

PREVENTION AND DETECTION OF DEADLOCK IN DISTRIBUTED SYSTEMS
-- A SURVEY OF CURRENT LITERATURE

by

RAYFORD B. VAUGHN JR.

B.S., UNIVERSITY of SOUTHERN MISSISSIPPI, 1969

A MASTERS REPORT

Submitted in partial fulfillment of the
requirements for the degree .

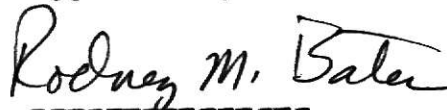
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

- 1982 -

Approved by:



Major Professor

SPEC
COLL
LO
2668
.R4
1982
V38
c.2

A11203 652718

ACKNOWLEDGEMENTS

I would like to take this opportunity to acknowledge several individuals who have been instrumental in the preparation of this report. On the professional side, I must express my sincere appreciation to my committee members, Professors Rod Bates, Beth Unger, and Paul Fisher, for their counsel and recommendations. A special thanks must be given to my major professor, Dr. Bates, for the many long hours of editing, advise, and explanations he provided on a weekly basis. Additionally, Dr. Tom Gallagher, Director of the Computing Center, provided a great service in helping me to understand and use the word processing facilities necessary for the physical preparation of this document.

On the personal side, I want to recognize my wife Dianne and my son Kenneth, for the understanding, love, and support they gave me throughout this endeavor. For this I will always be grateful. Finally, I give thanks to my Maker for providing the strength and opportunity to achieve this milestone in my life.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

CONTENTS

ACKNOWLEDGEMENTS	ii
----------------------------	----

<u>Chapter</u>	<u>page</u>
I. INTRODUCTION	1
II. DEFINITIONS AND TERMS	6
III. DEADLOCK DETECTION ALGORITHMS	17
INTRODUCTION	17
EATES MODEL	18
MENASCE and MUNTZ PROTOCOLS.	25
Hierarchically Organized Deadlock Detection	
Prctccol	25
Distributed Control Deadlock Detection	
Protocol	32
Summary.	35
COUNTEREXAMPLE TO THE MENASCE/MUNTZ PROPOSAL.	35
THE ISLCC - MARSLAND ON-LINE DETECTION	
TECHNIQUE	40
OBERMARCK'S DEADLOCK DETECTION ALGORITHM	42
Introduction	42
The Obermarck Algorithm	45
Summary	48
THE RECOVERY PROBLEM	48
CONCLUSION	49
IV. AVOIDANCE AND PREVENTION OF DISTRIBUTED DEADLOCK	51
INTRODUCTION	51
AVOIDANCE OF DEADLOCK	53
PREVENTION OF DEADLOCK	56
Introduction	56
The Prevention Algorithm	59
V. CONCLUSION	63
BIBLIOGRAPHY	66

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
1. Distributed Deadlock (bidirectional communication) .	4
2. Traffic deadlock (8)	6
3. Local Wait Graph with Remotely Connected Arcs. . . .	21
4. Reduction of Graph Shown in Figure 3.	22
5. Lock Controller Hierarchy (12)	26
6. Deadlock Cycle Involving Transactions T1, T2, ... , Tk.	34
7. Initial TWF Status by Site Origin	34
8. Isloor - Marsland Example	40
9. Wait-for graph in centralized system (OBERMARCK) . .	43
10. Obermarck's wait-for graph - distributed	44
11. OBERMARCK's graph with communication links	45
12. Example of a Claimed System Resource Allocation Graph.	54

Chapter I

INTRODUCTION

Since the first notion of multiprogramming was implemented, the data processing community has been forced to consider the issue of "deadlock" in terms of its prevention, avoidance, or detection. In general terms deadlock occurs when two or more processes each hold some resource in the system and are blocked from proceeding until they receive another resource that is already assigned to one of the other blocked processes. When this situation occurs no further action can take place by any of the involved processes until at least one releases its claim on its assigned resource. This claim release normally entails an abort, rollback, and a restart. These terms will be further discussed and defined in Chapter II.

In 1971, HOLT (6) published a study of the deadlock problem as it occurred in a single system multiprogramming environment. This paper became a basis for others to expand upon and many of his ideas were carried over into strategies for distributed systems. HOLT delineated three classes of deadlock strategies: (1) Prevention, (2) Detection, and (3) Crash. This report will examine the first two classes as they apply to distributed systems. The third class, crash,

really is a strategy of "no strategy" and simply allows the system to fail requiring a manual restart. Although this could very well be a viable alternative in some circumstances, this paper will not address its merits. The prevention of deadlock was addressed in 1969 by HABERMANN (5) by requiring that all processes state their "claim" for required resources prior to executing. Once the resources required by all processes are known, the system can insure that a state is never entered where deadlock might occur. This same basic preclaiming requirement exists today in distributed systems deadlock prevention algorithms and is obviously subject to two significant drawbacks. First, in many database systems resource demands are created dynamically and cannot be predetermined. Second, this prevention mechanism can cause unnecessary delays in process execution in that a particular process may be delayed so that it does not move into a state with "potential" deadlock. HOLT's detection strategy assumes a different perspective. Deadlock is allowed to occur (hopefully infrequently) and periodically the system will execute a detection algorithm to check for deadlock and abort processes found to be in such a state. The technique is based upon the creation and systematic reduction of a bipartite directed graph known in the literature as a general resource graph. A successful reduction indicates no deadlock and an unsuccessful reduction causes some abortion

technique to be used. Many refinements have been made to HOLT's algorithm and the problem of deadlock detection has been well studied and implemented in conventional systems. Its direct extension to distributed systems however is not possible and has opened up a new area of research.

During the early to mid 1970's the ADP community began to experience a natural convergence of technologies - that of communications and networking of computers. This evolved into what we now term distributed systems and distributed databases with their concomitant problems. Operating systems had to contend with a more autonomous behavior which aggravated control problems, with communication time delays which rendered coordination between system controllers more difficult, and with geographically distributed databases. The information needed to detect deadlocks was now distributed over many separate sites (8). The communication time delay problem is of particular significance when one considers that a deadlock may be reported when it no longer exists or may exist and not be reported in meaningful time. The distributed deadlock problem can be viewed in its simplest form in figure 1 below.

Here we find that processes A, B, C, and D each have claimed a resource (a,b,c, or d) at their respective sites and require another resource at a remote site. A claimed resource is denoted by ASGD--->, and a requested resource is shown as REQ---> (e.g., ASGD--->d, REQ--->a). Each

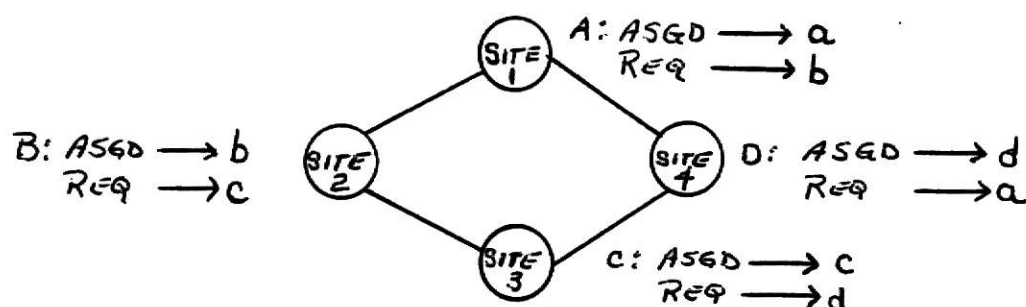


Figure 1: Distributed Deadlock (bidirectional communication)

autonomous operating system can only detect deadlock within the confines of its site and cannot detect the global deadlock that has obviously occurred. Communication (additional overhead) becomes the next solution, but it is certainly not a trivial one. When redundancy of data items is considered within the network of databases, the problem grows exponentially. When coupled with out-of-sequence message receipt between nodes (sites) on a network, one can readily understand the challenge that results. It is the intent of this report to investigate current thoughts on the distributed deadlock problem and to synthesize these approaches into a single document.

In researching material from which to write this report, it became increasingly evident that not many unique approaches existed to the distributed deadlock problem and few papers had been published on the subject - not all of which are practical or even correct. A complete bibliography is included to aid the reader in further studying this problem. The report itself is divided into

five chapters. The introduction is designed to provide a short historical brief of the problem and to demonstrate the difference between conventional deadlock and distributed deadlock. Chapter II will introduce terminology found throughout the literature and present definitions, references, and diagrams to further the reader's understanding of subsequent topics. The terminology used to describe the deadlock problem varies considerably between authors and is a potential issue for standardization. The third chapter will present deadlock detection strategies recently published for distributed architecture. This technique is by far the most popular and will constitute the bulk of the report. A forth chapter will discuss prevention and avoidance of deadlock in distributed systems and comments concerning the feasibility of utilizing these techniques. The report concludes with a summary of information presented and a brief discussion of a "mixed approach" to the deadlock problem. Comments concerning recommended areas for further research are included.

...

Chapter II

DEFINITIONS AND TERMS

The problem of deadlock is actually an outgrowth of the larger set of problems contained within concurrency and synchronization topics. Deadlock occurs in a distributed system when two or more processes request the use of a set of shared resources in such a sequence that results in the activation of each process being dependant upon the acquisition of a resource held by another process whose activation is also blocked (11). This situation can then only be alleviated by the involvement of an external agency. ISLOOR and MARS LAND (8) draw an analogy between computer system deadlock and a traffic situation where four cars arrive at a fourway stop intersection at the same time (see figure 2).

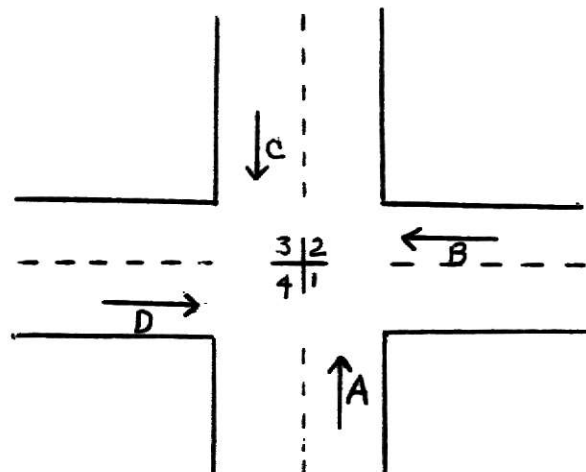


Figure 2: Traffic deadlock (8)

In this example the automobiles represent computer processes and the four quadrants of the intersection represent the shared resources. Each automobile (process) requires exclusive control of its quadrant (resource) when it enters the intersection in order to successfully proceed. If two or more of the automobiles are blocked and prevented from proceeding, deadlock has occurred. The traffic example demonstrates another important point in the deadlock problem - if on one occasion all four vehicles arrive at the intersection at the same time and all turn right, no deadlock occurs, but if on a second occasion all decide to proceed straight ahead or make a left turn, deadlock most certainly occurs. In a computer system the same situation occurs with shared resources. If by chance the resources are not required at the same time no deadlock occurs, but if each process requires another's resource, the system enters the deadlock state.

