Database sharing—An efficient mechanism for supporting concurrent processes

by PAUL F. KING and ARTHUR J. COLLMEYER
Xerox Corporation
El Segundo, California

INTRODUCTION

The advent of transaction-oriented data processing systems has offered a number of new challenges to designers of database management systems. Requisites for efficient transaction processing include (1) a multiprogramming system oriented toward maximizing throughput subject to the response-time requirement of the interactive environment, and (2) an integrated database with centralized access control. An integrated database implies the elimination of redundant data processing. Such is necessary (though not sufficient) to achieve acceptable performance in transaction processing. The necessity of an efficient, responsive multiprogramming system is, of course, obvious. But efficiency in the transaction environment necessitates certain system provisions peculiar to the environment. One of these is the provision for the shared use of data. Time-sharing systems, while they generally provide for shared procedures, do not generally provide elaborate facilities for data sharing, since users typically do not require access to files other than their own. In the transaction environment, typified by a number of users operating on a single integrated database, elaborate provisions for database sharing are required.

In the sense that the use of a database is a privilege frequently extended to different users, shared databases are quite common. This kind of sharing is done primarily to avoid the proliferation of redundant data. But for the purposes of this paper, database sharing refers to the granting of simultaneous access to a database. This kind of sharing is motivated by the necessity for efficient, responsive multiprogramming in the transaction environment. By making a database available to several programs simultaneously, the mean number of programs eligible for execution increases, as does the mean occupancy of the disk queue, thereby increasing the potential for effective utilization of resources. The potential for improved performance depends, of course, on the manner in which the database is organized on the storage medium. Random (physical) organizations are more appropriate in transaction processing than the sequential organizations of batch processing systems. The potential for improved performance likewise depends on the extent to which the database can be shared.

The most sophisticated approach to database sharing admits concurrent WRITERS as well as READERS. This approach necessitates the most elaborate mechanism for regulating access to data within the database. Such mechanisms have been described for numerous applications.\textsuperscript{1-9} Indeed the CODASYL DBTG Report,\textsuperscript{10} in its proposed specification for a database management systems, details a KEEP-FREE mechanism to support concurrent update operations. However, the KEEP-FREE mechanism is not without shortcomings. The programmer using KEEP-FREE is, in essence, availing himself of a facility designed to inform him of any untoward interactions with other update programs. KEEP-FREE is not a mechanism for "locking" and "unlocking" data. While it avoids the "deadlock" problem, it does place a significant (perhaps excessive) burden on the application programmer insofar as the integrity of the database is concerned.

A LOCK-UNLOCK mechanism enabling the locking and unlocking of data for update purposes is a preferable solution. In general, however, a LOCK-UNLOCK mechanism is substantially more complex, as it introduces the potential for deadlock. Where databases are constructed solely on hierarchical structures it may be appropriate to insist that consecutive LOCK requests be separated by an UNLOCK request, so as to eliminate the possibility of deadlock. However, where network structures are exploited, such a limitation may be unreasonable.

In this paper, a LOCK-UNLOCK Mechanism enabling the incremental allocation of data elements to processes is described, along with an efficient method for detecting potential deadlocks. The mechanism prevents inter-process interference and permits simple automatic recovery from deadlocked processes as well.

LOCK-UNLOCK

An environment not unlike that of transaction processing is assumed. A number of programs, WRITERS as well as READERS, operate concurrently on a single database. In this mode of operation, READERS are assumed impervious to the effects of concurrent WRITERS. In other words, READERS are fully aware of the possibility that the database may be altered during the course of their task and that information extracted from the database may be out-of-date by the time the task is completed. If such is unacceptable, an inquiry program may use the LOCK-UN-
LOCK facility, in which case its behavior is likened to that of a WRITER who locks records one at a time until he has acquired exclusive control over all records affected by the intended update. When the update has been accomplished, the records are unlocked via the UNLOCK Function. For the sake of simplicity, we will classify a user program as either (1) a READER, impervious to changes in data, or (2) a WRITER, who allocates records for exclusive use via the LOCK Function. It is users in the second category who make database sharing a difficult proposition.

WRITERS, using the LOCK-UNLOCK Mechanism, lock records one at a time until all the records involved in the intended operation have been acquired. Hence, an operational characteristic peculiar to a WRITER is that some of the records allocated for exclusive use via the LOCK Function. For the purposes of this paper this set is known as the lock list. Taken as a set, the lock lists of all the WRITERS currently active in the system constitute a listing of all the records which cannot be allocated for exclusive use. If a WRITER happens to request one of these records for his exclusive use, he must be placed in a queue for said record. Under these circumstances, the WRITER is said to be blocked. The blocked WRITER cannot proceed until the record in question has been unlocked.

