Technical Report

RESEARCH IN FUNCTIONALLY DISTRIBUTED COMPUTER SYSTEMS DEVELOPMENT

Kansas State University

Virgil Wallentine
Principal Investigator

Approved for public release; distribution unlimited

VOLUME VIII

A DEADLOCK PREVENTION ALGORITHM FOR DISTRIBUTED DATA BASE MANAGEMENT SYSTEM

U.S. ARMY COMPUTER SYSTEMS COMMAND FT BELVOIR, VA 22060

01 8 9 075
**Title:** A DEADLOCK PREVENTION ALGORITHM FOR DISTRIBUTED DATA BASE MANAGEMENT SYSTEMS.

**Author(s):** Fred Maryanski

**Performing Organization:**
- Kansas State University
  - Department of Computer Science
  - Manhattan, KS 66506

**Monitoring Agency:**
- US Army Computer Systems Command
- Attn: CSCS-AT
- Ft. Belvoir, VA 22060

**Report Date:** February 1977

**Security Class.:** Unclassified

**Distribution Statement:** Approved for public release; distribution unlimited.

**Supplementary Notes:** The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

**Keywords:** Deadlock

**Abstract:**

---

**DD FORM 1473**

**UNCLASSIFIED**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)
-ABSTRACT-

The problem of deadlock in distributed data base management is analyzed in terms of performance effects of potential deadlock handling schemes. The performance tradeoffs of deadlock detection and deadlock prevention for distributed data base management systems are compared. Since the run-time overhead in deadlock prevention is projected to be less than for deadlock detection, an algorithm for preventing deadlocks in distributed data base systems is developed. The critical information for the deadlock prevention algorithm is maintained in a shared record list. The shared record list contains all shared access records for a set of tasks. Shared record lists are maintained dynamically by the run-time system. A proof that the algorithm prevents deadlocks in a distributed data base management system is provided along with a comprehensive example.
A Deadlock Prevention Algorithm
for Distributed Data Base Management Systems

Technical Report CS77-02

Fred J. Maryanski

Computer Science Department
Kansas State University
Manhattan, Kansas 66506

February 1977

1 The work reported herein was supported by the United States Army Computer Systems Command, Grant No. DAAG 29-76-C-0108.
Abstract

The problem of deadlock in distributed data base management is analyzed in terms of performance effects of potential deadlock handling schemes. The performance tradeoffs of deadlock detection and deadlock prevention for distributed data base management systems are compared. Since the run-time overhead in deadlock prevention is projected to be less than for deadlock detection, an algorithm for preventing deadlocks in distributed data base systems is developed. The critical information for the deadlock prevention algorithm is maintained in a shared record list. The shared record list contains all shared access records for a set of tasks. Shared records lists are maintained dynamically by the run-time system. A proof that the algorithm prevents deadlocks in a distributed data base management system is provided along with a comprehensive example. A discussion of the efficiency of the deadlock prevention algorithm indicates that partitioning the data base into sub-schemas reduces the overhead.
Key Words and Phrases:

* Distributed database management systems, deadlock prevention, system deadlocks. *

CR Categories:

4.33
1 Introduction

One of the major trends in computer systems is toward the decentralization of resources and facilities. This phenomenon has lead to a requirement for distributed data base management systems. A distributed DBMS permits an application program executing on a processor in a computer network to access data on any other node in the network. In an optimal situation the only limits on data access are communication linkages and security. A distributed DBMS relies upon its underlying network for communication facilities. The basic data base software in a distributed system is functionally identical to a centralized (one machine) system.

From the preceding statements it appears that a distributed DBMS can be realized by interfacing a centralized data base system with a network communication facility. However, as indicated by Fry and Sibley [10], when a data base system is extended to operate over several machines a whole new class of problems arises while many existing problems become more complex. Among those problems that are complicated by the distribution of the data base is that of deadlock. This paper proposes a method for preventing deadlock in a distributed data base system. The algorithm stated here is intended to avoid deadlock in a manner that is transparent to the application program and with minimal effect on processes that would not be involved in a deadlock situation.

2. Distributed Data Bases Systems

Before the presentation of the deadlock prevention algorithm can occur, terminology concerning data base management systems in general and distributed data base systems in particular must be established. A schema is a logical description of a data base. The portion of a schema that may be accessed by a particular application program is specified by the sub-schema associated
with that program. The sub-schema defines the data records that the program may operate upon and indicates the logical relationships among the records in the data base.

In a centralized data base system, the application program, sub-schema and data base software all reside on one processor which is physically connected to the secondary storage containing the data base. A distributed data base management system has resources and their control spread among the processors of a computer network. In a distributed DBMS, an application program executing on one network node may access data that reside at several distinct nodes. A computer that executes data base application programs is a host machine. A back-end machine is one which controls access to data. A machine with both capabilities is termed a bi-functional machine. Figure 1 portrays a distributed DBMS with host, back-end, and bi-functional nodes. A discussion of the general organization of distributed data bases can be found in Reference [18].