As stated in the introductory chapter, several classes of strategies have evolved. One approach is that of deadlock prevention. Prevention entails building the system such that deadlock cannot occur. Different methods of accomplishing this have been proposed and may involve preemption, preclaiming, or resource ordering. Preemption means that whenever a process requests a resource that cannot be granted, other resources held by the process are taken away. The prime disadvantage here is the potentially

large number of unnecessary preemptions. Preemption in itself is not necessarily expensive. For example, if the operating system decides that a processor must be preempted in order to free blocked processes, the cost is negligible and can be done with almost no loss of progress. If, however, preemption of a DB request is required, several costly actions are necessary to maintain consistency of the data base. Generally, a rollback is required to a previous safe state and any modification accomplished to the DB must be undone. Preclaiming techniques require that processes identify all resources that will be required prior to assignment of any resources. Resources then may all be exclusively assigned at once to the process and retained until process termination. One must keep in mind that resources can be consumable (data items or intra-process communications for example) or reusable (tape drives, disk units, memory space for example). In many cases it becomes extremely difficult, if not impossible, to determine all data base requirements of a process in a distributed environment with a distributed data base (DDBMS). Often these requirements are created dynamically. For example, it is not uncommon for the transaction or process to first look at a value contained in some table or index and assume various, different resource requirements based on that table. Each time the process is run, the resources required are different. The other disadvantage is that resources

assigned will be held for much longer periods than actually needed thereby reducing concurrency of operations. The resource ordering strategy is a sophisticated technique that has been developed for conventional systems. It involves placing all resources in ordered classes. Processes may request resources one class at a time and only in ascending order. For example, the process must claim all required resources in class 1 before it can claim a resource in class 2. Once class 2 is entered it can never go back and claim more resources in class 1. The rationale behind this procedure is the prevention of cycles, therefore the prevention of deadlock. Major disadvantages are prolonged holding of resources when they aren't required and a re-education of the user as to what classes contain needed resources. Additionally, this technique may be impossible to implement with dynamically created requirements.

Deadlock detection strategies are far more common for the distributed environment. This technique involves the periodic execution of some algorithm searching throughout the network for cycles of resource requests. The detection methodology does not solve the problem, it only identifies it. Once identified, some other procedure must be invoked in order to abort a process, thereby freeing another process. Detection strategies rely heavily on graph theory and its practical representation in some machine acceptable data structure. This theory will be covered in some detail

within the next chapter - DEADLOCK DETECTION. A disadvantage of detection is that deadlocks are allowed to occur and remain in effect tying up resources until located and broken. How often to execute the detection algorithm becomes a system performance consideration. The advantage to this method is that only processes that were actually in a deadlock situation are preempted.

Deadlock avoidance, a third strategy, is sometimes addressed in the literature as a combination of detection and prevention. An excellent definition proposed by HOWARD (7) is that an avoidance algorithm projects detection into the future in order to keep the system from committing itself to an allocation which will eventually lead to a deadlock. The resource allocator must test each possible allocation and make only those that will lead to a safe state. This technique has the advantage of avoiding preemption but tends to be overly cautious and may cause process delays that are not necessary. This also requires a pre-established claim and may be impossible with dynamically created requirements.

A unique anomaly that may occur in a distributed system deadlock detection algorithm is that of a "false deadlock" (or sometimes referred to as a phantom deadlock). Unlike a conventional environment, a communications complication may occur in a distributed system resulting in messages arriving out of order. In a detection algorithm that utilizes the

construction of a directed graph to keep track of blocking pairs, it is quite possible that a message to delete an arc between two nodes may arrive prior to the message that created the arc. If the system does not allow for management facilities to store out-of-order requests, the delete request may be lost resulting in the graphic representation of two blocked processes that in fact are not actually blocked and have released whatever resources they once held. This then creates the false deadlock situation and may invoke a preemption unnecessarily.

Previous discussion has suggested the heavy reliance on graph theory in modeling current states of system operation. A brief discussion of terms commonly found is in order at this point. The following definitions were extracted from HOLT (6) for the most part and seem to be well known. A "directed graph" is a pair (N, E) , where N is a nonempty set of "nodes" and E is a set of "edges" (or arcs). Each edge in E is an ordered pair (a, b) , where a and b are nodes in N . For given nodes a and b it is permissible for E to contain more than one edge of the form (a, b) . An edge (a, b) is said to be "directed from" node a and "directed to" node b . The graph is said to be "bipartite" if the set of nodes N can be partitioned into disjoint subsets Φ and Ψ such that each edge has one node in Φ and the other node in Ψ . A "sink" is a node with edges directed to it but no edges directed from it, and a "source" is a node with no edges

directed to it but has at least one edge directed from it. A "path" is a sequence $(a,b,c, \dots, r,s)$ containing at least two nodes, where (a,b) , (b,c) , $\dots$, and (r,s) are edges. A "cycle" is a path whose first and last nodes are the same. Deadlocks are frequently detected by following paths and looking for cycles. The "reachable set" of a node is the set of all nodes b such that a path is directed from a to b .

Detection of deadlocks in a centralized database system can be accomplished by periodically searching for cycles in a graph that represents the current state of the system in terms of processes (nodes) and resources assigned (e.g., data items or areas). This is a good application for the bipartite graph implementation previously discussed. If one considers the set of resource nodes to be PHI and the set of process nodes to be the set RHO , the two sets are disjoint. Each arc in the cycle is an ordered pair with one node a member of PHI , and the other a member of RHO . In a DDBMS it becomes quite inefficient to maintain such a "state graph". MENASCE and MUNTZ (12) described a more efficient mechanism termed a "transaction-wait-for (TWF) graph". This proposal is based on the consideration that users interact with the database via transactions. A "transaction" is a sequence of actions which can be either read, write, lock, or unlock operations. Furthermore, transactions are required to lock resources before and unlock them after operations are performed. A transaction itself does not migrate from site

to site, but does generate an "agent" or "transaction incarnation" at other sites to operate in its behalf if needed. In a TWF there is a directed arc created from T' to T" if T' is blocked by T". Other rules defined will be more fully expressed in Chapter III, however, the detection process is predicated upon the search for cycles among transactions that are blocked pending assignment of a requested resource.

Many of the unique deadlock problems are caused by the distribution of a database over several sites. For this reason it is appropriate at this point to discuss transaction considerations in the sense described in the preceding paragraph. If the action of a transaction involves only a single site, it is known as a "local" transaction. A distributed transaction involves resources at several sites. The assumption is generally made that transactions are implemented as a collection of processes (12) which then act on behalf of the transaction. These processes are termed transaction incarnations. Once a transaction requires interaction with another site, it loses its identity as a transaction at any single site and is represented by an incarnation (process). It is only seen as a "transaction" at the global level. One or more of the transactions may exist at any particular site and may be in one of two states - either "active" or "blocked". A transaction is blocked if it cannot proceed because a needed

resource is being held by another transaction. Otherwise, it is considered active.

Several definitions of terms associated with concurrency control and synchronization techniques in decentralized systems are now in order. A transaction may "lock" a resource to insure its inaccessibility by other transactions while the resource is in some inconsistent state, such as a partially completed update. In an environment where a transaction can modify any object it accesses, it must conform to the following rules:

1. Lock all objects before accessing them.
2. Do not lock an object that is already locked.
3. Before completion, unlock every object that it locked.

Transactions that abide by these rules are said to be "well-formed". As a follow-on, a transaction is "two-phase" if it first locks all objects before it unlocks any and once an unlock is issued, no further locks are allowed. A "timestamp" is a unique number assigned to a transaction and is chosen from some monotonically increasing function (e.g., system timer) (9). Timestamps used in a concurrency control mechanism can be helpful in a distributed environment to avoid deadlocks since they are unique and specify an ordering of all transactions in the system. Deadlocks can also be broken utilizing a similar technique known as "time-outs" in which transactions or processes are

terminated after having been in a wait state for some predetermined time. The obvious problems here are twofold; first, how does one select the proper time period for the time out, and second, restarts will be unnecessarily generated.

Once deadlock has been detected a new realm of problems is entered, that of recovery from preemption. It is not the intent of this report to examine the problems and techniques of recovery, but a general familiarity of terms associated with the process will be beneficial to the reader. Recovery refers to a process the system must go thru in order to put itself in a state of normal operation after some abnormal event has occurred such as system crash, deadlock, user abort, or unforeseen application program problems. When deadlock occurs the only solution is to abort one or more deadlocked processes. The operating system should have some capability to "rollback" the aborted process to a previous "safe state" and "restart" it at a time when deadlock is not likely to again occur. For some peculiar situations in database systems two or more processes may loop by continually blocking, aborting, and restarting each other. This is known as "cyclic restart" and must be avoided by the restart mechanism (8).

Other terms the reader might find useful while examining this subject are direct versus indirect wait, reliability, failure, and ordered blocked process lists (OBPL).

Transaction T is said to be "waiting directly" on T' if the resource needed for its continued execution is being held by T' . Transaction T_1 is said to be "waiting indirectly" on T_k if there exists a set of transactions $T_2, T_3, \dots, T_{k-1}$ such that T_i is waiting directly on T_{i+1} for $i=1$ to $k-1$. The "reliability" of a system (9) is said to be a measure of the success with which the system conforms to some authoritative specification of its behavior. When behavior deviates from that which is specified for it, the system encounters "failure". An ordered blocked process list is a detection technique where each process in the OBPL is waiting for a resource held by the next process in the list. This ordered list is transmitted between installations in the detection algorithm (8).

This concludes definitions and terms considered necessary for an understanding of material in subsequent chapters. The reader is invited to review the noted references for a more complete discussion of any singular term.