Two (or more) WRITERS incrementally allocating records to themselves pose the problem of deadlock. A blocked WRITER is deadlocked when there is no way through normal operations for him to become not blocked. To illustrate, suppose WRITER 1 has locked record $j$ and wants record $k$, and WRITER 2 has locked record $k$ and wants record $j$. Neither can proceed; the update of WRITER 1, as well as that of WRITER 2 is deadlocked. It is necessary to "backtrack" one of the stymied programs under system control and then restart it.

The detection of incipient deadlocks can be a more or less complex operation, depending on the detection algorithm. The computational overhead of a mechanism to solve for necessary and sufficient conditions has to date been considered intolerable. Simpler mechanisms based on necessary (but not sufficient) conditions have been proposed, at the cost of increased detection frequency. In this paper, a simple algorithm is proposed for deadlock detection based on necessary and sufficient conditions.

THE MODEL

The algorithm that has been developed for deadlock detection is best presented by means of a graphical model of database access. To introduce the model, some graph-theoretic definitions are first required.

A directed graph is a pair $<N, A>$, where $N$ is an abstract set and $A \subseteq N \times N$. Each element in $N$ is called a node and each pair $(a, b)$ in $A$ is termed an arc. The arc $(a, b)$ is said to be directed from node $a$. A path is a sequence of two or more nodes $(n_1, n_2, \ldots, n_m)$ with each connected to the next by an arc. That is, for each $n_i, 1 \leq i \leq m-1, (n_i, n_{i+1}) \in A$. A path is a loop if its first and last nodes are the same.

From the discussion in the previous section, it should be clear that the state of all accesses with respect to a given database can be defined by describing:

1. The allocatable data elements (e.g., records) in the database,
2. The active processes (WRITERS), and
3. The lock list associated with each process.

Therefore, the state of all accesses of a database can be defined by a database access state graph $<P \cup E, L>$. The set of nodes within each of these graphs consists of the union of the set of active processes, $P$, and the set of allocatable data elements, $E$. Each lock list is represented by a path beginning at an active process and connecting it with each data element allocated to that process. Thus, the set of arcs within the lock lists comprises the set $L$.

More formally, a database access state graph is a directed graph $<P \cup E, L>$ where the set of process elements

$$P = \{p_i \mid p_i \text{ is the } i\text{-th oldest process} \}$$
$$E = \{e \mid e \text{ is an allocatable data element} \}$$
$$L = A \cup B$$

The set of lock lists, $L$, is composed of the set of allocated elements,

$$A = \{ (a, b) \mid a = p_i \text{ and } b \text{ is the } j\text{-th oldest data element allocated to } p_i \}$$
$$A = \{ (e_{i-1}, e_{ij}) \in A \}$$

and the set of blocked allocation requests,

$$B = \{ (a, b) \mid a = p_i \text{ or } a = e_{i-1} \text{ and } (e_{i-1}, e_{ij}) \in A \text{ with process } p_i \text{ being blocked when requesting allocation of data element } b = e_{ij} \} \in A$$

Since each access state of a database is represented by an access state graph, operation of the LOCK-UNLOCK Mechanism can be modeled by state transition functions that map access state graphs to access state graphs. The four required functions are:

1. The LOCK Function. If database access state $s = <P \cup E, L>$ then LOCK(s) = $s' = <P' \cup E, L'>$. If $P = \{p_i \mid p_i \text{ is a process, } 1 \leq i \leq n \}$ then $p_i' = p_i \cup \{p_{n+1}\}$. That is, the LOCK Function adds a process node to the graph.
2. The UNLOCK Function. This function is the inverse of the LOCK Function, and its application deletes an isolated process node from a graph.
3. The ALLOCATE Function. If database access state $s = <P \cup E, L>$, then ALLOCATE $(a, p_i, e_i)$ = $s'$. If $L = A \cup B$ then $s' = <P' \cup E, L'>$ and $L' = A \cup B \cup \{(p_i, e_i) \text{ or } (e_{i-1}, e_{ij})\}$. This function adds an arc to
the graph and thereby models the allocation of a data element to a process.

(4) The DEALLOCATE Function. This function is the inverse of the ALLOCATE Function. Its application deletes an arc from the graph and thus represents the release of a data element from a lock list.

Figure 1 illustrates the application of the LOCK and UNLOCK Functions to a simple database access state. In the first graph shown, \( P = \{p_1, p_2, p_3\} \), \( E = \{d_1, d_2, \ldots, d_6\} \), and \( L = \) the set of paths, \( \{(p_1, d_1, d_6, d_5), (p_1, d_1, d_2), (p_2, d_4)\} \). The arc \((p_1, d_1) \in B\) indicates \( p_1\) is blocked, while all other arcs are elements of \( A\). The figure shows the LOCK Function adding process node \( p_4\), and the UNLOCK Function is shown deleting it. Figure 2 gives an example of the application of the ALLOCATE and DEALLOCATE Functions.

