) There are multiple signals to be serviced. If there



*
*
sl_p= SaveWork()
$r_i[p]$
*
*
sl_q= SaveWork()
$r_i[q]$
sl_s = SaveWork()
$r_i[s]$
GetSignal[sl p-> ...;
sl q-> ...
sl s -> ...]
This savepoint covers $r_i[p]$, $r_i[q]$ & $r_i[s]$
This savepoint covers $r_i[q]$ & $r_i[s]$
This savepoint covers only $r_i[s]$

Figure 6: Choosing a signal to be serviced

is no signal from the Trusted Lock Manager, the GetSignal returns a nil value and the computation continues from the next instruction following the GetSignal. If there is only one signal, it indicates to the transaction T_i that the lower level data item x, which was read by T_i, has now a new value. In such a case, the transaction manager TM for T_i returns to the GetSignal call the identifier sl_x of the savepoint that immediately precedes the read of x in the code of T_i, and the exceptional continuation represented by the handler associated with sl_x, is executed.

If there are multiple signals to be serviced, the transaction manager considers all these signals and selects only one to be handled. To understand how this selection of a single signal is achieved, consider the transaction fragment shown in Figure 6. The high transaction T_i in the figure performs three low read operations $r_i[p]$, $r_i[q]$ and $r_i[s]$ in this order. The savepoints immediately preceding these low read operations are labeled sl_p, sl_q and sl_s respectively. The transaction manager for T_i selects the signal label of that savepoint which immediately precedes the earliest low read operation among all the low reads signaled. For example, the transaction manager returns the signal label sl_s, if a signal is raised for $r_i[s]$. If there is a signal corresponding to $r_i[q]$ and $r_i[s]$ only, then the signal label returned would be sl_q. If, on the other hand, there is a signal for each of the three low reads, then sl_p would be raised.

The default invocation for GetSignal is GetSignal[$\rightarrow$ RollBack] which does not specify any signal label. When a signal is to be serviced by this default GetSignal, it rolls back the transaction T_i to the savepoint immediately preceding the earliest read operation among all the reads on the low data items performed by T_i, that have to be redone owing to updates by low transactions. If no savepoint has been explicitly established in the transaction, then this default invocation rolls back T_i, to the default savepoint coinciding with the Begin_Transaction. It should be noted that the GetSignal call is non-blocking, i.e. the call does not wait for the arrival of a signal. If a signal is already available, it is serviced, otherwise no action is taken.

4.2 Trusted Lock Manager

The duties of the Trusted Lock Manager are summarized by the algorithm in Figure 7. Here a request is a tuple of the form ($Sender$, T_i, *op*, x, *mode*), where $Sender$ is the name of the TM sending the request, T_i is the identifier of the transaction that is requiring the operation, *op* is the operation requested on data item x (e.g. lock, unlock, etc), and *mode* denotes the required lock mode. Before setting any lock, the Trusted Lock Manager checks if the security level of the TM and of the transaction are compatible with the lock mode requested for the data item x. It is assumed that the function to determine the security level of a subject, and in particular of the sender of a message, is trusted.

It is worth noting that the signal by the Trusted Lock Manager is a *delayed signal*. The Trusted Lock Manager notifies the relevant high level TMs of all the low-read/write conflicts that have occurred, only when one of the low level transaction invokes a RaiseSignal command. One of the advantages of this scheme based on delayed signals is that if a high level transaction T_j has read a low data item x, which in turn has been written by a low transaction T_i, and T_i invokes RaiseSignal after the last GetSignal of T_j then no signal will be serviced by T_j, and T_j can go ahead and commit. This preserves serializability, as shown in Section 5 because T_j has read the previous committed value of x, which is a correct value. Note that as a consequence the Signal locks can be released earlier than the commit time; in particular, the TMs can release the Signal lock of a transaction right after the last GetSignal returns a nil value. The earlier release of signal locks minimizes the communication between Trusted Lock Manager and the TMs. In the example above the Trusted Lock Manager would provide no signal to T_j. In addition, the delayed signals reduce the overhead due to signal processing. Once a low transaction T_i acquires a write lock on a data item x, the number of higher transactions to be notified can only decrease, not increase. This is because some of these higher transactions may commit before they are notified and all transactions arriving after T_i that want to read x, have to wait till T_i commits. The Trusted Lock Manager can further minimize the number of messages to be sent to a particular TM by grouping together in a single message, all signals to different transactions at the same security level.

4.3 Some Example Transactions

In this section we provide two examples of transactions in our models. The first example shows a transaction that is guaranteed to maintain consistency in the file system. Later on in Section 5, we will show that this transaction conforms to the definition of *well-formedness*

```
TrustedLockManager( )
repeat
Receive(TM$_l$,$T_i$,op,x,mode); /*TM$_l$ is the transaction manager for $T_i$ */
case mode do
    Read / Write:
        If L(TM$_l$)$\neq$L($T_i$)$\neq$L(x) then
        Send(TM$_l$,$T_i$,lockIllegal);
    Signal:
        If L(TM$_l$)$\neq$L($T_i$) OR L($T_i$) $\prec$L(x)
        Send(TM$_l$,$T_i$,lockIllegal);
endcase
case op do
    Lock:
        if mode lock does not conflict with other locks that are already set on $x$
            then SetLock($T_i$,x,mode); Send(TM$_l$,$T_i$,lockOK)
            else Delay($T_i$,x,mode);
    Unlock:
        ReleaseLock($T_i$,x,mode);
        Send(TM$_l$,$T_i$,unlockOK);
        if there is some transaction $T_k$, that had previously requested
        a lock on $x$, but was not awarded the lock, then awake $T_k$;
        if there is a tie, then resolve in favor of lower level transactions.
        SetLock($T_k$,x,mode);
        Send(TM$_{L(T_k)}$,$T_k$,lockOK);
    RaiseSignal:
        let $x_1 ... x_n$ be the data items locked in write mode by $T_i$.
        for each $T_j$ that has a signal lock on $x_k$ do
            notify the TM of $T_j$ of a new value for $x_k$;
        Send(TM$_l$,$T_i$,raisesignalOK);
endcase
forever
```

Figure 7: Trusted lock manager module

Send(TO,TID,$response$)	sends a message to TO; the message contains the identifier of a transaction TID and a $response$.
Receive($FROM$,TID,op,x,$mode$)	receives a service request from $FROM$; the request contains the operation *op* to be performed on behalf of the transaction TID on data item x; if the operation is a lock or unlock operation, then *mode* specifies the locking mode.
SetLock(TID,x,$mode$)	adds to the lock table for TID, a lock of type *mode* on data item x.
ReleaseLock(TID,x,$mode$)	removes from the lock table for TID, the lock of type *mode* on data item x.
Delay(TID,x,$mode$)	puts the transaction TID, waiting for a lock of type *mode*, in the wait queue for data item x.