In order to implement a distributed DBMS, the software required for a centralized DBMS is necessary plus some communication and control software. As indicated in Reference [18], communication between application programs and data base tasks can best be accomplished by means of a generalized message system which is capable of handling communication among heterogeneous machines. The message system must also allow for transmission over a wide range of inter-machine connections, from conventional phone lines (typically 1200 baud) to shared memory (approximately 10M baud).

A group of machines tied together via memory-to-memory linkages is known as a cluster. A network can be formed from a collection of clusters joined by lower speed connections. In a distributed DBMS communication between machines in the same cluster can occur with little overhead. However, intercluster communication may result in noticeable performance degradation.
The impact of intermachine communication upon system performance is an important consideration in the treatment of deadlock in a distributed DBMS.

3. Deadlock in a distributed DBMS

In general, deadlock occurs when two or more processes request a set of shared resources in a sequence that results in the activation of each process being dependent upon the acquisition of a resource held by another process whose activation is also blocked. A considerable amount of study has been done on the deadlock problem of operating systems [5,6,9,12-14,19-20]. The particular form of deadlock under consideration in this report involves processes that are data base application tasks residing on (potentially) different machines in a data base network. The shared resource is a collection of records dispersed among the data bases of the system that may be updated by more than one application program. It is assumed that no user imposed restrictions concerning accessibility have been placed upon the data records. This form of deadlock is a DBMS problem, not a user problem. The means of treating deadlock must be totally transparent to the application program.

The special difficulty of deadlock in a distributed data base is that since there is no central control point in the network, the responsibility for noting the actual or potential occurrence of a deadlock situation can not easily be assigned. The task requesting the resource can be informed that the request was unable to be fulfilled. However, if that resource is controlled by one (or more) different processors, considerable manipulation and intermachine communication would be required to determine if a deadlock situation exists.
4. Deadlock Detection and Prevention

There are two basic mechanisms for treating the deadlock problem [4]. One approach is to prevent deadlock before it can occur. Deadlock prevention requires a prior knowledge of the shared records to be operated upon by all active application tasks in the system. The alternative to deadlock prevention is deadlock detection which involves noting the existence of a deadlock situation, and then resolving the dilemma. Generally, a deadlock situation can be resolved by halting one of the competing processes and freeing its resources for access by other processes. However, task "rollback" is detrimental to system performance and in some cases infeasible.

In a distributed DBMS, both deadlock prevention and detection produce considerable overhead; particularly if intercluster communication results. Deadlock prevention in a distributed data base requires that records that may be shared among several tasks (and updated by at least one) be identified. For this information to be meaningful, only the records shared with currently active tasks should be included. This implies that whenever a task that updates shared records enters the system the list of shared records must be revised. When this has taken place, the new task can proceed. Whenever a need to access a shared record arises, a prevention algorithm can be invoked to determine if the access may proceed. If not, the task is blocked until the resources are available.

The main cause of overhead in deadlock prevention in a distributed DBMS is the creation and communication of the list of shared records for active tasks. This is a continual operation which must occur whenever tasks are created and destroyed. As in all prevention schemes, some time is devoted to avoiding deadlocks which would not have occurred during a particular execution since a task may have access rights to many more records than it actually uses.
Deadlock detection schemes represent an a posteriori approach to the problem of avoiding deadlocks. In a distributed data base system, deadlock detection involves first identifying a set of two or more tasks blocking each other from a collection of shared records. Generally a "timeout" mechanism which involves noting that the effected tasks have been waiting for longer than some fixed time is used in deadlock detection. Once the set of deadlocked tasks and the conflicting resources are identified, one of the tasks must be rolled back to some point that will free the resources necessary to break the deadlock. Rollback involves restoring all data to the values held before the operations being retracted were performed. In a distributed environment, the data base operations may be initiated on one host processor and carried out by several different back-end processors. Rollback action would have to be initiated by the host processor and then carried out by the back-end processor in a manner analogous to the execution of a standard data base access. This will necessitate considerable message transmission activity to start and synchronize the rollback operation.

An additional negative performance factor in the deadlock detection approach is that tasks not involved in the deadlock situation can be effected if they have accessed data written by the sequence of commands being rolled back. In this case, the task accessing the data would also have to be rolled back. It is possible for the rollback to cascade throughout the system in the worst case.