Chapter III

DEADLOCK DETECTION ALGORITHMS

3.1 INTRODUCTION

Given the increasing popularity of distributed systems and growing interest in distributed databases, the requirement to detect deadlock states becomes more critical to system performance. In a distributed system a collection of processes run concurrently on geographically separate computers and communicate with each other via message exchange. There are two important differences here between a conventional multiprocessing system and a distributed one as demonstrated by GLIGOR and SHATTUCK (4). There exists no globally shared memory and secondly, delays may occur, in message transfer between sites, that are critical to the proper operation of the system. From a theoretical viewpoint, deadlock detection in distributed systems can be accomplished in much the same manner as in traditional systems, however the lack of a globally shared memory forces the network to maintain and communicate both local and network state information in some type of graph-like data structure. This becomes a source of significant overhead and may result in performance degradation that is not negligible. Additionally, the possibility of control

messages arriving throughout the network out of sequence further complicates the problem. Detection mechanisms must somehow account for these delays and insure that controlling functions continue to operate properly. All detection techniques assume that requested resources will be eventually granted and then periodically invoke some algorithm to search the system looking for any deadlock states. If deadlock is found, the system must select processes and preempt their resources in order to free others. This procedure should be transparent to the user and involves rather sophisticated rollback and recovery techniques. This chapter will examine recent proposals for detection algorithms and explain their approach.

3.2 BATES MODEL

The Bates model (1) was developed in 1978 and involves the use of "wait graphs" located at each node of the network. The wait graph is a directed graph with a node representing each process in the system. It is constructed by creating arcs between nodes that have demonstrated a need to wait on each other because of a resource sharing conflict. If process "a" is waiting on a resource which, because of queueing rules, must be granted to process "b" before a, then a is said to be waiting on b and an arc is created from node a to node b. The algorithm then searches for the existence of cycles in a two step procedure. First,

the local network node is checked for deadlock in the same manner one might expect in a traditional system, then the graph is reduced at each site (a procedure to be discussed later) and a second step is started in looking for global deadlock thru a token passing scheme. The algorithm itself searches for deadlocks on a continuous basis by occupying a background process at each site. This, of course, necessitates an architecture capable of multiprogramming throughout the network. Two graphs are actually maintained at each site - a current state graph maintained by the site operating system and updated each time resources are requested or released, and a copy of the current wait graph made by the background process each time it begins the reduction process. This procedure will sometimes result in deadlock being shown in the current graph but not in the graph maintained at the moment by the background process - however, by definition of deadlock, once it occurs it will remain in effect until broken, therefore the deadlock will be detected during subsequent iterations of the algorithm.

When the detection process begins at any given site, a copy of the current wait graph is made by the background process. It is then checked for local deadlock by searching for cycles. If there are any nodes which are sources or sinks, they are deleted along with their attached arcs. This procedure is repeated until there are no nodes remaining or until no further reduction can be accomplished. If any

nodes and arcs remain, they are cyclic and participate in deadlock. The system must then choose some method of preemption and restart to break the deadlock in such a manner that it is unlikely to immediately recur.

After determining that the local site is free of deadlock, global deadlock must be checked. This is accomplished by augmenting the up-to-date wait graph with a representation of arcs that originate in this network node but terminate at another network node and vice versa. The identity of the remote node and process is included in the augmented graph. The following procedure for global deadlock checking was extracted from reference (1). First it is insured that the local wait graph has no deadlocks. Since the current wait graph could have changed by the time this is done, there must be a copy of the original wait graph which has just been checked for local deadlocks. This copy is reduced by removing every graph node which has no arc whose other end is connected outside the network node. For each graph node n so removed, and for every pair of arcs x and y such that x terminated at n and y originated at n , construct a new arc which originates where x originated and terminates where y terminated. Since n had no remotely connected arcs, this new arc will not be remotely connected at either end. If there are two or more arcs which all originate at the same node and all terminate at the same node, all but one are eliminated. This reduced graph now

contains only graph nodes with remotely connected arcs and has local arcs for each local path which the original graph contained between graph nodes that now remain. Figure 3 shows a conceptual example of a local wait graph with some remotely connected arcs as well as existing local arcs. The remote arcs show direction by the arrow heads or tails at the opposite end. The dotted boundary is that of the network node. Figure 4 shows the corresponding reduced graph with the labeling of graph nodes and remotely connected arcs preserved. Local arcs are labeled with all the arc identifiers that it replaced in the unreduced graph.

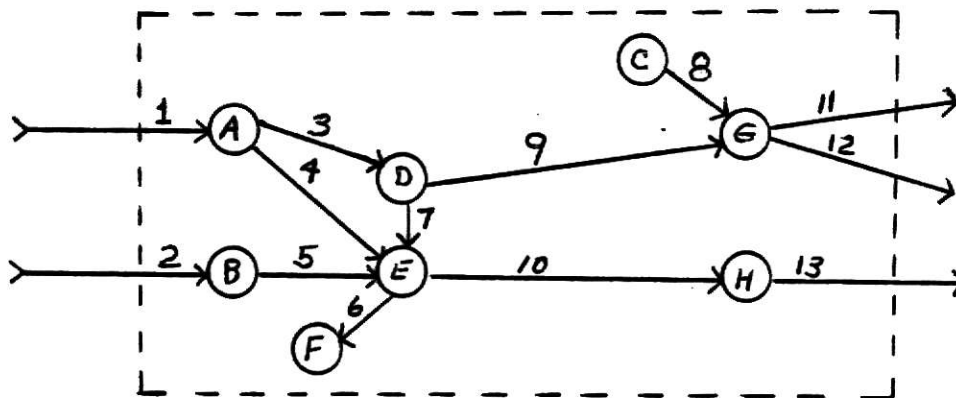


Figure 3: Local Wait Graph with Remotely Connected Arcs.

Upon completion of the graph reduction shown above, the deadlock detection process sends a message corresponding to each remotely connected arc (arcs 11, 12, and 13 in figure 4). This message contains the identity of the network node, the identity of the original process, and the arc iden-

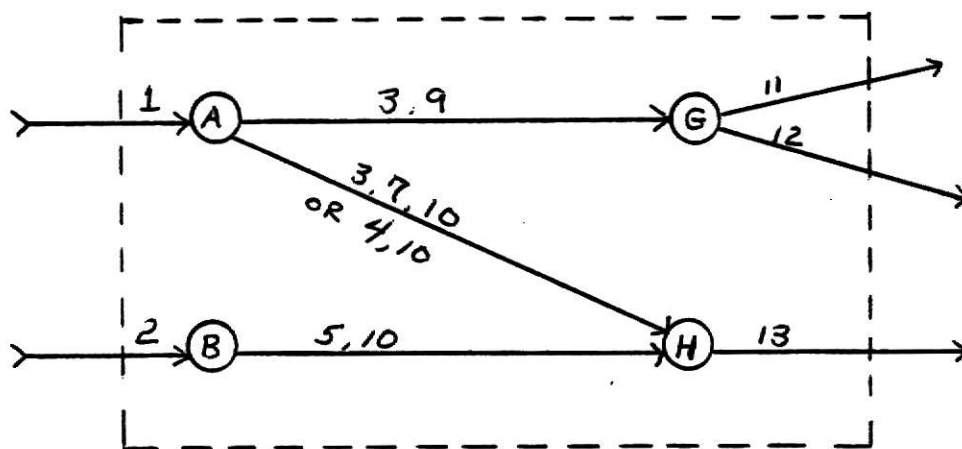


Figure 4: Reduction of Graph Shown in Figure 3.

tification. When it arrives at the receiving node at a different site it is further propagated throughout that sites reduced graph by transmitting it along exiting arcs from the receiving process node. If none exist, the message is discarded. Upon termination of this process, the message originally transmitted will have arrived at (and only at) each graph node which has a remotely connected exiting arc and which is accessible from the graph node where the message originally arrived. Since the message contains all the identification information of where it originated, it is possible to determine whether or not a message returns to its original process node. Everytime an incoming message is received, it is checked to insure that it did not originate at the receiving process node. If not, it is propagated along exiting nodes as explained before; if so, a cycle has been formed and therefore, this process node is involved in a deadlock. Recovery procedures are then initiated to preempt, rollback, and restart selected processes.

Since each separate site is performing the above process in a background environment, one must consider the impact of communication among wait graphs that are not all equally timely. Several cases are possible when considering two sites s and r and their wait graphs:

1. Arcs that are present in one reduced graph but have been eliminated in its corresponding up-to-date graph pose no problem since they could not have been involved in a deadlock to begin with (deadlock cannot disappear on its own).
2. Arcs not shown in the reduced graph, but which have been added to the up-to-date graph, will be detected during subsequent executions of the detection process.
3. If node s shows an entering arc from node r , but node r shows no such exiting arc, no problem exists since node r will not attempt to send a message to s . This situation will occur frequently because of the differences in times each node creates its reduced wait graph.
4. If r 's reduced graph shows an exiting arc and r sends a message over it but s 's reduced graph does not show the arc entering, we have two possible circumstances - either the missing arc has been eliminated in the up-to-date graph of r or it has been created in the up-to-date graph of s . If it has recently been

created in s , the message must be held pending the next graph reduction cycle. If it has been deleted in r , s should discard the message. Two possible solutions exist here to handle this particular situation. The algorithm can check the up-to-date graphs of r and s to determine what to do with the message or it can simply discard the message in all cases, since the deadlock will still exist in the next cycle and will be detected then. Naturally this second alternative results in longer delays for processes involved in a deadlock state - a performance tradeoff.

The communications requirements in the above procedures are quite heavy. If this traffic seriously degrades performance, a rule could be imposed that a node only originates a cycle tracing message once in the life of an outgoing arc. If this approach is chosen, the node receiving a message over an arc of which it has no record in its reduced graph, may not discard that message without checking up-to-date graphs. If it is permitted to discard the message a deadlock may be allowed to occur without detection.

The BATES model has several advantages associated with it. It is somewhat flexible in its implementation, easy to comprehend, checks for deadlock continuously, and makes allowances for reduced network communication. It is

designed for use in an environment where deadlock is infrequently experienced. This is a common characteristic of detection algorithms since the recovery procedure is expensive and has a severe impact on system performance.

3.3 MENASCE AND MUNTZ PROTOCOLS.

MENASCE and MUNTZ (12) describe two protocols for the detection of deadlocks in distributed databases - a hierarchically organized one and a distributed one. Both will be addressed in this report. A graph model, very similar to the one described in the previous section, is used by both protocols and a cycle in the graph is considered a sufficient condition for the existence of deadlock. The reader may wish to again review the definitions of transaction, incarnation, and transaction-wait-for (TWF) graph presented in Chapter II before continuing.