Figure 8: Notations used in Figure 7

of a transaction in our model. The second example demonstrates the flexibility available to the programmer in developing a transaction, so as to tradeoff consistency for isolation.

Example 4.1 The structure of the transaction is shown in Figure 9. The relevant code for the transaction is delimited by a pair of Begin_Transaction and End_Transaction brackets.

When a transaction T_i is started, an implicit SaveWork corresponding to the Begin_Transaction is executed. Prior to any read or write operation, the transaction manager for T_i, say TM_j, requests the Trusted Lock Manager for the necessary locks; the Trusted Lock Manager in turn grants the lock according to the rules stated in the lock compatibility matrix. During the execution of the transaction, further savepoints can be established as determined by the system programmer. In our case, three additional savepoints have been established in lines 5, 15 and 21. We assume that there are only two low read operations in the transaction viz. $r[x]$ and $r[y]$ in lines 6 and 18.

Suppose that the GetSignal in line 13 of the transaction is currently being executed. If there has been no low level transaction which has updated data item x, then there will be no signal to be serviced, and hence the handler associated with the GetSignal will not be executed. If on the other hand, an update of x is signaled by a low level transaction, then the handler rolls back the transaction to the savepoint denoted by S_1. Consequently, all operations in the transaction, starting from line 13 backwards till line 6 are first undone and then the commands from line 6 to line 13 are re-executed. After this the transaction continues execution from line 14 onwards.

As T_i continues its execution, at some point it performs the RaiseSignal at line 20. When this call is executed, the transaction manager, TM_j, informs the Trusted Lock Manager to signal all those TMs that are at levels higher than TM_j and have transactions with signal locks on z. T_i blocks on the RaiseSignal call until the Trusted Lock Manager acknowledges that all the relevant higher level TMs have been informed. After this acknowledgement, T_i continues from line 21.

```
/* $x$, $y$ are low level data items which can be read but not written to. */
/* z and u are data items at the same security level as the transaction. */
Begin_Transaction
        ...
        ...
        ...
        ...
        $S_1$ = SaveWork ;
        r[x] ;
        r[z] ;
        /* some computation on local program variables */
        ⋮
        ⋮
        /* some computation on local program variables */
        ⋮
        GetSignal[$S_1 \rightarrow$ RollBack($S_1$)];
        ⋮
        $S_2$ = SaveWork ;
        /* some computation on local program variables */
        ⋮
        r[y] ;
        w[z] ;
        RaiseSignal;
        $S_3$ = SaveWork ;
        ⋮
        /* some computation on local program variables */
        ⋮
        GetSignal[$S_1 \rightarrow$ Rollback($S_1$) ;
                  $S_2 \rightarrow$ Rollback($S_2$)];
        ⋮
        r[u];
        GetSignal[ $\rightarrow$ rollback] ;
        /* some computation on local program variables */
End_Transaction
```

Figure 9: A transaction that ensures consistency

Continuing with its execution, T_i executes the GetSignal at line 25. Here four cases may arise: A low level transaction has updated x, a low level transaction has updated y, both x and y have been updated, or there is no signal. If x is updated, then the handler corresponding to signal label S_1 is executed and the transaction rolls back to the savepoint S_1. If y is updated the handler corresponding to signal label S_2 is executed. If both x and y have been updated then the transaction manager signals S_1; this is because S_1 is the savepoint immediately preceding the earliest of the two read operation in T, viz. $r[x]$ and $r[y]$. Finally, if there is no signal the transaction proceeds from line 26 onwards.

In line 28 of the transaction, we have a default invocation of the GetSignal command. If there is any signal to be serviced at this point, the transaction is rolled back to the savepoint immediately preceding the earliest low read operation among those signaled. For example, if both x and y are signaled, then the transaction will rollback to the savepoint S_1. If on the other hand, only y is signaled, then the transaction will rollback to S_2.

Savepoints can also be used in the transaction code merely to control the flow and rollback even without the occurrence of any low level read operation. An example of this is the savepoint S_3 established in line 21.

We would like to mention here, that if the GetSignals on line 13 and 25 are removed from the program code in order to simplify it, the transaction still remains well-formed. We explain why this is so later on in Section 5. □

Example 4.2 The transaction in this example is quite similar to the one in the previous example. Consequently, we describe only those steps that are different in this transaction. In particular, the only differences between the transaction in Figure 10 and that in Figure 9 are in the GetSignal calls in lines 13 and 25.

Consider the GetSignal in line 13. If an update of x is signaled by a lower level transaction, then instead of rolling back the transaction, as in Figure 9, the handler rereads the new value of x and updates the local program variables according to the new value. Note that there is no need to reread the high data item z. After this the transaction continues execution from line 14 onwards. It should be noted that here the programmer has exploited the semantics of the transaction in order to perform a forward recovery instead of a rollback. In particular, the forward recovery is possible because there is no update of the data store between the savepoint S_1 and the GetSignal in line 13.

Consider next the GetSignal in line 25 of Figure 10. If x is updated, then the handler corresponding to signal label S_1 is executed and the transaction rolls back to savepoint S_1. Note here that the programmer has specified a different handler for the signal label S_1 than was used previously with the GetSignal in line 13; the handler in line 13 performed a forward recovery.