The deadlock detection algorithms described in references [2,3,7,15] all require a dynamic list of processes and the records that they access. This information is similar to that required in a deadlock prevention scheme. Acquiring and maintaining an accessibility list for deadlock detection could require a heavy communication load in a distributed system as in the case of deadlock prevention.
A comparison of the two alternatives for avoiding deadlock indicates that both types of algorithms would require some form of a record accessibility list. Since deadlock prevention requires continual computational and communication activity to avoid deadlock whereas deadlock detection measures are most likely to be invoked infrequently, there is potentially more overhead in deadlock prevention. However, the overhead is fixed and the prevention algorithm has no effect upon processes that may not be involved in any deadlock situations. In a distributed data base system, the rollback and timeout mechanisms of deadlock detection could result in substantial computation and communication overhead and also the rolling back or blocking of tasks not involved in the deadlock situation.

Because of the uncertain and potentially serious performance degradation that may result from rollback in a distributed DBMS deadlock prevention is a safer strategy for handling deadlocks in a distributed data base environment. Thus, this paper concentrates upon deadlock prevention.

4. Information Required to Prevent Deadlock

In a distributed DBMS utilizing the proposed prevention algorithm, each back-end processor will be responsible for preventing deadlock situations involving the portion of the data base under its control. Since the back-end processor performs the function of manipulating the data, it is best suited to assume the responsibility for deadlock prevention. In this way, the prevention of deadlock is removed from the application task.

For each active task that it is serving, the back-end processor maintains a list of records that may be accessed by several tasks and updated by at least one task.

The shared record list is derived from the subschema of task. When a task is initiated its shared record list is circulated among the back-end
processors to determine if any interaction with other task exists. A list of interacting tasks is maintained by the back-end processors. In order to minimize the communications overhead upon task initiation, each back-end processor can maintain a list which indicates those back-end processors which may control records shared with any given sub-schema. Only these back-ends with potential interaction need be contacted. Upon task termination similar action must be taken to withdraw the task and its records from the task interaction and shared record lists. The shared record list is conceptually similar to the process set of Chu and Ohlmacher [4].

6. Deadlock Prevention

An algorithm for the prevention of deadlock in a distributed DBMS is developed in this section. Initially some notational conventions must be established.
Definition 1. (Notation)

1. \( r_j \) - a record in the data base
2. \( T_K \) - an application task.
3. \( R_K \) - potential shared record list of \( T_K \). A set of shared records is accessed by several tasks and updated by at least one.
4. \( X_T \) - a task interaction list. A set of tasks whose potential shared record lists have non-empty pairwise intersections.
5. \( S_T \) - the shared record list of \( X_T \). All records appearing in more than one potential shared record list of the tasks in \( X_T \).
6. \( B_K \) - the back-end processor executing a data base request for \( T_K \).
7. \( S_{T,K} \) - the shared record list of a set of tasks \( T \) on back-end processor \( K \). A record in a shared record list is marked with a task identifier when it is requested or locked.
8. \( \text{m}(S_{T,K}) \) - the number of distinct tasks that have records marked in \( S_{T,K} \).
9. \( \text{l}(S_T) \) - the number of distinct tasks that have records locked in \( S_T \).

For a given task interaction list \( X_T \), a copy of \( S_T \), the shared record list, is maintained on each back-end processor executing data base operations for a task in \( X_T \).
In order to properly prevent deadlock, the state of the system immediately prior to a deadlock state must be described and recognized. If an algorithm can develop which insures that the distributed DBMS will never enter a state that can immediately lead to deadlock, then the algorithm will prevent deadlock. First let us formally define deadlock.

Definition 2
A set of tasks \( T = (T_1, T_2, \ldots, T_m), m \geq 2, \) is deadlocked if for
1. \( i \cdot m-1, T_i \) is blocked by \( T_{i+1} \) and \( T_m \) is blocked by \( T_1. \)

Example 1
Assume there are five tasks \( T_1, T_2, T_3, T_4, T_5, \) active in the system.

Let
\[
R_1 = \{r_1, r_2, r_3\} \\
R_2 = \{r_1, r_4, r_7\} \\
R_3 = \{r_3, r_5, r_7\} \\
R_4 = \{r_6, r_9\} \\
R_5 = \{r_8, r_9\}
\]

Then
\[
X_1 = (T_1, T_2) \\
X_2 = (T_3, T_4) \\
X_3 = (r_2, r_3) \\
X_4 = (T_1, T_2, T_3) \\
X_5 = (T_4, T_5)
\]

and
\[
S_1 = \{r_1\} \\
S_2 = \{r_3\} \\
S_3 = \{r_7\} \\
S_4 = \{r_1, r_2, r_7\} \\
S_5 = \{r_9\}
\]
If $T_1$ is blocked by $T_2$, $T_2$ is blocked by $T_3$, and $T_3$ is blocked by $T_1$, then $X_4$ is deadlocked.

**Definition 2**

A set of tasks $T = \{ T_1, T_2, \ldots, T_m \}$, $m \geq 2$, is in a deadlock-prone state if there is a sequence of unfulfillable requests that can be issued by the tasks in $T$ that will place $T$ in a deadlocked state.