3.3.1 Hierarchically Organized Deadlock Detection Protocol

The following explanation has been basically extracted from reference (12) with some paraphrasing. We assume that our database (DB) is partitioned into a set of subdata bases DBi's such that DB is the union of all DBi's and DBi and DBj are disjoint for $i > j$ (i.e redundancy is not allowed, and each site has exactly one DBi). The protocol utilizes a hierarchy of lock controllers which interact to detect

deadlock. Two types of controllers exist - leaf controllers (Lk's) and nonleaf controllers (NLK's). A leaf controller Lki is assigned to each subdata base DBi. Figure 5 shows three leaf controllers and two nonleaf controllers in their hierarchical arrangement.

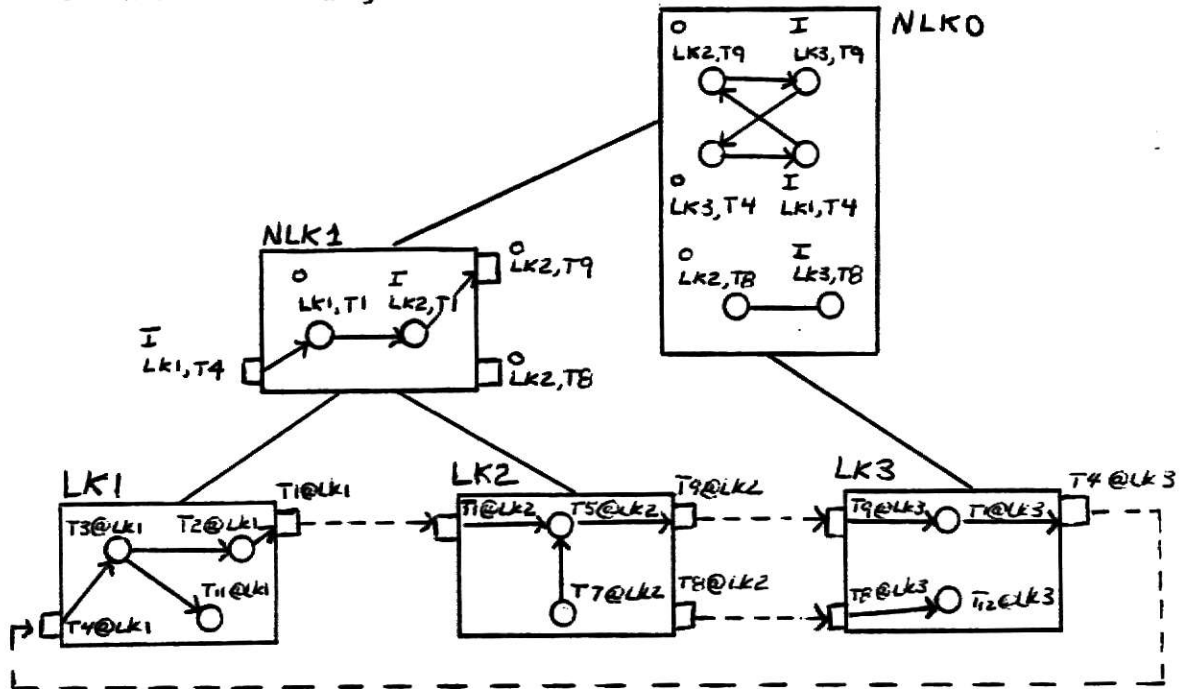


Figure 5: Lock Controller Hierarchy (12)

Each leaf controller maintains a TWF for its site containing all the nodes of the global TWF associated with incarnations local to Lki. This local TWF is represented by $TWF(Lki)$. Two special kinds of nodes are recognized within a leaf controller and are used to represent arcs between sites:

1. A node in $TWF(Lki)$ is known as an "output port" and is denoted $O(Lki, T)$ if the global TWF contains an outgoing arc from an incarnation of transaction T local to Lki into a nonlocal incarnation of T.

2. A node in $TWF(Lki)$ is called an "input port" and is denoted $I(Lki, T)$ if in the global TWF there is an incoming arc into an incarnation of transaction T local to Lki from a nonlocal incarnation of T .

The rule is made that there can exist only one incarnation per site per transaction, therefore all labels assigned to input and output ports will be unique. Dashed lines in figure 5 indicate arcs in the global TWF (e.g., $T1 @ Lk1$ to $T1 @ Lk2$). Nonleaf controllers only concern themselves with the input and output ports of the leaf controllers and the nonlocal arcs connecting them. Each NLK maintains an input-output-ports (IOP) graph. Nodes of an IOP are associated with input and output ports of leaf controllers and are termed i-nodes and o-nodes. It is permissible for certain i-nodes and o-nodes to be input ports and output ports respectively, for the IOP. The IOP for controller $NLKi$, denoted $IOP(NLKi)$, is defined by the following rules:

1. Arcs from i-nodes can only go to o-nodes and vice versa.
2. There is an arc from o-node Oa to i-node Ib if Oa is an output port of a leaf controller in the subtree rooted at $NLKi$ and Ib is a corresponding input port of another leaf controller in the same subtree. An example is the arc from $O(Lk1, T1)$ to $I(Lk2, T1)$ in the IOP of $NLK1$ in figure 5.

3. There is an arc from the i-node I_a to c-node O_b in $IOP(NLKi)$ if there is a path from an input port I_a to an output port O_b of a son of $NLKi$. The path must be within one controller. This is demonstrated in figure 5 by the arc from $I(Lk1, T4)$ to $O(Lk2, T9)$ in $NLK0$.
4. An input (output) port of $IOP(NLKi)$ is also an input (output) port of a leaf controller in the subtree rooted by $NLKi$. In figure 5, the input port of $IOP(NLK1)$ is also an input port in $Lk1$.

Before the hierarchical protocol is explained it is important to define one more term unique to this technique. The "lowest common ancestor" between controllers $K1, K2, \dots, K_n$, denoted $LCA(K1, K2, \dots, K_n)$, is the common ancestor between them at the lowest level in the hierarchy of controllers (the root is considered the highest level). Rules 1 through 3 below developed by MENASCE and MUNTZ describe the protocol:

1. Rule 1 (Transaction incarnation T requests a local resource): The requested resource R is in the same subdata base as the transaction incarnation T . Let Lki be the controller for resource R .
 - a) If the resource R is not being held by any transaction and is free for assignment, assign it to transaction incarnation T .

- b) If the resource cannot be granted because the set of transactions $(T_1, T_2, \dots, T_k)$ currently hold resource R , then add an arc from (T, L_{ki}) to (T_j, L_{ki}) for $j=1$ to k . Check the $TWF\{L_{ki}\}$ for the existence of cycles.
 - c) If cycles were formed, there exist one or more local deadlocks and an appropriate recovery mechanism should be invoked.
 - d) The addition of the arcs mentioned in b above may have created one or more paths between input and output ports of L_{ki} . For each such path, send the (input port, output port) pair which delimits the path to the father of L_{ki} .
2. Rule 2 (Transaction T requests a nonlocal resource):
- The requested resource R is in a different subdata base from the previously requested resource, therefore it has to be acquired by an incarnation of T local to R . Let L_{ki} be the controller of the site where the request is made and let L_{kj} be the controller of the site where resource R is located. The incarnation of T at L_{ki} becomes blocked and waiting for a message from the incarnation of T at L_{kj} . The node (T, L_{ki}) is now an output port of the TWF graph at L_{ki} and the node (T, L_{kj}) is an input port of the TWF graph at L_{kj} .

- a) Create an arc from $O(L_{ki}, T)$ to $I(L_{kj}, T)$ in the IOP graph of the lowest common ancestor between L_{ki} and L_{kj} .
 - b) Add an o-node labeled $O(L_{ki}, T)$ to the IOP graph of each controller in the path between (but not including) L_{ki} and $LCA(L_{ki}, L_{kj})$. Each such o-node is also an output port of the corresponding IOP graph.
 - c) Add an i-node labeled $I(L_{kj}, T)$ to the IOP graph of each controller in the path between (but not including) L_{kj} and $LCA(L_{ki}, L_{kj})$. Each such i-node is also an input port of the corresponding IOP graph.
3. Rule 3: When an arc is added to $IOP(NL_{Ki})$, the following occurs:
- a) If a cycle is generated by the addition of the new arc, then a global deadlock has been detected and appropriate action is required to resolve it.
 - b) If no cycle was generated, then check whether any input-output port connection has been generated in $IOP(NL_{Ki})$ and report the end points of any such connections to the father of NL_{Ki} .

Nonleaf controllers must be continually updated. When a lock release occurs, one or more arcs must be deleted from a TWF graph of a leaf controller. All i-o paths that contained this arc are now broken and must be reported up

the chain of controllers until the deletion of an arc from a graph does not cause any i-o path to be broken. As with the BATES model, the algorithm can be modified to reduce the constant communication burden imposed on the system. Information concerning connection between input and output ports can be transmitted periodically. Depending on the time interval chosen for the period, a reduction in communication may be achieved at the expense of allowing deadlock to exist for the same interval of time or possibly longer. A proof of the hierarchical deadlock detection protocol is offered in reference (12).

The selection of controller locations can be considered a tuning factor in attempts to optimize system performance. One should monitor the DB activity in the distributed system and note which "clusters" of leaf controllers have a high percentage of DB traffic among themselves while demonstrating little activity with other clusters. The assignment of nonleaf controllers to each cluster so identified will result in better system performance. This concept can be continued up to the root NLK. The optimization of such a system is certainly an area for further research and investigation.

3.3.2 Distributed Control Deadlock Detection Protocol

The second protocol proposed by MENASCE and MUNTZ involves the decentralization of detection to site level, as opposed to a centralized detection mechanism. In this protocol some redefinitions are in order. Each database site controls a set of resources. Transactions request a resource through the site controller. Controllers are responsible for processing lock/unlock requests for their resources originating from any network node, and for building a simplified version of a TWF graph to detect deadlocks. A TWF in this protocol is somewhat different than as explained before. In this version transaction incarnations do not exist. Nodes are associated with transactions and directed arcs reflect transactions that are blocked. A nonblocked transaction is represented by a sink node. The blocking set of a transaction T is the set of all nonblocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T . The execution of a transaction is described as follows. A transaction T , has a site of origin which is defined as the site where the transaction enters the system. It is represented as $S_{orig}(T)$. The transaction starts running at this site, performing local operations until some nonlocal data are required. When this occurs, a lock request is formed and sent to the controller for the requested data. The controller must either accept or reject

the request and send a reply to the $S_{orig}(T)$. Each controller maintains a TWF graph at its respective site which is a subgraph of the global TWF. $TWF(k)$ represents the TWF graph at site S_k . Each $TWF(k)$ maintains wait relationships involving transactions local to S_k . Both direct and indirect waits are considered. Communication of wait information is accomplished by a transmission of "blocking pairs" as defined in the protocol below (12).