On the other hand, if y is updated the handler corresponding to signal label S_2 is executed. In this handler the programmer has specified a conditional rollback. If the total number of rollbacks for this transaction is more than 4 till this point, the programmer does not want another rollback for y. Instead he wants that an alert message be delivered and the execution continue forward from line 26. This construct for conditional rollback, if properly used in all the rollback handlers, prevents the indefinite delay for completion of a high level transaction and thereby allows the programmer to find a trade off between starvation and transaction isolation.

```
/* $x$, $y$ are low level data items which can be read but not written to. */
/* z and u are data items at the same security level as the transaction. */
/* RollBackCount is a user accessible system variable which is */
/* incremented by 1 each time a rollback operation is performed. */
/* This variable is initialized to zero only at the transaction beginning */
Begin_Transaction
        ...
        ...
        ...
        ...
        $S_1$ = SaveWork ;
        r[x] ;
        r[z] ;
        /* some computation on local program variables */
        ...
        ...
        /* some computation on local program variables */
        ...
        GetSignal[$S_1 \rightarrow$ r[x];
                update the local variables according to the new value of $x$ ] ;
        ...
        $S_2$ = SaveWork ;
        /* some computation on local program variables */
        ...
        r[y] ;
        w[z] ;
        RaiseSignal;
        $S_3$ = SaveWork ;
        ...
        /* some computation on local program variables */
        ...
        GetSignal[$S_1 \rightarrow$ rollback($S_1$) ;
                $S_2 \rightarrow$ If RollBackCount $< 4$ then rollback($S_2$)
                        else send appropriate message to user] ;
        ...
        r[u];
        GetSignal[ $\rightarrow$ rollback] ;
        /* some computation on local program variables */
End_Transaction
```

Figure 10: A transaction providing more flexibility

Finally, if both x and y have been updated, then the transaction manager signals S_1 as it is the savepoint immediately preceding the earliest of the two read operations in T. □

5 The Formal Transaction Model

This section formally defines a transaction in our model and presents the notions of histories, rollback-free projections of histories and serializable histories.

Definition 5.1 A transaction T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \Re \cup \mathrm{rs}_i \cup \{\mathrm{a}_i, \mathrm{c}_i\}$ where
    (a) $\Re$ is the powerset of $\{r_i[x], w_i[x], sw_i(sl), gs_i(sl), rb_i(sl) \mid x$ is a data item and sl is a signal label$\}$, where $sw_i(sl)$ establishes a savepoint denoted by the signal label sl, $gs_i(sl)$ is an execution of the command GetSignal, servicing a signal label sl, and $rb_i(sl)$ denotes the command RollBack with the parameter sl.
    (b) rs_i is the command RaiseSignal
    (c) a_i is an abort and
    (d) c_i is a commit
2. $\mathrm{a}_i \in T_i$ iff $\mathrm{c}_i \notin T_i$
3. if t is a_i or c_i, then for any other operation $p \in T_i$, $p <_i t$

The result of the $rb_i(sl)$ command is the execution of a series of $undo(op_i)$ operations, the *duals* of op_i. For each op_i that precedes $rb_i(sl)$ and up to the command SaveWork identified by the signal label sl, an undo(op_i) is executed. The order of execution of these undo commands is in the reverse of the order that the operations op_i appeared in the transaction. The effect of a *undo* operation is to remove the result of op_i from the system, as if op_i was never executed. As a result of these undo commands, the file store and the local program variables are restored to the state denoted by the savepoint sl which is the parameter of the RollBack command. Specifically, the undo of a read operation will release the lock on the data item, and the $undo(gs(sl))$ resets all the local program variables to the state denoted by sl.

The execution of a RollBack command in a transaction T_1 is shown in the following examples. The operations of T_1 till the RollBack command, are executed in the sequence shown below in Example 5.1.

Example 5.1 $r_1[y]\ sw_1(sl)\ r_1[x]\ w_1[z]\ gs_1(sl)\ rb_1(sl)$ □

The above transaction with the RollBack command expanded with the complete sequence of *undo* commands only is shown below.

Example 5.2 $r_1[y]\ sw_1(sl)\ r_1[x]\ w_1[z]\ gs_1(sl)\ undo(gs_1(sl))\ undo(w_1[z])\ undo(r_1[x])$ □

Once the state has been restored to the savepoint (i.e. all undo operations have been performed), the transaction is re-executed from the savepoint as shown in example 5.3 below. Here, this is reflected by the second $r_1[x]$ which is nothing but the re-read of data item x followed by $w_1[s]$. The reason why $w_1[s]$ is shown instead of $w_1[z]$ (i.e. re-writing of z) is to emphasize the fact that after the RollBack is executed, it is not necessarily true that all commands are re-done. This is specifically true if there is a conditional command after the step $r_1[x]$ which depend on the value of x. In such a case a different flow of control than the original one may result in a different list of commands being executed.

Also of interest is the $gs_1(nil)$ command in Example 5.3. If the execution of a GetSignal command in a transaction does not return a signal label, we say that no signal is *serviced* by the GetSignal and this is denoted by $gs(nil)$.

Example 5.3 $r_1[y]$ $sw_1(sl)$ $r_1[x]$ $w_1[z]$ $gs_1(sl)$ $undo(gs_1(sl))$ $undo(w_1[z])$ $undo(r_1[x])$ $r_1[x]$ $w_1[s]$ $r_1[u]$ rs_1 $r_1[p]$ $gs_1(nil)$ c_1 □

The interleaving of a set of transactions, when they execute concurrently, is modeled by a *history* [BHG87].

Definition 5.2 Two operations p and q are said to *conflict*, if they both operate on the same data item and at least one of them is a write operation.

Definition 5.3 A *complete history* H over a set of committed transactions $T = \{T_1, \ldots, T_n\}$ is a partial order with the ordering relation $<_H$ where:

1. $H = \cup_{i=1}^{n} T_i$;