**Lemma 1**

A set of tasks cannot enter a deadlock state without first entering a deadlock-prone state.

This result follows immediately from Definitions 2 and 3.

**Example 2**

Consider the set of tasks in the previous example. Assume that

- $T_1$ has locked $r_3$;
- $T_2$ has locked $r_1$;
- and $T_3$ has locked $r_7$.

This is in a deadlock-prone state since the following sequence of commands results in a deadlock state:

- $T_1$ requests $r_1$
- $T_2$ requests $r_7$
- $T_3$ requests $r_3$.

Since $T_1$ will be blocked by $T_2$, $T_2$ will be blocked by $T_3$, and $T_3$ will be blocked by $T_1$.

**Lemma 2** indicates a method for detecting the existence of a deadlock-prone state.
Lemma 2

A set of tasks $T$ is in a deadlock-prone state if and only if $\text{L}(S_T) = |T|$.

Proof

Let $T = \{T_1, T_2, \ldots, T_k\}$ where the numbering of tasks is arbitrary, $k > 2$.

We will first show that if $\text{L}(S_T) = |T|$, then $T$ is in a deadlock-prone state.

Let $\{r_1, r_2, \ldots, r_k\} \subseteq S_T$.

Let $Q_0$ be the state of the system when $\text{L}(S_T) = |T|$.

Since $\text{L}(S_T) = |T|$, each task in $T$ must have at least one record locked.

We can assume that for $1 \leq i \leq k$, $r_i$ is locked by $T_i$.

Since each record in $S_T$ is contained in the intersection of the record lists of at least two tasks, we can assume that for $1 \leq i \leq k-1$, $r_{i+1} \cap r_i \cap \cdots \cap r_1$ and $r_1 \cap r_k \cap r_{i+1}$.

From system state $Q_0$, let task $T_i$ request record $r_{i+1}$, $1 \leq i \leq k-1$, and let task $T_k$ request record $r_1$.

The system will then enter state $Q_1$ in which the following condition holds:

- $T_1$ is blocked by $T_2$;
- $T_2$ is blocked by $T_3$;
- $\vdots$
- $T_{k-1}$ is blocked by $T_k$;
- $T_k$ is blocked by $T_1$;

Thus state $Q_1$ is a deadlock state.

From definition 3, it follows that $Q_0$ is a deadlock-prone state.

Therefore $\text{L}(S_T) = |T|$ implies that $T$ is in a deadlock-prone state.

It must now be demonstrated that the existence of a deadlock-prone state implies that $\text{L}(S_T) = |T|$. 

Let $Q_0$ be a deadlock-prone state such that there is a sequence of unsatisfiable requests which lead to deadlock state $Q_1$.

Assume that the tasks in $T$ are blocked in state $Q_1$ as shown;

- $T_1$ is blocked by $T_2$;
- $T_2$ is blocked by $T_3$;
- $\vdots$
- $T_{k-1}$ is blocked by $T_k$;
- $T_k$ is blocked by $T_1$;

If a task is blocking another task, the intersection of their record lists must be non-empty. Therefore, there is a set of records $(r_1, r_2, \ldots, r_k)$ such that

$$ S_T \subseteq S_T $$

for $1 \leq i \leq k{-}1$, $r_{i+1} \cap R_i \cap R_i+1$ and $r_1 \cap R_k \cap R_1$.

Since each task in $T$ is blocking another task, each task in $T$ must have at least one record locked in state $Q_1$. Therefore in state $Q_1$,

$$ L(S_T) = |T| $$

Since $Q_1$ was reached from $Q_0$ by a sequence of unsatisfiable requests, the set of records locked in $Q_1$ is identical to the set of records locked in $Q_0$. Therefore in state $Q_0$, $L(S_T) = |T|$ where $Q_0$ is a deadlock-prone state.

Example:

In the deadlocked and deadlock-prone states of preceding examples,

$$ S_4 = \{r_1, r_2, r_3\} $$

where the integer beneath the records indicate the task locking the record.

Thus, $L(S_4) = 3$. 
6. An Algorithm for the Prevention of Deadlock

From Lemma 2, we can see that if $L(S_T) < |T| - 1$ for all sets of shared records, then the system will be free of deadlock. This relationship between the number of tasks potentially and actively sharing data and the occurrence of deadlock forms the basis for a deadlock prevention algorithm.