In terms of the model, the normal access sequence for a given process consists first of an application of the LOCK Function. This is followed by some number of applications of the ALLOCATE and DEALLOCATE Functions. At some point, the number of applications of ALLOCATE is equaled by applications of DEALLOCATE, and the access sequence ends with an UNLOCK.

DEADLOCK DETECTION

From the discussion above, it should be obvious that the ALLOCATE Function is the only function defined that can precipitate a deadlock. This is clearly the case, for ALLOCATE is the only function capable of blocking a process.

It is now possible to describe a simple and efficient deadlock detection algorithm in terms of the model just presented. The following theorem provides the theoretical basis for the detection procedure.

**THEOREM:** If a valid database access state \( s \) is not a deadlocked state, then ALLOCATE \((s', p_i, e)\) = \( s'\) is a deadlocked state if and only if

1. process \( p_i\) is blocked on attempting to allocate data element \( e\), and
2. the database access state graph representing \( s'\) contains a loop.

**PROOF:** To establish these conditions as necessary for \( s'\) to be a deadlocked state, notice first that if \( p_i\) is not blocked then, by definition, \( s'\) is not a deadlocked state. Now, let \( <P \cup E, L'>\) be the database access state graph representing \( s\) with \( P = \{p_i \mid 1 \leq i \leq n\} \) and \( n \geq 2\). Assume ALLOCATE \((s, p_i, e)\) = \( s'\) is a deadlocked state and that \( s' = <P \cup E, L'>\) does not contain a loop.

Since \( s'\) is a deadlocked state, in \( s'\) there is a set of \( P_d\) of \( m\) deadlocked processes, where \( m \leq 2\), and \( P_d = \{p_{d1}, p_{d2}, \ldots, p_{dm}\} \subseteq P\). By definition, each \( p_{di}\) is blocked. Furthermore, each \( p_{di}\) is blocked by a \( p_{dj}\) in \( P_d\), with \( i \neq j\). If \( p_{di}\) were not blocked by some \( p \in P_d\), then \( p_{di}\) would have to be blocked by a nondeadlocked process; therefore, \( p_{di}\) would not be deadlocked. Thus, if there are \( m\) processes in \( P_d\), then \( p_{d1}\) is blocked by one of the \((m-1)\) processes \( \{p_{d2}, p_{d3}, \ldots, p_{dm}\} \).

Assume for convenience that \( p_{d2}\) blocks \( p_{d1}\). Now, \( p_{d2}\) must in turn be blocked by one of the \((m-2)\) processes \( \{p_{d3}, p_{d4}, \ldots, p_{dm}\} \). If not, then \( p_{d1}\) would have to be blocked by \( p_{d2}\). If \( p_{d2}\) were blocked by \( p_{d3}\), then for some data element \( c'\) the path \((p_{d2}, \ldots, c', c) \in A\) and the arc \((c, c') \in B\) while the path \((p_{d1}, \ldots, c') \in A\) and the arc \((b, b') \in B\) while the path \((p_{d2}, \ldots, b', \ldots, c') \in A\). This, however, violates the assumption that no loop is contained in \( <P \cup E, L'>\), since the path \((b', \ldots, c', c', \ldots, b, b')\) is a loop (see Figure 3). Hence, \( p_{d1}\) must be blocked by one of the processes \( \{p_{d2}, p_{d3}, \ldots, p_{dm}\} \).

![Figure 1-The LOCK and LOCK functions](image1.png)

![Figure 2-The ALLOCATE and DEALLOCATE functions](image2.png)

![Figure 3-A deadlocked state involving processes \( p_{d1}\) and \( p_{d2}\)](image3.png)
More generally, note that if a sequence of \(k\) processes \(p_1, p_2, \ldots, p_k\) exists such that each \(p_{i-1}\) is blocked by \(p_i\) for all \(2 \leq i \leq k\), then a path exists from \(p_1\) through elements of each \(p_i\)'s lock list to the last element in the lock list of \(p_k\). The lock list of each \(p_{i-1}\) is connected to the lock list of \(p_i\) by the arc representing a blocked request, which is always the last arc in a lock list.

Now consider the \(j\)-th process of the \(m\) deadlocked processes of state \(s'\). Assume for convenience that for \(2 \leq i \leq j\), \(p_{i-1}\) is blocked by \(p_i\). Then, \(p_i\) must be blocked by one of the \((m-j)\) processes \(\{p_{j+1}, p_{j+2}, \ldots, p_m\}\). For each \(p_i\) were blocked on allocating \(b'\) by some \(p_i\) with \(1 \leq j \leq m\), then the path \(\{p_{j+1}, \ldots, b', \ldots, b\}\) exists and the arc \((b', b')\) from the lock list of \(p_{j+1}\) to that of \(p_i\) creates the loop \((b', \ldots, b, b')\)—contrary to the assumption that no loop exists. Hence, \(p_i\) must be blocked by one of the processes \(\{p_{j+1}, p_{j+2}, \ldots, p_m\}\).