2. $<_H \supseteq \cup_i^n <_i$; and

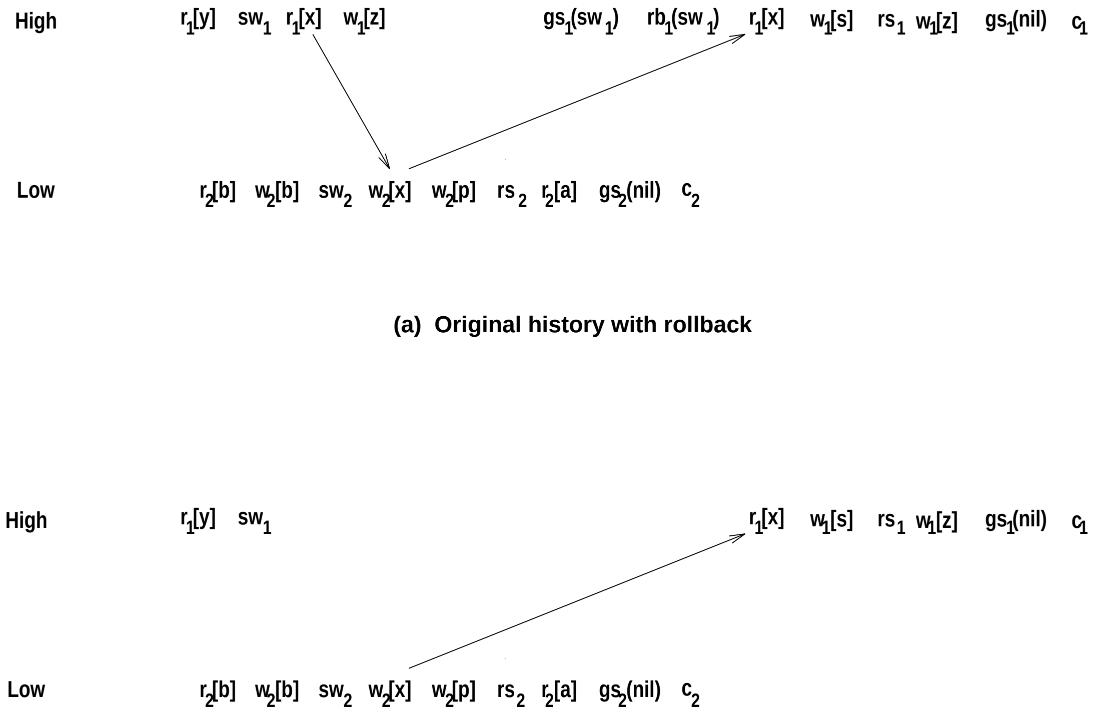
3. for any two conflicting operations p, $q \in H$, either $p <_H q$ or $q <_H p$.

Henceforth, by history we always mean a complete history. A serialization graph is a pictorial representation of a history and is defined as follows.

Definition 5.4 The *serialization graph* $SG(H)$ for a history H is a directed graph whose nodes are the transactions in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations.

Definition 5.5 A history H_s over transactions $T_1, T_2, \ldots, T_n$ is said to be a *serial history* if for every pair of transactions T_i, $T_j \in H_s$, either all the operations of T_i execute before any of the operations of T_j or vice versa.

In the transaction model proposed in this paper, a serial history H_s cannot contain a GetSignal which generates a rollback computation, since there is no interleaving of transactions. In other words, only $gs(nil)$ can be in H_s. It is not proper to define a serializable history H in this model, as being equivalent to some serial history H_s. This is because, the history H may contain some rollback computations, triggered by a GetSignal command, which will be absent in any H_s. Nonetheless, a rollback computation being formed of the dual $op_i, undo(op_i)$, in effect does not modify the file store. Thus, we are led to a slightly modified definition of serializability in terms of the $rollback - free\ projection$ of a history H. Informally, given a history H, the rollback-free projection of H is the history obtained from H by deleting all those operations that were undone due to a rollback command.


High
r1[y] sw1 r1[x] w1[z] gs1(sw1) rb1(sw1) r1[x] w1[s] rs1 w1[z] gs1(nil) c1
Low
r2[b] w2[b] sw2 w2[x] w2[p] rs2 r2[a] gs2(nil) c2
High
r1[y] sw1 r1[x] w1[s] rs1 w1[z] gs1(nil) c1
Low
r2[b] w2[b] sw2 w2[x] w2[p] rs2 r2[a] gs2(nil) c2

(a) Original history with rollback

(b) Rollback-free projection history

Figure 11: Rollback equivalent histories

Definition 5.6 The rollback-free projection, $R(H)$, of a history H is a restriction of the partial order $(H, <_H)$, such that

1. $R(H) \subseteq H$, that is, the operations in $R(H)$ belong also to the set of operations of the transactions in H;

2. if op_i, $sw_i(sl)$ and $rb_i(sl) \in H$ such that $sw_i(sl) <_H op_i \; <_H \; rb_i(sl)$, then both operations $rb_i(sl)$, $op_i \notin R(H)$;

3. any $op_i \in H$ not excluded by 2 above is included in $R(H)$;

4. for all op_i, $op_j \in R(H)$, $op_i <_{R(H)} op_j$ iff $op_i <_H op_j$.

Figure 11 shows a history H and its rollback-free projection history $R(H)$. We are now in a position to introduce the notion of a serializable history.

Definition 5.7 A history H is *serializable* if its rollback-free projection $R(H)$ is conflict equivalent to some serial history H_s.

Definition 5.8 A transaction T_i is said to be *well-formed* if

1. Transaction T_i acquires a lock in an appropriate mode on a data item x before performing an operation of that mode on x. Each lock operation of T_i is eventually followed by a corresponding release of the lock.

2. Transaction T_i, which is requesting a lock on a data item, has to wait if a conflicting lock has already been acquired on the same data item by another transaction.

3. If T_i has to rollback to a savepoint sl, it releases all the locks that it acquired after sl and up to its current point of execution and then reexecutes the code starting from sl, reacquiring all relevant locks as it proceeds.

4. Each GetSignal command in T_i is well-formed. That is, for every $gs_i(sl) \in T_i$ such that $sl \neq nil$, $gs_i(sl) <_i rb_i(sl)$.

5. In the execution of T_i, at least one GetSignal follows the RaiseSignal call. The RaiseSignal command should follow the last write lock operation in T_i and the GetSignal command should be after the last lock operation in T_i.

In order to satisfy the conditions 4 and 5 above, the code of T_i should be organized as follows: (1) For every SaveWork command of the form sl_i := SaveWork, which is defined before a GetSignal, there should be a handler in the GetSignal of the form $sl_i \rightarrow handler_i$; also, $handler_i$ should contain the command Rollback(sl_i), and (2) The control flow analysis of the transaction should ensure that at least one RaiseSignal after the last write and one GetSignal after the last lock operation is executed.

An example of a transaction that is well-formed is given in Figure 9. In contrast, the example of the transaction in Figure 10 is not well-formed, according to the definition above for two reasons: (1) The GetSignal on line 13 does not contain any Rollback command; and (2) The GetSignal on line 25 contains a conditional Rollback. Such constructs can be convenient in practical system programming, since in many applications it is useful to have a compromise between correctness and performance [GR93]. However, in order to ensure serializability the strong definition of well-formedness is required as in Definition 5.8. The simplest way to make the transaction in Figure 10 well-formed, is to replace all the GetSignals by the default invocation of GetSignal, viz. GetSignal[→ RollBack] thus satisfying condition 4 above.