Three commands and a response are necessary for operation in a deadlock-free environment. All commands and responses are transmitted among back-end processors. The commands are LOCK, UNLOCK and REQUEST. When a task desires to update a shared record, the back-end processor controlling that record issues either a LOCK or REQUEST command to the other back-end processors controlling records of tasks in any of the task interaction lists of the requesting task. The decision as to whether LOCK or REQUEST is sent is based upon relative task priorities. LOCK commands are sent to the back-end processors of lower priority tasks, while back-end processors serving tasks of higher priority receive REQUEST commands. If a back-end processor that has received a REQUEST command determines that the record is available, it issues a POSITIVE response. It is important to note that under the deadlock prevention algorithm a negative response is not necessary since a back-end processor issuing a REQUEST for an unavailable record or a REQUEST that would lead to deadlock will receive a LOCK command which invalidates the REQUEST. The UNLOCK command relinquishes control of a record. The detailed effect of each function is covered in the following definitions. A command or request is not necessary for the query of a shared record. A check of the shared record list will indicate if the record is available. In this discussion the term "update" will indicate both a read and write, while "query" implies a read only.
**Definition 4**

The REQUEST \( r_j \rightarrow T_k \) command issued by \( B_k \) to \( B_i \) results in the following operations:

If for all \( S_{T, i} \) containing \( r_j \),

a. \( r_j \) is unmarked in \( S_{T, i} \) and

b. either \( T_k \) has a record marked in \( S_{T, i} \) or \( m(S_{T, i}) \leq |M_i| - 1 \) for \( |M_i| \leq 2 \),

then \( B_i \) marks \( r_j \) in all \( S_{T, i} \) with the identifier of \( T_k \) and transmits a \( \text{POSITIVE} \ r_j \rightarrow T_k \) response to \( B_k \).

Otherwise \( B_i \) does not respond to \( B_k \).

Definition 4 indicates that two conditions must be satisfied before a back-end processor can signify that a record is available:

1. The record must not be claimed by another task.
2. If the first condition is satisfied, then it must be certified that granting control of the record to \( T_k \) would not cause any set of tasks to enter a deadlock-prone state.

**Definition 5**

The \( \text{LOCK} r_j \rightarrow T_k \) command, when issued by \( B_k \) to \( B_j \), causes \( r_j \) to be marked as locked by \( T_k \) in all shared record lists of \( B_j \).

**Definition 6**

The \( \text{UNLOCK} r_j \) command causes \( r_j \) to be unmarked in all shared record lists.

The purpose of the \( \text{REQUEST} \) command and the \( \text{POSITIVE} \) response is to confirm the availability of a record with higher priority tasks. Since the distributed database environment permits concurrent asynchronous operations on shared data, a back-end processor must query those back-ends that contain higher priority tasks which interact with the requesting task to verify the status of the record. If the record is available, a \( \text{POSITIVE} \) response is sent. If the record is unavailable, at the time of the \( \text{REQUEST} \) a \( \text{LOCK} \) command would
already be in transit to the back-end of the requesting task. The receipt of a LOCK command from a higher priority task invalidates a REQUEST. The LOCK command binds a record to a task while UNLOCK is used to release records.

The functions and responses are employed by the following algorithm to prevent the distributed DBMS from entering a deadlock state.

Algorithm 1

PART A

When task $T_k$ desires to update shared record $r_j$, the following steps must be taken by $B_k$ to prevent a deadlock state.

1. Check if $r_j$ is marked in any $S_{T,k}$ containing $r_j$. If so, $T_k$ must wait until $B_k$ receives an UNLOCK $r_j$ command. Note that if $r_j$ is marked in one $S_{T,k}$, it is marked in all $S_{T,k}$.

2. If $\exists S_{T,k}$ such that $m(S_{T,k}) = |X_T| - 1$, then $T_k$ must wait until a record in $S_{T,k}$ is unlocked.

3. Mark $r_j$ with the identifier of $T_k$ in all $S_{T,k}$ containing $r_j$.

4. For all higher priority tasks in any $S_T$ containing $T_k$, issue a REQUEST $r_j$, $T_k$ command to their back-end processors.

5. Wait for POSITIVE $r_j$ responses from all back-ends of step 4.

6. If while waiting, $B_k$ receives a LOCK $r_j$, $T_k$ command, then $B_k$ must issue UNLOCK $r_j$ commands to all back-ends which have transmitted POSITIVE $r_j$ responses, and then $B_k$ must return to step 1.

7. If while waiting, $B_k$ receives a LOCK $r_n$, $T_k$ command ($r_n \neq r_j$), and $m(S_{T,k}) = |X_T| - 1$, then $B_k$ must issue UNLOCK $r_j$ commands to all back-ends which have transmitted POSITIVE $r_j$ responses and then return to step 2.

8. When $B_k$ receives POSITIVE $r_j$ responses from all tasks in step 4 it issue a LOCK $r_j$, $T_k$ command to all lower priority tasks in any $X_T$ containing $T_k$. 
9. $T_k$ may then operate upon $r_j$.

10. Upon completion of the operations in step 9, $B_k$ issues an UNLOCK $r_j$ command to all tasks in any $X_T$ containing $T_k$.