However, for \(j = m\) the above implies that \((m-m)\) or zero choices remain for the process that blocks \(p_m\). Therefore, it was false to assume that if \(s'\) is a deadlocked state, then a loop did not exist in \(G_{F, E, L'}\).

To establish the sufficiency of the conditions in the theorem, suppose that the blocking of \(p_1\) creates a loop in the access state graph. Since a single lock list cannot contain a deadlocked state of arbitrary complexity are easily and efficiently detected. A deadlock detection procedure that is both simple and efficient. Since a deadlock can occur only when an allocation request results in a process being blocked, which is assumed to be an infrequent event in a transaction processing environment, only infrequently will an examination of the database access state graph be necessary. In those instances when a process is blocked and it becomes necessary to test for a loop in the access state graph, the computation required for the test is nearly trivial since (1) the data element requested by the blocked process must be in the loop, and (2) the out-degree of every node in a database access state graph is one. Thus, deadlocked states of arbitrary complexity are easily and efficiently detected.

This detection method has another useful characteristic—it directly identifies those processes that are responsible for the deadlock. The processes that are responsible are, of course, those which are blocking each other. In general, however, it is possible to encounter a deadlocked access state in which an arbitrary number of processes participate, but only a small fraction of these are responsible for that state. This condition can exist since any number of processes can themselves be blocked while either not blocking other processes or not blocking others in a deadlocked manner (i.e., processes participating in the deadlock whose lock lists can be removed from the database access state graph without removing the deadlock condition). However, it is obviously those processes whose lock lists participate in the loop that cause a deadlock condition to exist. By detecting the existence of a loop, the algorithm has also isolated the processes responsible for the deadlock, and, thus, the method has also accomplished the first essential step in the recovery process.

**RECOVERY**

In the context of shared databases, Recovery is the procedure by which the effects of a (necessarily) aborted process on the object database are reversed so that the process can be restarted. On the detection of a deadlock a Recovery Procedure must be invoked. The first element of the Recovery Procedure is the determination of which process to abort. This decision might be based on any of several criteria. For example, it might be advisable to abort the most recent of the offending processes, or the process with the fewest allocated data elements. In any event, the information required for this decision is readily available (see above). This decision is, however, a small part of the recovery problem. To recover efficiently, the LOCK-UNLOCK Mechanism requires features beyond an efficient detection algorithm. One such feature is a checkpoint facility—a facility that records the state of a process and thus enables it to be restarted. Clearly, a checkpoint must be performed at each LOCK. Furthermore, to enable efficient restoration of the database, utilization of a process map is appropriate. A process map is basically a map from a virtual addressing space to a real addressing space, maintained for each process in the database access state graph. Database management systems are typically characterized by three levels of addressing: content addressing, logical addressing, and physical addressing. Associative references are processed in two steps: a content-to-logical address transformation, followed by a logical-to-physical address transformation. This "virtual" secondary storage characteristic by a logical-to-physical storage map provides device independence and facilitates the efficient use of storage hierarchies. The process map is similarly a logical-to-physical storage map. A process map is created and associated with process at the execution of a LOCK Function. With each execution of an ALLOCATE Function, a (physical) copy of the allocated data element is created and an entry is made in the associated process map. Subsequent references by the process to the database are routed through its process map; hence, incremental updates are performed on the copies of the data elements. The DEALLOCATE Function effects a modification of the database storage map and the deletion of the associated entry in the process map. ALLOCATE therefore has the effect of creating a physical copy of the object data element accessible only to the allocator, and DEALLOCATE has the effect of making the physical copy available to all processes and the

---

From the collection of the Computer History Museum (www.computerhistory.org)
A process map makes the recovery from a deadlocked access state a relatively simple matter. Once a decision is reached as to which process to abort, that process is merely restarted at the checkpoint performed by the LOCK Function. Implicit in this action, of course, the restoration of the lock list of that process to an empty state. That is, each data element that was allocated to the process is released, and the copies of these elements are discarded. Clearly, priorities must be appropriately arranged to insure that a process blocked by the aborted process is allocated the released data element for which it was previously blocked.

No further action by the Recovery Procedure is required, for due to the process maps, the actual database was unaltered by the aborted process. Note further that the utilization of process maps significantly reduces the probability of WRITERS interfering with READERS, since references to data elements by READERS are always directed to the actual database.

REFERENCES