In order to demonstrate that the relative order of the RaiseSignal and GetSignal, as stated in criterion 5 of well-formed transaction, is necessary to guarantee serializability, consider the example history in Figure 12. (In this figure, the dashed arrows indicate the temporal ordering of the operations, and not conflicts among them.) Transaction T_2 is not well-formed because the GetSignal command precedes the RaiseSignal command in T_2. Evidently this history which is rollback-free contains the cycle $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$, and hence is not serializable. However, if transaction T_2 were well-formed, the history would be serializable. Note that in this latter case the RaiseSignal of T_3 is serviced by the GetSignal of T_2, and the RaiseSignal of T_2 is serviced by the GetSignal of T_1.

It should also be noted that only the last RaiseSignal, GetSignal pair contributes towards the well-formedness of a transaction. Indeed, even if all the GetSignal calls other than the last one in a transaction services all the relevant signals, it can always happen that the last GetSignal misses at least one signal as in the non-serializable history of Figure 12. In

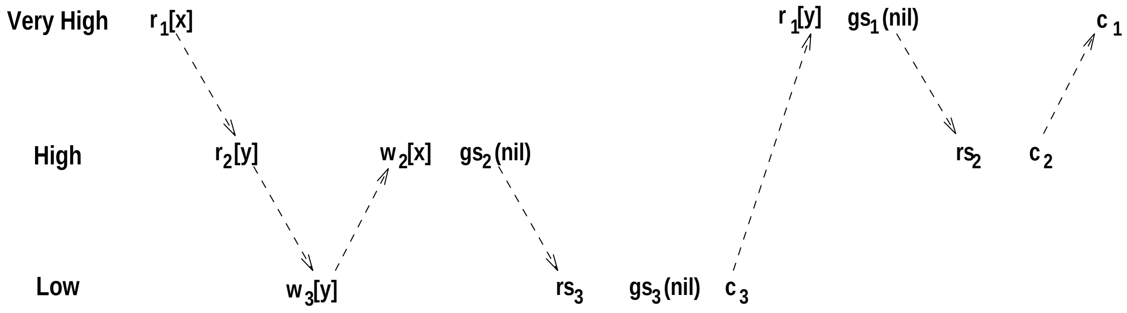


Figure 12: A history with a transaction that is not well-formed

subsequent discussions whenever we speak of rs_i, gs_i for a transaction T_i we mean the last RaiseSignal - GetSignal pair.

From properties 1, 4, and 5 of well-formed transactions, it follows that $T_i \rightarrow T_j$, where transaction T_i is at a level higher than T_j, only if T_i reads a low data item x which is subsequently written by T_j, and T_j raises the signal after the last GetSignal of T_i. If this is not the case then T_i is rolled back, the new value written by T_j, is re-read by T_i, and hence the conflict $T_i \rightarrow T_j$ is removed. We summarize this observation as a proposition as follows:

Proposition 5.1 Given two well-formed transactions T_i and T_j such that $L(T_j) \prec L(T_i)$, if $T_i \rightarrow T_j$ in $SG(R(H))$ then $gs_i <_{R(H)} rs_j$.

This proposition is generalized to include also transactions at any two security level by the following lemma.

Lemma 5.1 Given two well-formed transactions T_i and T_j, if $T_i \rightarrow T_j$ in $SG(R(H))$ then $gs_i <_{R(H)} gs_j$.

Proof: There may be three cases: T_i and T_j are at the same security level, T_i is at a lower level than T_j, and T_i is at a higher level than T_j. Note that if T_i and T_j are at incomparable security levels, they cannot have any conflict. Each of these cases is considered in turn:

1. Since transactions at the same security level use strict 2PL, at least the conflicting operation op_j in T_j must follow the commit of T_i. Thus, $gs_i <_{R(H)} c_i <_{R(H)} op_j <_{R(H)} gs_j$.

2. If T_i is at a lower level than T_j, again the reasoning in case 1 applies since the conflicting read operation of T_j is delayed until after the commit of T_i.

3. If T_i is at a higher level than T_j and $T_i \rightarrow T_j$ in the history, by proposition 5.1, $gs_i <_{R(H)} rs_j$. Thus, $gs_i <_{R(H)} rs_j <_{R(H)} gs_j$.

□

Theorem 5.1 A history H consisting solely of well-formed transactions is serializable.

Proof: By definition of serializable histories, the rollback-free projection $R(H)$ of the history H need be considered. We will show that the serialization graph for $R(H)$, $SG(R(H))$ does not contain any cycle.

Suppose $SG(R(H))$ contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, $n > 1$. By Lemma 5.1, it follows that $gs_1 < gs_2 < \ldots < gs_{n-1} < gs_n < gs_1$, which is a contradiction. □

6 Hierarchical Lock Protocol

So far the file store has been viewed as an unstructured collection of data items. However, it is useful to view the file store as a hierarchy of different granules of data items. Although the granularity of data items is unimportant for correctness, it is relevant for system performance because the concurrency among transactions can be increased by the use of locks on increasingly finer granules of data items. A small granule lock is costly for a complex transaction that accesses a large number of granules. A coarse granule lock on a data item, on the other hand, would discriminate against the many small transactions that want to access just a tiny part of the data item. Thus the choice of a lock granule presents a trade-off between concurrency and overhead. A hierarchical lock protocol is particularly useful in a multilevel secure file store since it can reduce the number of signals to higher-level transactions.

We will assume that the lock manager uses the natural hierarchical relationship among the different data granularities present in conventional systems, that is: file store, area, file, block (page) and record. Such a hierarchical relationship of data items is represented by a *lock type graph*. Each edge in this graph connects a data type of coarser granularity to one of finer granularity. A set of data items that is structured according to a lock type graph is called a *lock instance graph*.

As stated in Section 3.1, the file store is partitioned according to security levels. This results in a number of lock instance graphs, one for each security level, with a file store granule at the root of each lock instance graph. All the data items in a particular lock instance graph have the same security classification. This results in a single level file store for each security level. In contrast, if all the data items at different security levels were to belong to the same lock instance graph, then a pseudo data granule, coarser than the physically coarsest, would be required at the root of this graph. This pseudo data granule would have to be classified at the lowest security level in the security lattice. Otherwise, any low data item which is in some interior node, would be invisible to a low transaction. However, if the root is at the lowest classification level, then a lowest level transaction can put a write lock on the root and thereby prevent transactions at other security levels to modify those portions of the lock instance graph which is at their level. Moreover, the costs associated with maintaining one extra granule in the lock instance graph is high [GR93]. Thus, it is better to have single level file stores for each security level, for implementing a secure hierarchical locking protocol.