PART B

When a back-end processor, $B_i$, receives a REQUEST $r_j$, $T_k$ command, it transmits a POSITIVE $r_j$ response if the requirements of Def. 4 are satisfied. If a POSITIVE response is transmitted, $r_j$ is marked with the identifier of $T_k$ in all $S_{T,i}$.

PART C

When a back-end processor, $B_i$, receives a LOCK $r_j$, $T_k$ command and it does not have a REQUEST $r_j$ command outstanding such that the conditions in steps 6 or 7 of Part A arise, $r_n$ is marked with the identifier of $T_k$ in all $S_{T,i}$.

It must now be demonstrated that Algorithm 1 prevents deadlock.

Lemma 3

Algorithm 1 prevents the system from entering a deadlock-prone state.

Proof

From Def. 1, it can be seen that under all circumstances, for any $S_T$, $m(S_{T,k}) = L(S_T)$ for all back-ends $B_k$.

Under Algorithm 1, a back-end processor may only issue a LOCK command if $m(S_{T,k}) = |X_T| - 1$ for all $X_T$. Thus immediately prior to the issuance of a LOCK command, $L(S_T) = |X_T| - 1$. After, the LOCK command has been issued, the maximum value of $L(S_T)$ is $|X_T| - 1$ for all $S_T$. Therefore according to Lemma 2, if the system operates under Algorithm 1, it cannot enter a deadlock-prone state.
Theorem 1

Algorithm 1 prevents the system from entering a deadlock state.

Proof

The theorem follows immediately from Lemmas 1 and 3.

The following example, illustrates the operating of a distributed DBMS with the deadlock prevention algorithm in effect.

Example 4

Assume the set of tasks in Example 1. We will follow the actions of the back-end processors $B_1$, $B_2$, $B_3$ which control data base access for $T_1$, $T_2$, and $T_3$ respectively. The only task set in which the value of $m(S_T)$ can be greater than 2 is $S_4 = \{T_1, T_2, T_3\}$.

In the example under consideration, $S_4 = \{r_1, r_3, r_7\}$. For notational convenience each $S_{4,i}$ will be denoted by an ordered triple in which the first element corresponds to the task, if any, marking $r_1$, the second element indicates the task marking $r_3$, while the final element represents the task marking $r_7$.

Assume the following set of references in the sample system.

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Record</th>
<th>Referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$T_1$</td>
<td></td>
<td>$r_1$</td>
</tr>
<tr>
<td>$t_0$</td>
<td>$T_2$</td>
<td></td>
<td>$r_7$</td>
</tr>
<tr>
<td>$t_0$</td>
<td>$T_3$</td>
<td></td>
<td>$r_3$</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>$T_1$</td>
<td></td>
<td>$r_3$</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>$T_2$</td>
<td></td>
<td>$r_1$</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>$T_3$</td>
<td></td>
<td>$r_7$</td>
</tr>
</tbody>
</table>

Once a task receives the second record it has requested, it requires 5 time units to complete its operation on those records.
For purposes of this example, the delays between back-end processors shown below are assumed.

- $B_1 \leftrightarrow B_2$ - 2 time units
- $B_1 \leftrightarrow B_3$ - 3 time units
- $B_2 \leftrightarrow B_3$ - 1 time unit.

Also, let the task priorities be $T_1 > T_2 > T_3$.

Given the requests listed above the following operations are performed in the system.

1. At time $t_0$
   \[ S_{4,1} = \{1, 2\} \quad S_{4,2} = \{1, 2\} \quad S_{4,3} = \{1, 3\}. \]

   $B_1$ issues $\text{LOCK}_{r_1} - T_1$ to $B_2$ and $B_3$.
   $T_1$ proceeds to operate on $r_1$.
   $B_1$ issues $\text{REQUEST}_{r_1} - T_2$ to $B_2$.
   $B_3$ issues $\text{REQUEST}_{r_1} - T_3$ to $B_1$ and $B_2$.

2. At time $t_1$.
   \[ S_{4,2} = \{1, 3, 2\}. \]
   $B_2$ transmits a $\text{POSITIVE}_{r_3}$ to $T_3$.

3. At time $t_2$.
   \[ S_{4,1} = \{1, 2\}. \]
   $B_1$ transmits a $\text{POSITIVE}_{r_2}$ to $B_2$.
   $B_2$ receives a $\text{LOCK}_{r_1} - T_1$.
   \[ S_{4,2} = \{1, 3, 2\}. \]
4. At time $t_4$,

$B_1$ receives $\text{REQUEST}_{r_3}, T_3$. Since $m(S_{4,1}) = 2$, the request is ignored.

$B_3$ receives $\text{LOCK}_{r_3}, T_1$. This command also indicates that the request for $r_3$ by $B_3$ will not be granted by $T_1$. Thus,

$$S_{4,3} = \{1, 2\}.$$

$B_3$ issues an $\text{UNLOCK}_{r_3}, T_3$ to $B_2$.