1. Rule 1: If the resource R cannot be granted to T because it is being held by transactions $T_1, T_2, \dots, T_k$, then add an arc from transaction T to each of the transactions in the set $(T_1, T_2, \dots, T_k)$. If the addition of these arcs causes a cycle to be formed in $TWF(k)$, then a deadlock is detected, and appropriate action is required for its resolution. For each transaction T' in blocking set (T) , send the blocking pair (T, T') to $S_{orig}(T)$ if $S_{orig}(T) \neq S_k$, and to $S_{orig}(T')$ if $S_{orig}(T') \neq S_k$.
2. Rule 2: When a blocking pair (T, T') is received, add an arc from T to T' in $TWF(k)$. If a cycle is formed then a deadlock exists and must be resolved by an appropriate action. If T' is blocked and $S_{orig}(T) \neq S_k$, then for each transaction T'' in the blocking set (T) , send the blocking pair (T, T'') to $S_{orig}(T'')$ if $S_{orig}(T'') \neq S_k$.

MENASCE and MUNTZ offered the following intuitive proof of the algorithm when first published. Consider the global deadlock state shown in figure 6 below.



Figure 6: Deadlock Cycle Involving Transactions $T_1, T_2, \dots, T_k$.

Initially, transactions T_1 through T_{k-1} are blocked and each of the controllers at the site of origin of these transactions has the knowledge of a single transaction ahead in the cycle. At this point the TWF at the site of origin of each transaction is shown in figure 7.

```

Sorig(T1):    T1 ----> T2
Sorig(T2):    T2 ----> T3
      :
      :
      :
Sorig(Tk-1):  Tk-1 ----> Tk
  
```

Figure 7: Initial TWF Status by Site Origin

When T_k makes a request and is blocked by T_1 , the blocking pair (T_k, T_2) will be sent to $Sorig(T_2)$ where a new blocking

pair (T_k, T_3) is formed and sent to $Sorig(T_3)$. There the blocking pair (T_k, T_4) is formed and sent to $Sorig(T_4)$ and so on until the blocking pair (T_k, T_{k-1}) reaches $Sorig(T_{k-1})$. This causes the arc from T_k to T_{k-1} to be added to the TWF graph at that site. At this point a cycle is formed and detected. This intuitive proof was presented as a worst case model, however it will be shown in the next section that it is not entirely correct and requires modification.

3.3.3 Summary.

The centralized protocol just presented has the advantage of detecting deadlock close to the sites involved and can be tuned to some degree by proper placement of controllers. The distributed protocol is better suited for an environment where site failure is not infrequent, however, this approach has been subject to the problem of false deadlock. Further research into the performance of centralized versus distributed protocols is warranted.

3.4 COUNTEREXAMPLE TO THE MENASCE/MUNTZ PROPOSAL.

Shortly after the Menasce/Muntz protocol was published, GLIGOR and SHATTUCK (4) discovered that the distributed protocol required some refinement to be truly correct and presented a counterexample to the proof presented above. Arbitrary delays in message delivery throughout a network sometimes cause out-of-order graph updates and may result in

anomalies within the detection protocol. The counterexample assumes an initial situation in which a network of three sites exist where each site, S_i ($i=1$ to 3), contains a resource R_i and a transaction T_i such that $S_{orig}(T_i)=S_i$. Each resource R_i is available and each transaction T_i is active. The system then proceeds as follows:

1. Event 1: At each site S_i , T_i requests R_i and the resource is granted for $i=1$ to 3 .
2. Event 2: T_1 requests R_2 at site S_2 , but the request is denied. Rule 1 of the Menasce/Muntz protocol is now applied.
 - a) The arc $T_1 \rightarrow T_2$ is added to the $TWF(2)$;
 - b) No deadlock is detected at S_2 ;
 - c) Since T_2 is still active, T_2 is the only transaction in the blocking set of T_1 at site S_2 . The blocking pair (T_1, T_2) is sent to site S_1 as message m_1 .
3. Event 3: T_2 requests R_3 at site S_3 , but the request is denied. Rule 1 is again invoked and,
 - a) The arc $T_2 \rightarrow T_3$ is added to the $TWF(3)$;
 - b) No deadlock is detected at S_3 ;
 - c) The blocking pair (T_2, T_3) is sent to site S_2 as a message m_2 .
4. Event 4: T_3 requests R_1 at site S_1 , but the request is denied. Rule 1 is again applied resulting in,
 - a) The arc $T_3 \rightarrow T_1$ is added to the $TWF(1)$;

- b) No deadlock is detected at site S1;
- c) The blocking set T3 at S1 contains only T1, since m1 has not yet arrived. The blocking pair (T3,T1) is sent to S3 as message m3.

5. Event 5: At this point, no deadlock is detected at any site, yet a global deadlock exists in the form $T1 \rightarrow T2 \rightarrow T3 \rightarrow T1$. Now consider messages m1,m2, and m3 arriving in any order throughout the network.

- a) $M1=\{T1,T2\}$ is received at S1. Rule 2 of the protocol is applied resulting in arc $T1 \rightarrow T2$ being added to TWF(1). No deadlock is detected at S1 and since T2 is unblocked and $Sorig(T1)=S1$, no messages are generated.
- b) The same events outlined in a above occur at S2 and S3 when messages m2 and m3 arrive - no deadlock is detected and no messages are generated.

At this point all deadlock detection activity ceases and the protocol has failed. The failure of the algorithm is due to incorrect determinations as to whether or not a transaction is blocked. This determination must be held in abeyance until the results of an outstanding request for a nonlocal resource is known. The problem that has occurred here is quite common and is an example of the communication delay anomaly mentioned in Chapter I. One cannot assume that messages transmitted by several sites simultaneously, will

arrive at their destinations in any particular order or in a timely fashion. MENASCE subsequently modified the distributed detection algorithm to provide for the extra communications required to detect deadlock in the counterexample. This modification requires the presentation of two additional definitions (4). A "potential-blocking set" of a transaction T is the set of all waiting transactions that can be reached from T . A pair (T, T') is a potential-blocking pair if T' belongs to the potential-blocking set of T . The correct protocol is now as follows (portions of the protocol originally presented and unchanged are also repeated here for the reader's convenience):

1. Rule 0: When transaction T requests a nonlocal resource it is marked as a waiting transaction.
2. Rule 1: If the resource R cannot be granted to T because it is being held by transactions $T_1, T_2, \dots, T_k$, then add an arc from transaction T to each of the transactions in the set $(T_1, T_2, \dots, T_k)$. If the addition of these arcs causes a cycle to be formed in $TWF(k)$, then a deadlock is detected and appropriate action is required for its resolution. For each transaction T' in blocking set (T) , send the blocking pair (T, T') to $Sorig(T)$ if $Sorig(T) \neq Sk$ and to $Sorig(T')$ if $Sorig(T') \neq Sk$. Form a list of potential blocking pairs associated with T .

3. Rule 2: When a blocking pair (T, T') is received, add an arc from T to T' in $TWF(k)$. If a cycle is formed, then a deadlock exists and must be resolved by appropriate action.

a) Rule 2.1: If T' is blocked and $Sorig(T) \neq Sk$, then for each transaction T'' in the blocking set (T) , send the blocking pair (T, T'') to $Sorig(T'')$ if $Sorig(T'') \neq Sk$.

b) Rule 2.2: If T is waiting and $Sorig(T) = Sk$, then for each potential blocking pair (T'', T) , send the blocking pair (T'', T') to $Sorig(T'')$ if $Sorig(T'') \neq Sk$. Then discard the potential blocking pair (T'', T) and remove the waiting mark of T .

A major disadvantage of this protocol is the large amount of communication required in the maintenance of TWF graphs at each site. When one considers the necessary actions required to remove arcs and nodes as a result of unlock requests in a system with many transactions active and waiting, the problem becomes quite complex. This complexity raises questions as to the practicality of the algorithm (4).

3.5 THE ISLOOR - MARS LAND ON-LINE DETECTION TECHNIQUE

The on-line protocol proposed by ISLCOR and MARS LAND (8) differs from the Menasce/Muntz protocol in that it maintains a complete bipartite process-resource graph depicting the entire system at each site and updates the graph after every resource allocation request. Its main purpose is to achieve a timely recognition of deadlock as it occurs. The protocol utilizes the concept of a reachable set as defined by HOLT (see Chapter II), thus in a system graph representing the interactions of processes, the set of all nodes traversed by a directed path from a given node constitutes its reachable set. If the reachable set includes a cycle, the process is deadlocked. The method is best explained through the use of a conceptual example provided in reference (8) and keyed to figure 8.

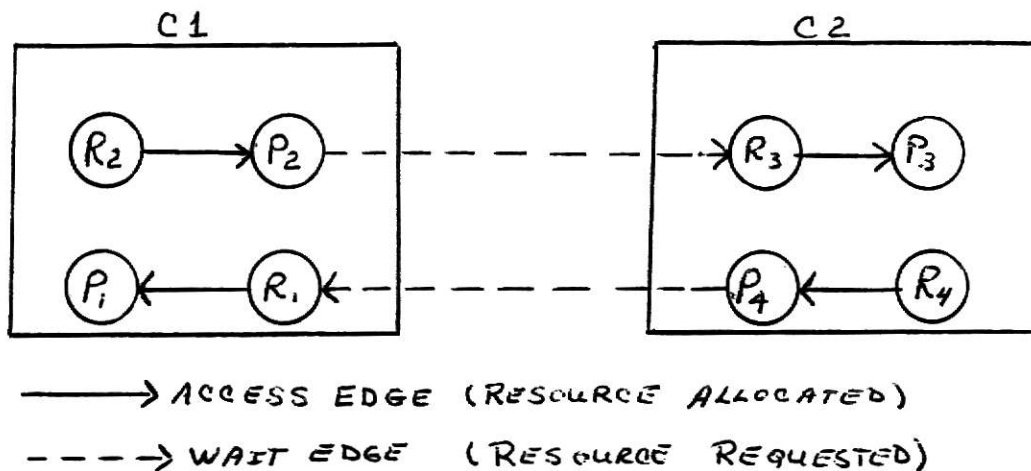


Figure 8: Isloor - Marsland Example

C1 and C2 represent separate computer sites in the network. Processes P1 and P2 and resources R1 and R2 reside on C1. Likewise, P3, P4, R3, and R4 reside on C2. For the purposes of this example, we assume an initial state where each resource R_i is allocated to process P_i , for $i=1$ to 4. Process P4 requests R1 and P2 requests R3 (both represented by wait edges in figure 8). When the allocation request is made, it is broadcast to all sites in the network. At any instant (say, t_0) both C1 and C2 possess updated reachable sets for R2 and R4 of $(P2, R3, P3)$ and $(P4, R1, P1)$ respectively. At time t_1 let P3 request R4 and P1 request R2. Locally no deadlock has occurred, but globally it has. Once the resource requests are broadcast to each site for inclusion in the reachable set deadlock will be detected. The reachable set for R2 becomes $(P2, R3, P3, R4, P4, R1, P1, R2)$ and a cycle exists. The same circumstance occurs in the reachable set of R4. Now it becomes necessary to apply a preemption mechanism to regain control of the system.