A lock on a coarse data x *explicitly* locks x and *implicitly* locks all descendants of x; for example, a read lock on a file implicitly read locks all the records in that file.

We define seven different kinds of locks. Of these, three are the read lock, write lock and signal lock described in section 3. The other four are generally termed as the *intention* locks. These intention locks allow the lock manager to ensure before locking a data object

x, that there are no locks on any ancestor of x that implicitly lock x in a conflicting mode. The different intention locks are described below.

Intent Read If a transaction T wants to read a data object x which is at the same security level as T, then it has to set *intent read locks* on all data items which are the ancestors of x in the lock instance graph, before setting the read lock on x.

Intent Write If a transaction T wants to write a data object x which is at the same security level as T, then it has to set an *intent write lock* on x's parent before it can set a write lock on x.

Intent Signal If a transaction T wants to read a data object x which is at a lower security level than T itself, then it has to set *intent signal locks* on all data items which are the ancestors of x in the data granularity hierarchy, before setting the signal lock on x.

Read Intent Write If a transaction T wants to read a data object x which is at T's security level and also wants to write to some part of x (i.e. to a finer granule of x) then T sets a *read intent write lock* on x. This is equivalent to setting a read lock on x and an intent write lock on x. The intent write lock on x ensures that write locks can subsequently be set on finer granules of x for writing.

The compatibility matrix for hierarchical locks is given in Figure 13.

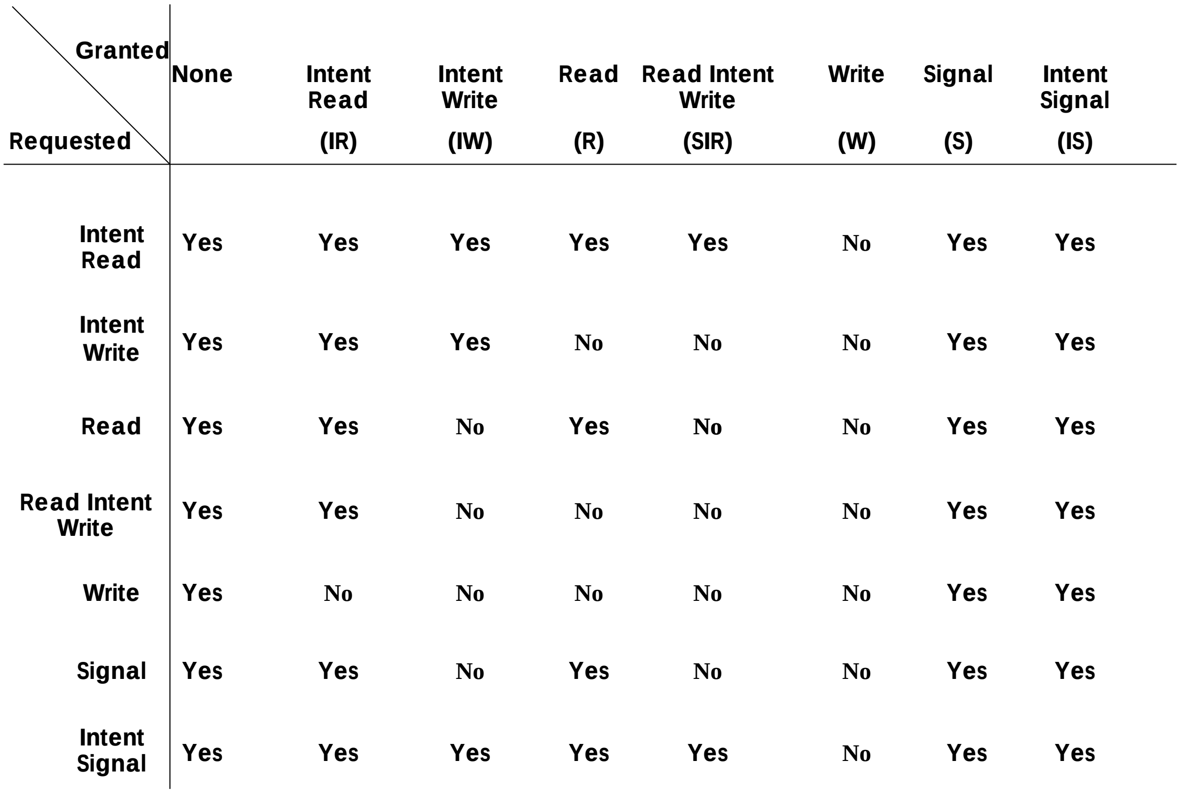
Apart from the different compatibility matrix used, the Trusted Lock Manager for hierarchical locking is very much similar to the standard Trusted Lock Manager discussed in section 4.2. The only difference is in the handling of RaiseSignal. In the Trusted Lock Manager module with hierarchical locking, if $x_1 \ldots x_n$ are the data items previously locked in Write mode, then for each T_j that has a Signal lock or an IntentSignal lock on x_k the TM of T_j has to be notified of a new value for x_k; if $y_1 \ldots y_n$ are the data items previously locked in IntentWrite or ReadIntentWrite mode by T_i, then for each T_j that has a Signal lock on y_k, the TM of T_j has to be notified of a new value for y_k. If T_j has a IntentSignal lock on y_k, then it is not required to be notified. Figure 14 shows the modified Trusted Lock Manager module with hierarchical locking. Only the portion for the RaiseSignal has been shown in the figure as the rest of the algorithm is the same.

Given a lock instance graph G that is a tree, the lock manager sets and releases locks for each transaction T_i according to the following hierarchical locking protocol.

1. If the data object x is not the root of G, and is at the same security level as T_i, then to set a read lock or an intent read lock on x, T_i must have an intent read lock or an intent write lock on x's parent.

2. If the data object x is not the root of G, but is at a security level lower than T_i, then to set a signal lock on x, T_i must have an intent signal lock on x's parent.

3. If x is not the root of G, and is at the same security level as T_i, then to set a write lock or an intent write lock on x, T_i must have an intent write lock on x's parent.