5. At time $t_5$,

$B_2$ receives $\text{POSITIVE}_{r_7}$ from $B_4$.

$T_2$ proceeds to operate on $r_7$.

$B_2$ issues a $\text{LOCK}_{r_7}, T_2$ to $B_3$.

$B_2$ also receives $\text{UNLOCK}_{r_3}, T_3$ from $B_3$.

$$S_{4,2} = \{1, 2\}.$$

6. At time $t_6$,

$B_3$ receives $\text{LOCK}_{r_7}, T_2$.

$$S_{4,3} = \{1, 2\}.$$

7. At time $t_{10}$,

$B_4$ issues $\text{LOCK}_{r_3}, T_1$ to $B_2$ and $B_3$.

$$S_{4,1} = \{1, 1, 2\}.$$

$T_1$ proceeds to operate on $r_3$.

8. At time $t_{12}$,

$B_2$ receives $\text{LOCK}_{r_1}, T_1$.

$$S_{4,2} = \{1, 1, 2\}.$$

9. At time $t_{13}$,
R_3 receives \text{LOCK}_{r_1} \ldots T_1
\quad S_{4,3} = \{1,1,2\}.

10. At time \( t_{15} \),
\( R_1 \) issues \text{UNLOCK}_{r_1} and \text{UNLOCK}_{r_3} to \( B_2 \) and \( B_3 \).
\quad S_{4,1} = \{1,1,2\}.

11. At time \( T_{17} \),
\( R_2 \) receives \text{UNLOCK}_{r_1} and \text{UNLOCK}_{r_3} and issues \text{REQUEST}_{r_1} \ldots T_2
\quad S_{4,2} = \{2,1,2\}.

12. At time \( t_{18} \),
\( R_3 \) receives \text{UNLOCK}_{r_1} and \text{UNLOCK}_{r_3} and issues \text{REQUEST}_{r_3} \ldots T_3
to \( R_1 \) and \( R_2 \).
\quad S_{4,3} = \{1,3,2\}.

13. At time \( t_{19} \),
\( R_1 \) receives \text{REQUEST}_{r_1} \ldots T_2
\quad S_{4,1} = \{2,1,2\}.
\( R_1 \) issues \text{POSITIVE}_{r_1} to \( R_2 \).
\( R_2 \) receives \text{REQUEST}_{r_3} \ldots T_3.
\quad S_{4,2} = \{2,1,2\}.
\( R_2 \) issues \text{POSITIVE}_{r_3} to \( R_3 \).

14. At time \( t_{20} \),
\( R_3 \) receives \text{POSITIVE}_{r_3} from \( R_2 \).
15. At time $t_{21}$,
   $B_1$ receives $\text{REQUEST } r_3, T_3$.
   $S_{4,1} = \{2,3,2\}$.
   $B_1$ issues $\text{POSITIVE } r_3$ to $B_3$.
   $B_2$ receives $\text{POSITIVE } r_1$ from $B_1$ and then issues a $\text{LOCK } r_1, T_2$ to $B_3$
   and proceeds to operate upon $r_1$.

16. At time $t_{22}$,
   $B_3$ receives $\text{LOCK } r_1, T_2$.
   $S_{4,3} = \{2,3,2\}$.

17. At time $t_{24}$,
   $B_3$ receives $\text{POSITIVE } r_3$ from $B_1$ and then proceeds to operate on $r_3$.

18. At time $t_{26}$,
   $B_2$ issues $\text{UNLOCK } r_1$ and $\text{UNLOCK } r_7$ to $B_1$ and $B_3$.
   $S_{4,2} = \{3,3\}$.

19. At time $t_{27}$,
   $B_3$ receives $\text{UNLOCK } r_1$ and $\text{UNLOCK } r_7$ from $B_2$ and then transmits $\text{REQUEST } r_1, T_3$ to $B_1$ and $B_2$.
   $S_{4,3} = \{3,3\}$.

20. At time $t_{28}$,
   $B_1$ receives $\text{UNLOCK } r_1$ and $\text{UNLOCK } r_7$ from $B_2$.
   $S_{4,1} = \{3,3\}$.
   $B_2$ receives $\text{REQUEST } r_1, T_3$.
   $B_2$ issues $\text{POSITIVE } r_7$ to $B_3$.
   $S_{4,2} = \{3,3\}$.
21. At time $t_{29}$,

$B_3$ receives $\text{POSITIVE } r_7$ from $B_2$.

22. At time $t_{30}$,

$B_1$ receives $\text{REQUEST } r_7, T_3$ from $B_3$.

$S_{4,1} = \{ 3, 3 \}$.

$B_1$ transmits $\text{POSITIVE } r_7$ to $B_3$.