I have been unable to find any support for this technique in current literature. Although it can be demonstrated to detect deadlock, it does so at great cost (e.g., frequent broadcasts of every graph update throughout the system). This technique becomes one of maintaining a centralized detection graph in a decentralized manner. A problem the author fails to address that could occur is that of detection of deadlock at multiple sites simultaneously.

Possible solutions would be to designate priority sites for preemption responsibility based on some established criteria or the maintenance of a global preemption status table used to double check deadlock situations. This particular problem should not occur as often in the Menasce/Muntz protocol since the condensed TWF at each site is not likely to be exactly like another in the network due to message delay. The Bates model makes allowances for multiple site detection by suggesting a procedure that establishes priority for breaking deadlock depending upon a fixed ordering of the network sites.

3.6 OBERMARCK'S DEADLOCK DETECTION ALGORITHM

3.6.1 Introduction

OBERMARCK proposed a distributed deadlock detection algorithm (13) for DDBMS very similar to the Menasce/Muntz attempt but purportedly without the performance problems that their algorithm was expected to have. A transaction-wait-for graph is developed and maintained at each site, transactions may migrate from their original site and be represented at a new site by an incarnation (OBERMARCK uses the term "agent"), and transaction-wait-for information is transmitted from site to site in the form of "ordered blocked process lists". In order to represent the communication status between nodes at different sites (waiting to send or waiting to receive) a special node

called External (Ex) is created in each TWF graph. Unlike the Isloor-Marsland technique, only potential multisite deadlock information is transmitted between sites. Conceptually, agents of transactions can be viewed as nodes of a graph and their communication links become bi-directional edges (or arcs). Each site maintains a distributed deadlock detector. It can communicate with deadlock detectors at other sites which have links to its local transactions. These concepts can be demonstrated by the figures below which were used in Obermarck's explanation. Figure 9 shows a wait-for graph as seen in a centralized system.

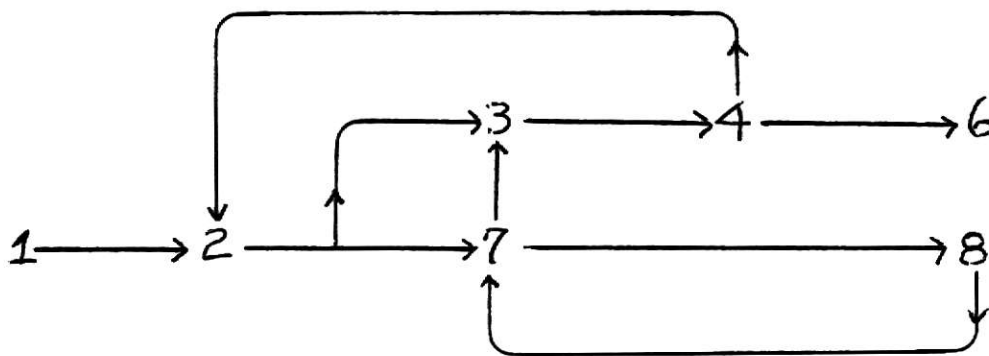


Figure 9: Wait-for graph in centralized system (OBERMARCK)

The numbers represent transactions and the directed arcs are the wait-for relations. It is apparent that three deadlock cycles exist globally (2--->3--->4--->2, 2--->7--->3--->4--->2, and 7--->8--->7). When the same situation is applied to a distributed system the graph also becomes distributed as demonstrated in figure 10.

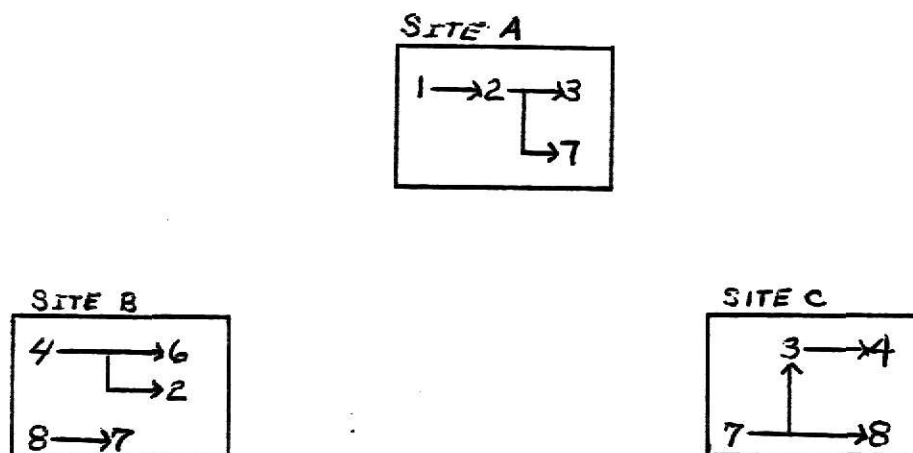


Figure 10: Obermarck's wait-for graph - distributed

Figure 10 now shows the transaction migration concept. Each local TWF fragment contains nodes with local agents. Transaction 2 originated at site B and established an agent at site A. Its agent at site A waits for some resource held by transactions 3 and 7 (this may occur if 3 and 7 have both requested shared access, but 2 requires exclusive access). Transaction 3 originated at A, migrated to C, and waits for transaction 4. This explanation can be carried forth for each transaction in the system by intuitive analysis if desired by the reader.

Figure 11 now adds in the required communication links between the nodes and makes use of the external node concept previously mentioned. Assume transaction 7 originates at site B and migrates to site C. A communications link must be established and is shown here as a directed hashed line which represents agents waiting for communication from other

agents. Transaction 7 at site B is now waiting for its agent at site C. Agent 7 at site C now requests information from site A. A communications link is established, an agent is created for 7 at site A and the requested DB information is transmitted from A back to C. Agent 7 at site A remains available (by not unlocking its resource) and awaits further communication from agent 7 at site C. Meanwhile agent 7 at site C waits for a resource held by transactions 3 and 8.

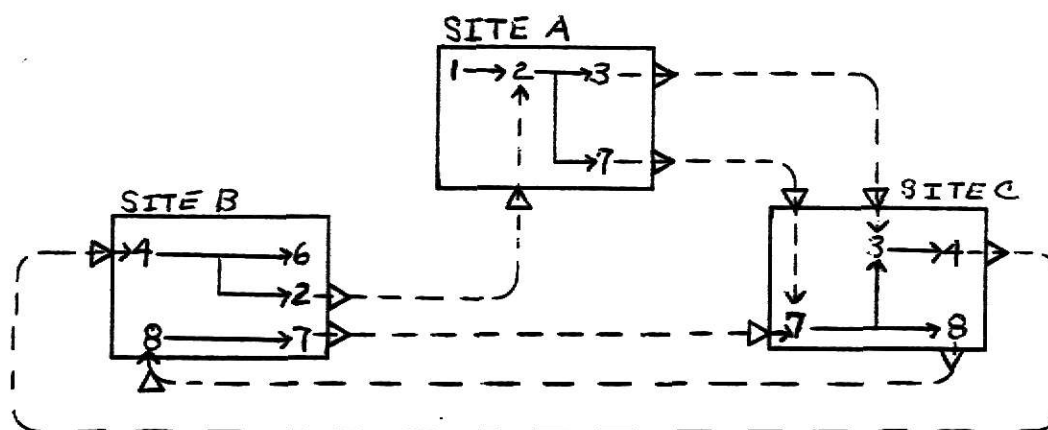


Figure 11: OBERMARCK's graph with communication links

3.6.2 The Obermarck Algorithm

With the three figures in mind from the previous section, Obermark's algorithm is now presented. The term "string" is used uniquely in this algorithm. It can be considered to be a path that is transmitted from one site to one or more other sites as a result of deadlock detection. The term "external" refers to that portion of the global TWF graph

that is unknown to the local site (see figure 11). The cycle $Ex \rightarrow Tran1 \rightarrow Tran2 \rightarrow Ex$, can be interpreted at the local site as a case in which an external transaction/agent is waiting on Tran1, which is waiting on Tran2, which requires a resource from another external transaction/agent. The absolute identification of the externals is unknown at this point. The detection algorithm at each site performs the following actions (13):

1. Build a wait-for graph using the transaction-to-transaction wait for relationships (obtainable from any resource allocation table at the site).
2. Obtain and add any strings of nodes transmitted from other sites to the existing wait-for graph.
 - a) For each transaction identified in a string create a node in the TWF graph if none exist at this site.
 - b) For each transaction in the string, starting with the first (always Ex), create an edge to the node representing the next transaction in the string.
3. Create wait-for edges from Ex to each node representing a transaction's agent that is expected to send on a communications link.
4. Create a wait-for edge from each node representing a transaction's agent, which is waiting to receive from Ex.
5. Analyze the graph and list all elementary cycles.

6. Select a "victim" (transaction or agent) to break each cycle that does not contain the node Ex. Remove all cycles from the cycle list (see step 5) that contain the victim.
 - a) The transaction ID of the victim must be remembered by site at least through the next detection cycle so any strings containing that ID can be discarded.
 - b) The identity of the victim must be transmitted to all sites known to contain an agent for the victim. As an alternative, the information could be simply transmitted to all sites that have communication links established with this site.
7. Examine each remaining cycle containing the node Ex. If the transaction identifier of the node for which Ex waits is lexically greater than the transaction identifier of the node waiting for Ex then:
 - a) Transform the cycle into a string which starts with the transaction identifier of the node Ex, followed by each transaction identifier in the cycle, ending with the transaction identifier of the node that waits for Ex.
 - b) Send the string to each site for which the transaction terminating the string is waiting to receive.