Granted Requested	None	Intent Read (IR)	Intent Write (IW)	Read (R)	Read Intent Write (SIR)	Write (W)	Signal (S)	Intent Signal (IS)
Intent Read	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Intent Write	Yes	Yes	Yes	No	No	No	Yes	Yes
Read	Yes	Yes	No	Yes	No	No	Yes	Yes
Read Intent Write	Yes	Yes	No	No	No	No	Yes	Yes
Write	Yes	No	No	No	No	No	Yes	Yes
Signal	Yes	Yes	No	Yes	No	No	Yes	Yes
Intent Signal	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes

Figure 13: Compatibility matrix for hierarchical locking

```
TrustedLockManager( )
repeat
receive (TM$_l$,T$_i$,op,x,mode);
    . . .
    . . .
case op do
    . . .
    . . .
    RaiseSignal:
        let $x_1 ... x_n$ be the data items locked in Write mode by T$_i$.
        for each T$_j$ that has either a Signal or an
        IntentSignal lock on $x_k$ do
        notify the TM of T$_j$ of a new value for $x_k$;
        let $y_1 ... y_n$ be the data items locked
        in IntentWrite or ReadIntentWrite mode by T$_i$.
        for each T$_j$ that has a Signal lock on $y_k$ do
        notify the TM of T$_j$ of a new value for $y_k$;
        send(TM$_l$,T$_i$,raiseOK);
endcase
forever
```

Figure 14: Trusted lock manager module with hierarchical locking

4. To read x, where x is at the same security level as T_i, the transaction T_i must own either a read lock or a write lock on some ancestor of x either explicitly or implicitly. (Note possessing a write lock on an object, enables a transaction to read it also.)

5. To write x, T_i must own a write lock on some ancestor of x either explicitly or implicitly.

6. To read a low level object x, the transaction T_i must have an explicit or implicit signal lock on some ancestor of x.

7. The transaction T_i may not release an intention lock on a data item x, if it is currently holding a lock on any child of x.

The goal of the hierarchical locking protocol given above is to ensure that transactions never hold conflicting (explicit or implicit) locks on the same data item.

Theorem 6.1 If two transactions obey the hierarchical locking protocol with respect to a given lock instance graph that is a tree, then they cannot own conflicting locks on the same node of the graph.

Proof: It is sufficient to show that two transactions cannot hold conflicting locks when either one or both of the locks are implicit. Moreover, only leaf nodes of the lock instance graph need be considered since if two transactions held conflicting locks on a nonleaf node x, they would be holding conflicting locks on all descendants (including the leaf) of x.

Two conflicting locks on leaf x, with either one or both being implicit in nature, can occur in the following twelve ways.

	Transaction T_i granted on x	Transaction T_j requested on x
1	implicit *read* lock	explicit *write* lock
2	implicit *read* lock	implicit *write* lock
3	explicit *read* lock	implicit *write* lock
4	implicit *write* lock	explicit *write* lock
5	implicit *write* lock	implicit *write* lock
6	explicit *write* lock	implicit *write* lock
7	implicit *write* lock	explicit *signal* lock
8	implicit *write* lock	implicit *signal* lock
9	explicit *write* lock	implicit *signal* lock
10	implicit *write* lock	explicit *read* lock
11	implicit *write* lock	implicit *read* lock
12	explicit *write* lock	implicit *read* lock

Case 1. If T_j has to have an explicit w lock on data item x, then from rule 3 of the locking protocol, it must own an intent write lock on x's parent. By induction, T_j must have intent write lock on all ancestor's of x, in particular on data item y, an ancestor of x. Now T_i can have an implicit read lock on x, if it owns a read lock on some ancestor of x. By rule 4 of the locking protocol and induction, T_i possesses a read lock on y. This is not possible since a read lock and an intent write lock conflict.

Case 2. By rules 4 and 5, T_i owns a read lock on some ancestor y of x and T_j owns a write lock on ancestor y' of x. There are three cases:

a. $y = y'$ which is not possible because this implies that T_i and T_j are holding conflicting locks on the same node.

b. y is an ancestor of y'. In this case T_j must own an intent write lock on y, which conflicts with a read lock on y as in Case 1.

c. y' is an ancestor of y. In this case T_i must own an intent read lock on y' which conflicts with write lock on y'.

Case 3. By rule 4 of the locking protocol, T_i has a read lock on some ancestor y of x. And by rule 3 and induction T_j must have an intent write lock on y. This is not possible according to the lock compatibility matrix.

Case 4. If T_i holds an implicit write lock on x then T_i holds a write lock on some ancestor y of x. In order for T_j to get an explicit lock on x, it must have an intent write lock on x's parent and by induction an intent lock on y. this is not possible according to the lock compatibility matrix.

Case 5. By rule 5 T_i owns a write lock on y, some ancestor of x and t_j owns a write lock on y', some other ancestor of x. There are three possible cases:

a. $y = y'$. This implies that both T_i and T_j have conflicting locks on the same data item and is obviously not possible.

b. y is an ancestor of y'. This is not possible because if T_j holds a write lock on y' it must have an intent write lock on y and thus a conflict with T_i.

c. y' is an ancestor of y. In this case T_i holds an intent write lock on y'; thus T_j conflicts with T_i.

Case 6. T_i holds a write lock on y, some ancestor of x and T_j has to obtain an intent write lock on that y in order to have an implicit lock on x. This is not possible.

Case 7. For T_i to have an implicit write lock on x, it must have a write lock on some ancestor y of X, on which T_j has to acquire an intent signal lock, if it has to own a signal lock on x. This is not allowed according to the lock compatibility matrix.

Case 8. If T_i has an implicit write lock on x, it must own a write lock on y, some ancestor of x. Similarly T_j has to own a signal lock on y', some other ancestor of x. As in Case 5 above, three different cases arises :

a. $y = y'$. This is clearly impossible.

b. y is an ancestor of y'. For T_j to hold a signal lock on y' it must have an intent signal lock on y which will then conflict with the write lock of T_i.

c. y' is an ancestor of y. In this case T_i must have an intent write lock on y', but then T_j's signal lock will conflict with this lock.

Case 9. In this case T_i holds an intent write lock on some ancestor y of x, on which T_j tries a signal lock. According to the lock compatibility matrix, these two locks conflict and hence this situation is not possible.

Case 10. The argument is similar to Case 7.