23. At time $t_{33}$,

$B_3$ receives $\text{POSITIVE } r_7$ from $B_1$ and then begins to operate upon $r_7$. 
7. Efficiency of Deadlock Prevention Algorithm

It is difficult to treat the efficiency of Algorithm 1 without experimental evidence from a prototype distributed DBMS. There are several critical performance factors which could vary among distributed DBMS implementations. Network topology, degree of potential sharing among application tasks, and subschema size are among the system parameters that will have the strongest performance effects.

The number of back-end processors in the network along with the physical distance and type of connections among the back-end processors will influence the amount of communication overhead resulting from deadlock prevention. It should be noted that only back-end processors that execute database operations for tasks that share data have a need to exchange information in order to prevent deadlock. If both deadlock prevention and a high degree of efficiency are goals of a system design, then data shared by a group of tasks should be controlled by a minimal number of back-end processors. (Ideally, each such unit of shared data would reside on the storage of a single back-end processor.)

One environment under which the deadlock prevention algorithm could degrade performance is an on-line system in which each user may access any record in the entire database. In this situation, the deadlock prevention algorithm would force the DBMS to operate in a single threaded mode. If the subschema concept is applied to this unrestricted on-line environment, the undesirable effects of deadlock prevention can be virtually eliminated. Instead of permitting each on-line command unrestricted access to the entire database, the database can be partitioned into a logical collection of subschemas. Whenever a user issues an on-line database request, the appropriate subschema is invoked prior to the actual execution of the command.
The sub-schemas should be defined to encompass only that portion of the data base that the command may access. For example, if an airline reservation clerk updates a passenger list, the sub-schema would contain the passenger list for a given flight.

In an on-line environment in which the data base has been partitioned into sub-schemas the user need not sacrifice any flexibility of data access. Each user would interface with a re-entrant control program which parses each request, invokes the applicable sub-schema and then activates a host task to execute the request. By organizing an on-line system in this manner the negative performance effects of deadlock prevention are minimized.

8. Conclusion

The approach to deadlock prevention described here is a dynamic pre-claim technique [8] since it implicitly locks a set of records that could be required for an operation. The deadlock prevention scheme proposed for distributed data base systems has some similarity to the data base deadlock prevention mechanisms of Lomet [16] and Chu and Ohlmacher [4]. Lomet employs a graph-theoretical technique to avoiding deadlock. However, the information contained in the graphs is essentially the same as that maintained in the shared record list. The version of Lomet's algorithm presented in reference [16] does not consider the performance effects of operation in a distributed environment.

The process set of Chu and Ohlmacher is very close conceptually to the shared record list. The algorithm of Chu and Ohlmacher is also intended for distributed systems. The technique developed here differs from the approach of Chu and Ohlmacher in that it operates at the record level and in that a requesting task is given control of only that part of its shared record list necessary to avoid a deadlock-prone state. The feasibility of
Database sharing at the record level has been studied using simulation by Shemer and Collmeyer [20], who projected that even with a high degree of contention, performance degradation due to overhead would be minimal. Presently, several commercially available, single-machine database systems provide data sharing and locking at the record level [22].

The deadlock prevention algorithm is intended to prevent all possible deadlocks while allowing maximum database sharing at the record level. The results of the Section 6 demonstrate that the algorithm meets these criteria. Naturally, deadlock prevention incurs some overhead. However, careful planning by the designer of a distributed database application who is cognizant of the operation of the prevention algorithm can result in minimization of the overhead. For a distributed DBMS application to operate efficiently under the deadlock prevention algorithm, it is important that the database be partitioned into sub-schemas. However, once the sub-schema is defined, the individual application programs need not be aware of the mechanics of the deadlock prevention algorithm.

Due to the infrequency of deadlock situations, a deadlock detection schema requires less overhead than deadlock prevention in a single machine DBMS. However, the uncertain amount of overhead in distributed DBMS rollback added to the fixed overhead of the timeout mechanism, both of which are necessary operations in deadlock detection, leads to proposing deadlock prevention as the more satisfactory mechanism for handling deadlocks in a distributed database. The subject of rollback for a distributed DBMS is treated in reference [17], which presents an algorithm for minimizing the overhead of the rollback operation.

The lists computed by the prevention algorithm have potential application in two critical design areas of distributed database. The size of a task
interaction list, $X_T$, is an indicator of the amount of interference resulting from the activation of a data base task. Since task interference has an effect in system performance, the scheduler could use the size of $X_T$ as one of the weighting factors in the scheduling algorithm.

The contents of the shared record list would be of use to a prepaging memory manager [23]. Records that are unlocked, yet only requestable by a single task, could be transmitted to the page buffer associated with that task. The shared record list would be particularly valuable when used in conjunction with a Markovian paging model [11].
9. References


FIGURE 1

DISTRIBUTED DBMS