3.6.3 Summary

This algorithm is demonstrated in reference (13) using the example depicted in figure 11. Because of the lengthy explanation and numerous diagrams required to track the table/graph manipulations it will not be repeated here. The reader is invited to review it should further information be required. The algorithm offers several advantages in addition to not requiring a central control node.

1. Only potential multi-site deadlock cycle information is transmitted.
2. Because the sites are lexically ordered, information can be transmitted in a single direction along the path of the potential deadlock cycle.
3. The detection mechanism appears to be less prone to failure since it is distributed independently among sites.

3.7 THE RECOVERY PROBLEM

Several deadlock detection schemes have been presented - some centralized, some decentralized. All have one characteristic in common and that is they require the system to exercise a recovery procedure after deadlock has been detected. The subject of recovery in distributed systems is a complex issue and the topic of much current research. It is not the intent of this report to investigate the problems involved but rather to insure that the reader is aware of

the requirements. The need to recover is not just associated with deadlock. It can be required because of system crashes, user errors, or application program problems. We are primarily concerned here with the consistency of the database and the proper restart of the user program in a manner that is transparent to the user. Not all recovery processes involve larger amounts of time. If it becomes necessary to preempt a processor, recovery simply entails storing some register contents and restarting the user program at the appropriate time. On the other hand, a hardware failure occurring during an update of a distributed database, may be extremely difficult to recover from unless some journal of all transactions is maintained along with a copy of a previous safe state. A rollback and restart procedure may be successfully accomplished, but at great expense. Further readings on this subject may be found in references (1), (3), and (9).

3.8 CONCLUSION

This chapter has been rather lengthy but demonstrates a variety of thinking on the detection topic. Practically all current literature concentrates on the detection approach. One could assume from this that it is also the most practical approach for the future and will always be necessary as long as there are processes with dynamically determined resource needs. The majority of detection

algorithms depend on the maintenance of a directed graph either at all nodes or on one central node. The centralized approach seems to work better because it is a conceptually simpler procedure, the problems of false deadlock are easier to handle, and it requires less frequent communication. Disadvantages of a centralized approach include its vulnerability to failure, difficulty in implementing it from a practical standpoint, and the high cost of graph maintenance. The major advantage of a distributed protocol is its resilience in light of potential site failures. This is accomplished by accepting the higher communication costs. Additionally, one should be aware that published distributed protocols may not be totally correct and may be subject to false deadlock and/or unnecessary preemptions.

Chapter IV

AVOIDANCE AND PREVENTION OF DISTRIBUTED DEADLOCK

4.1 INTRODUCTION

Very little has been published on distributed deadlock avoidance or prevention schemes. Most likely, this is because it is easier to detect the occurrence of a deadlock after the fact, rather than trying to prevent or avoid it from occurring in the first place. This strategy may not be acceptable in all circumstances. For example, a real time system controlling the activities of a national defense system or perhaps some re-entry function of a space vehicle, cannot be allowed to deadlock. Such delays might be critical and totally unacceptable to the user. Herein lies the rationale for design and implementation of avoidance and prevention algorithms.

A fine line exists between what is actually prevention versus what is avoidance. In fact, some authors are found to use the terms interchangeably. In a general sense, a prevention algorithm is a design of the system such that deadlock may not occur. This design typically assumes one of three approaches. First, a rule may be imposed on the system that all processes must specify or acquire needed resources before the process is allowed to become actively

engaged in processing. This procedure may result in resources being held for longer periods of time than actually needed at the detriment of other processes that need the same resources. Additionally, it may be impossible to implement in systems that dynamically request resources. A second method sometimes used is that of automatic resource preemption. In this strategy blocked processes are required to release resources to active processes. This may or may not be tied to a predetermined minimum time interval. A disadvantage here is the possibility of "cyclic restart" where two processes continuously block, abort, and restart each other. A third method sometimes discussed requires a resource ordering strategy. This algorithm necessitates grouping all resources in classes that are then numbered linearly. All required resources from a lower class must be acquired before a resource from a higher class may be granted (see Chapter II). An avoidance technique requires that a resource be granted only if at least one way remains for all processes to complete execution. This involves a "look ahead" by the operating system to see if the requested allocation would result in deadlock. If it will, the allocation is delayed or denied, and if not, it is granted. Each allocation is tested and only those that lead to safe states are granted. A popular structure utilized in this technique is HABERMANN's "maximum claim model" (5).

4.2 AVOIDANCE OF DEADLOCK

A comprehensive review of current literature failed to produce any published avoidance algorithm for distributed systems, however, some writings exist on how it might be implemented (see references (2), (7), (10), and (15)). All authors in the cited references agree that some form of HABERMANN's model must be employed as either a maximum claim strategy where each process declares a maximum bound on its future requirements, or as task/step model where each process specifies its resource requirements ahead of time for each step or task that it will execute in its linear history. Naturally, the task/step model will potentially hold resources for a shorter period of time than will the maximum claims procedure. In either case, the system must be able to look ahead, know what resources are potentially going to be requested, and not allow the process to move into a potential deadlock. In other words, it must first test each resource allocation request before actually making the allocation.

RYPKA and LUCIDO (15) suggest the building and maintenance of a "claimed system resource allocation graph" which is virtually identical to the graph structure depicted at figure 8, with directed arcs between processes and resources representing requested and allocated resources. A process may claim a resource (for a given mode) before requesting it and delete the claim when the process no

longer requires the resource. Differentiation is made between a request for exclusive access (for an update) versus a request for shared access (read only mode). An example of this structure is shown in figure 12 below.

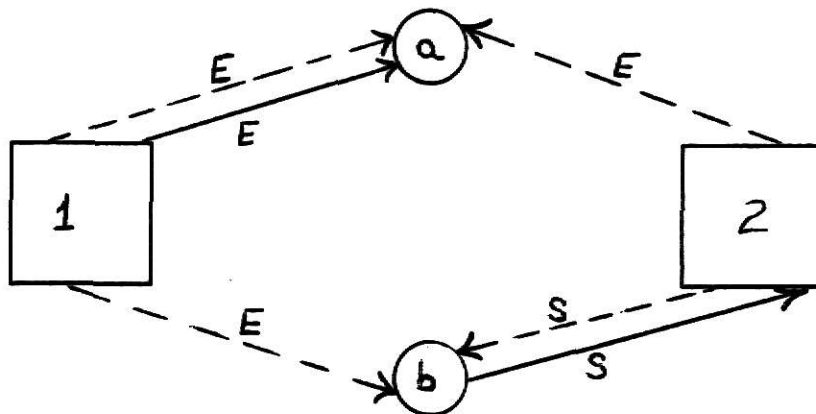


Figure 12: Example of a Claimed System Resource Allocation Graph.

In this example, process 1 has claimed resources a and b, shown by the dotted directed arc (claim edge) and has an E mode request in for a. Similarly, process 2 has claimed resources a and b and has been assigned b. Each claim edge also indicates whether the claim is for exclusive or shared access. An actual request for the resource is shown by a solid line directed to the resource which also indicates type of access desired. An assignment is shown by a solid line directed from the resource to the process. The resource allocation strategy temporarily prevents allocations that might lead to a deadlock state. A process which is not blocked but is denied a request because of this

allocation strategy is called allocation-blocked (A-blocked). In figure 12, the request by process 1 for resource A is A-blocked. Resource access is said to be mode compatible if any resource being accessed in mode A can be simultaneously accessed in mode B, otherwise they are incompatible. For example, if processes 1 and 2 both request shared access to resource a simultaneously, the requests are compatible and will be granted. If either process requests exclusive access, the requests are incompatible. A possible deadlock path (PD-path), is a path in which all claim and request links in the path are followed by incompatibly labeled assignment links. A PD-cycle is defined in the same manner. The allocation strategy then becomes one of deferring any allocation that would complete a PD-cycle. RYPKA and LUCIDO continue to develop the procedure for a traditional multiprogramming system, but if coupled with the Menasce/Muntz hierarchy of lock controllers, it might be applicable to a distributed system. This is certainly an area for further research.

DEVILLERS (2) proposed a mathematical model describing the avoidance problem with a gaming strategy where the processes competing for resources form one team that plays against the operating system. The operating system must insure that the system does not overcommit resources. The model makes many assumptions that cannot be made in a live environment, but does make allowances for dynamic resource

allocation through look ahead functions that investigate all possible strategies from any given safe state. DEVILLERS develops a flow chart model of the process execution history and considers three possible states during execution: the working (active execution) state, the transient state (movement between steps), and the terminated state. The model cannot be implemented at this point, but is worthy of mention as a fresh approach to the problem and a potential area for further investigation.

Conceptually, the avoidance problem is the same in both conventional and distributed systems, however implementation in a distributed system has yet to be accomplished. It is most likely that distributed systems will continue to choose between detection or prevention methods for dealing with deadlock in the immediate future.

4.3 PREVENTION OF DEADLOCK

4.3.1 Introduction

The only prevention algorithm applicable to distributed systems that was located in this literature review, was one proposed by MARYANSKI (11). Reference (11) contains many proofs and detailed examples of how the procedure functions. This report will restate the algorithm as presented by MARYANSKI, but will omit the many examples of its operation.

The algorithm assumes a DDBMS in a network where application programs are executed on a host machine and a

backend machine controls access to data. It is permissible for machines to be bifunctional. Furthermore, the algorithm requires records that may be shared among several processes, to be identified and their identities communicated to all backend machines that control the data items. It is the responsibility of the backend machines to prevent deadlock. The shared record list is obtained at each site from the sub-schema of a process. Each backend machine maintains a list of interacting processes. Although there are significant communication costs associated with this algorithm, the author claims that it may be more efficient because of the "uncertain and potentially serious performance degradations that may result from rollback in a distributed DBMS."

In order to present the algorithm, definitions and notations used by Maryanski must be reproduced here. The following symbols are used:

1. r_j - a record in the data base.
2. T_k - an application task.
3. R_k - a 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 all tasks whose potential shared record lists have non-empty pairwise intersections.