Case 11. Follows from a reasoning similar to Case 8.

Case 12. The same reasoning as in Case 9 applies.

□

Lemma 6.1 A transaction which satisfies conditions 3, 4 and 5 for well-formedness and obeys the hierarchical locking protocol for its lock and unlock operations is well-formed.

Proof Sketch: As shown in theorem 6.1 two transactions obeying the hierarchical locking protocol, cannot own conflicting locks simultaneously on the same node of the lock instance graph that is a tree. Thus transaction T_i, which is requesting a lock on an item, has to wait if a conflicting lock has been acquired on the same data item, by another transaction. Thus the transaction T_i satisfies condition 2 for well-formedness and is well-formed. □

Theorem 6.2 A history H consisting of well-formed transactions obeying the hierarchical protocol for locking and unlocking operations, is serializable.

Proof Sketch: The proof is similar to the proof for theorem 5.1 □

7 Conclusion and Future Work

This paper extended the two-phase locking mechanism into the realm of secure transaction processing using single-version data items. In the proposed scheme, the only way data sharing among transactions at different levels can take place is by having a higher level transaction T_j read a lower level data item x using a Signal lock on x. Since a lower level transaction T_i can never access higher level data item, T_i does not ever have to wait or be aborted owing to a conflicting higher level transaction T_j. This closes the timing covert channel that can arise owing to a lock based synchronization of concurrently executing multilevel transactions. In addition, deadlocks among transactions at multiple levels are also prevented. This is due to the fact that although, in this scheme, T_j may have to wait for T_i to release write locks, T_i never waits for T_j. This allows system deadlock detection by employing untrusted deadlock detectors at each security level. Each deadlock detector is responsible for possible deadlocks among transactions at its level.

The proposed GetSignal, RaiseSignal, SaveWork, and RollBack commands allow a simple generalization of the flat transaction model. The generalization is related to what are called sequential nested transactions in which transactions have an internal structure, though they are executed sequentially. In particular, transaction T starts a new subtransaction by executing a SaveWork call; sometimes T performs a Rollback to one of these savepoints aborting one of its subtransactions. The GetSignal corresponds to end subtransaction and, therefore, a commit operation, if nothing has to be notified. Indeed, in the sequential nested transaction model the commit of a subtransaction is essentially a null operation. The effects of the transaction are made durable and public only when the top level transaction commits.

In order to reduce the programming effort and make this approach more convenient for database applications, the proposed primitives can be invoked transparently in the application as follows. Every time a transaction requests a Signal lock, its transaction manager issues a SaveWork command. When the transaction completes and issues an End_Transaction command, the transaction manager invokes first the RaiseSignal command and then the GetSignal command. The GetSignal command invoked should be the default invocation, i.e., GetSignal[$\rightarrow$ RollBack] which, in case there is any signal to be serviced, will rollback the transaction to the savepoint immediately preceeding the earliest lower read operation that has to be redone. This transparent invocation of the system commands satisfies the conditions of well-formedness of transactions given in section 5 and hence guarantees serializable histories. Further, it appears that for transactions that do not read lower level data items, there is no need to include any GetSignal command. These transactions can be simplified by invoking only a RaiseSignal at the commit of the transactions.

The goal of this work is not exactly to improve performance. Rather it is to provide a secure concurrency control protocol that is based on locks on single-version data, and to provide flexibility to the system programmer. Indeed, our new system primitives allow the system programmer to emulate other secure concurrency control protocols like that of [Inf93a, Inf93b]. Nonetheless, it appears that the ideas presented here can be suitably modified to support higher levels of concurrency in the conventional transaction processing framework thus leading to improved performance. This is because an update transaction can acquire a write lock on a data item, even though a read lock is already set on the same data item. Note that the overhead due to possible rollbacks will be reduced for short-lived read transactions as they will tend to commit before getting signaled by the update transactions.

As part of our future work, we plan to investigate the impact of this novel concurrency control mechanism on different aspects of transaction processing. In particular, it appears that the problem of commit protocols for secure transactions can be solved more effectively within this scheme.

References

[AJ92] P. Ammann and S. Jajodia. A timestamp ordering algorithm for secure, single-version, multi-level databases. *Database Security, V: Status and Prospects, Carl E. Landwehr, ed., North-Holland, Amsterdam*, pages 23–25, 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, MA, 1987.

[BL76] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and multics interpretation,. Technical Report MTR-2997, The Mitre Corp., Burlington Road, Bedford, MA 01730, USA, March 1976.

[Cri82] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, June 1982.

[Den82] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1982.

[EMS91] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufman Publishers, Inc., San Mateo, California, 1991.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[IAT90] C. E. Irvine, T. B. Acheson, and M. F. Thompson. Building trust into a multilevel file system. In *Proceedings of the 13th. National Computer Security Conference*, Washington D.C., 1990.

[Inf93a] Informix Software, Inc., Menlo Park, CA. *Informix-OnLine/Secure Administrator's Guide*, April 1993.

[Inf93b] Informix Software, Inc., Menlo Park, CA. *Informix-OnLine/Secure Security Features User's Guide*, April 1993.

[Irv95] C. E. Irvine. A multilevel file system for high assurance. In *Proccedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.

[JA92] S. Jajodia and V. Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure database systems. In *Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California*, pages 216–224, May 1992.

[KTS90] T. F. Keefe, W. T. Tsai, and J. Srivastava. Multilevel secure database concurrency control. In *Proceedings of the IEEE 6th Int'l. Conference on Data Engineering, Los Angeles, California*, pages 337–344, February 1990.

[Lam73] B. W. Lampson. A note on the confinement problem. *Communication of the ACM*, 16(10):613–615, October 1973.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Ora92] Oracle Corp., Redwood City, CA. *Trusted Oracle Administrator's Guide*, 1992.

[RK79] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of ACM*, 22(5):115–123, May 1979.

[Sch74] M. Schaefer. Quasi-synchronization of readers and writers in a secure multi-level environment. Technical Report TM-5407/003, Syst. Develop. Corp., September 1974.

[SM94] S. K. Srivastava and D. L. McCue. Structuring fault-tolerant object systems for modularity in a distributed environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):421–432, 1994.

[SW88] M. A. Scaffer and G. Walsh. Lock/ix: On implementing unix on the lock tcb. In *Proceedings of the 11th. National Computer Security Conference*, pages 17–20, October 1988.