5. St - The shared record list of Xt ; all records appearing in more than one potential shared record list of the tasks Tk .
6. Bk - The backend processor executing a data base request for Tk .
7. St,k - The shared record list of a set of tasks t on backend processor k ; a record in a shared record list is marked with a task identifier when it is requested or locked.
8. $m(St,k)$ - The number of distinct tasks that have records marked in St,k .
9. $L(St)$ - The number of distinct tasks that have records locked in St .

For a given task interaction list, Xt , a copy of St , the shared record list, is maintained on each backend machine executing data base operations for a task in Xt .

Definition 1: A set of tasks $T=(T_1,T_2, \dots ,T_m)$, $m>1$, is deadlocked if for any i , such that i is between 1 and $m-1$, T_i is blocked by T_{i+1} and T_m is blocked by T_1 . In other words, a cycle exists.

Definition 2: A set of tasks $T=(T_1,T_2, \dots ,T_m)$, $m>1$, 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. Two lemmas result:

1. A set of tasks cannot enter a deadlock state without first entering a deadlock-prone state.
2. A set of tasks is in a deadlock state if and only if $L(St) = |T|$.

4.3.2 The Prevention Algorithm

Lemma 2 above forms the basis for the algorithm. If it can be shown that $L(St) < |T|$, then deadlock cannot exist. Three commands and one response are transmitted between the processes. The commands are LOCK, UNLOCK, and REQUEST. The one response is POSITIVE. When exclusive access is required of a resource, a process will issue either a LOCK or REQUEST command to the backend machine responsible for the resource. The decision whether to issue LOCK or REQUEST is a function of priority - issue a LOCK to for lower priority tasks and REQUEST for higher priority tasks. UNLOCK relinquishes control of a resource. Two conditions must be satisfied before a backend processor can declare that a record is available - first, it must not be claimed by another task (marked), and secondly, that granting control of this record will not cause any set of tasks to enter a deadlock-prone state. The LOCK command marks a record for a particular task and the UNLOCK removes the mark. With the above in mind, the algorithm is now presented in a step by step sequence:

1. When task Tk desires to update a shared record rj, the following steps must be taken by Bk to prevent a deadlock state.
 - a) Check if rj is marked in any St,k containing rj. If so, Tk must wait until Bk receives an UNLOCK rj command (if rj is marked in one St,k, it is marked in all St,k).
 - b) If there exists an St,k, such that $m(St,k) = |Xt| - 1$, then Tk must wait until a record in St,k is unlocked.
 - c) Mark rj with the identifier of Tk in all St,k containing rj.
 - d) For all higher priority tasks in any St containing Tk, issue a REQUEST rj,Tk command to their backend processors.
 - e) Wait for a POSITIVE rj response from all backends of step 1d.
 - f) If while waiting, Bk receives a LOCK rj,Ti command, then Bk must issue UNLOCK rj commands to all backends which have transmitted POSITIVE rj responses and then Bk must return to step 1a above.
 - g) If while waiting Bk receives a LOCK rn,Ti command ($rn \neq rj$), and $m(St,k) = |Xt| - 1$, then Bk must issue UNLOCK rj commands to all backends which have transmitted POSITIVE rj responses and return to step 1b.

- h) When Bk receives POSITIVE rj responses from all tasks in step 1d, it issues a LOCK rj,Tk command to all lower priority tasks in any Xt containing Tk.
 - i) Tk may then operate on rj. Upon completion of the operation, Bk issues an UNLOCK rj command to all tasks in any Xt containing Tk.
2. When a backend processor, Bi, receives a REQUEST rj,Tk command, it transmits a POSITIVE rj response if all the requirements of definition 2 are met. If a POSITIVE response is transmitted, rj is marked with the identifier of Tk in all St,i.
 3. When a backend processor, Bi, receives a LOCK rn,Tk command and does not have a REQUEST command outstanding such that the conditions in step 1f or g above arise, rn is marked with the identifier of Tk in all St,i.

This completes the algorithm. MARYANSKI goes on to demonstrate the procedure in a step by step example which is much too lengthy to present here, however the reader is encouraged to follow it through by reading pages 17 to 22 of reference [11]. This technique is an excellent application of the dynamic preclaiming strategy. Although no performance tests have been run, it would be an interesting topic to research the efficiency of this algorithm. The author claims that in a DDBMS environment with frequent

deadlock, the prevention algorithm is a more satisfactory approach over the detection techniques.

Chapter V

CONCLUSION

It is abundantly clear after reviewing many published works concerning the distributed system deadlock problem that no single solution exists today that is universally acceptable. Each technique (detection, prevention, or avoidance) has its own advantages and must be weighed against the actual requirements and architecture of the system. A summary of advantages and disadvantages is presented below:

1. DETECTION: This technique's advantages are that it suits the distributed data system very well, does not delay process initiation, can be invoked with user defined frequency, and is required in any system that permits dynamically determined resource needs. Its disadvantages include cost of recovery, cost of graph maintenance in terms of computation and communication, and potential waste of resources while deadlock goes undetected.
2. AVOIDANCE: Advantages in using the avoidance approach are that minimal, or no, rollback and restart is necessary, and no preemption of resources occur. Its disadvantages are primarily that no

algorithm currently exists for implementation in a distributed system, it is extremely difficult to adapt it to a distributed system, and it requires an advance announcement of resources required by the processes.

3. PREVENTION: Advantages include no overhead for detection checking and no rollback-restart caused by deadlock. Its disadvantages are that it is difficult to implement in a distributed environment, is costly in terms of computation and communication, and few algorithms exist.

More than one author has suggested that the problem of deadlock might be better solved through the use of a mixed solution. This approach has significant merit when one considers the "layer" development of modern software. Perhaps the best approach would be to implement a different technique to handle deadlock at each layer depending on which would be the most advantageous. For example, HOWARD (7) advances the argument that control of swapping space might be best preallocated so that deadlock prevention is employed, and avoidance might easily be used in allocating hardware resources since these are normally identified prior to process execution (through job control statements or program heading information). One could extend this argument to the employment of detection when concerned with allocation of data base resources. Further research is called for in the mixed solution approach.

The detection algorithms appear to be the most readily available and best documented. Of these, the centralized protocols appear to be the best choice for current systems even though the risk of site failure impacts the detection mechanism. In a system where deadlock is frequently experienced, the Maryanski prevention algorithm would be worth investigation and possible implementation although no performance figures are available for comparison with other approaches. One would think that prevention costs might be less than those of frequent rollbacks and restarts.

Future directions for research in this area should include performance measurements between techniques; optimal communications procedures to lessen the burden of graph updates and maintenance; investigation of imbedding different techniques of deadlock handling approaches at various layers of operating system software; and the development of highly efficient rollback and restart procedures.

BIBLIOGRAPHY

1. Bates, Rodney M. "A Recovery Architecture for Distributed Transaction Processing." Unpublished paper. Feb. 1978. 23 pp.
2. Devillers, R. "Game Interpretation of the Deadlock Avoidance Problem." Communications of the ACM. Vol. 20, No. 10, Oct. 1977, pp. 741-745.
3. Draffan, I. W. and Poole, F. Distributed Data Bases - An Advanced Course. Cambridge: Cambridge University Press, 1980.
4. Gilgor, Virgil D. and Shattuck, Susan H. "On Deadlock Detection in Distributed Systems." IEEE Trans on Software Engineering. Vol. SE-6, No. 5, Sept. 1980, pp. 435-440.
5. Habermann, A. N. "Prevention of System Deadlocks." Communications of the ACM. Vol. 12, No. 7, July 1969, pp. 373-385.
6. Holt, Richard C. "Some Deadlock Properties of Computer Systems." Computing Surveys. Vol. 4, No. 3, Sept. 1972, pp. 179-196.
7. Howard, John H. Jr. "Mixed Solutions for the Deadlock Problem." Communications of the ACM. Vol. 16, No. 7, July 1973, pp. 427-430.
8. Isloor, S. S. and Marsland, T. A. "The Deadlock Problem: An Overview." Computer. Sept. 1980, pp. 58-78.
9. Kohler, Walter H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems." Computing Surveys. Vol. 13, No. 2, June 1981, pp. 149-183.
10. Lomet, David B. "Subsystems of Processes with Deadlock Avoidance." IEEE Trans on Software Engineering. Vol. SE-6, No. 3, May 1980, pp. 297-304.
11. Maryanski, Fred J. "A Deadlock Prevention Algorithm for Distributed Data Base Management Systems." Technical Report CS77-02, Computer Science Dept., Kansas State University, Manhattan, Ks., Feb. 1977, 24 pp.

12. Menasce, Daniel A. and Muntz, Richard R. "Locking and Deadlock Detection in Distributed Data Bases." IEEE Trans on Software Engineering. Vol. SE-5, No. 3, May 1979, pp. 195-202.
13. Obermarck, Ron. "Distributed Deadlock Detection Algorithm." ACM Trans on Database Systems. Vol. 7, No. 2, June 1982, pp. 187-208.
14. Ries, Daniel R. and Smith, Gordon C. "Nested Transactions in Distributed Systems." IEEE Trans on Software Engineering. Vol. SE-8, No. 3, May 1982, pp. 167-172.
15. Rypka, David J. and Lucido, Anthony P. "Deadlock Detection and Avoidance for Shared Logical Resources." IEEE Trans on Software Engineering. Vol. SE-5, No. 5, Sept. 1979, pp. 465-471.

PREVENTION AND DETECTION OF DEADLOCK IN DISTRIBUTED SYSTEMS
-- A SURVEY OF CURRENT LITERATURE

by

RAYFORD B. VAUGHN JR.

B.S., UNIVERSITY OF SOUTHERN MISSISSIPPI, 1969

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

-1982-

ABSTRACT

Since the first notion of multiprogramming was implemented, the data processing community has been forced to consider the issue of "deadlock" in terms of its prevention, avoidance, or detection. With the advent of distributed system technology, this problem has taken on added dimension in that one must now consider the deadlock problem in an environment that has no globally shared memory and is subject to significant communications delays. In general terms, deadlock occurs when two or more processes each hold some resource in the system and are blocked from proceeding until they receive another resource that is already assigned to one of the other blocked processes.

This report examines current literature published on the subject of deadlock in distributed systems. It contains a tutorial introduction to the problem and a comprehensive explanation of terms and definitions commonly found in the literature. The report examines the deadlock problem in two major areas. First, algorithms and techniques proposed for the detection of deadlock in a distributed environment are outlined, followed by a description of deadlock avoidance and prevention techniques. Problems, advantages, and disadvantages of each technique are provided, along with recommendations for areas of further study.