IMPLEMENTING DEADLOCK DETECTION IN DISTRIBUTED PROCESSING

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR

THE DEGREE OF MASTER OF SCIENCE

BY

KWABENA BIMPONG-BOTA

DEPARTMENT OF MATHEMATICAL AND COMPUTER SCIENCES

ATLANTA, GEORGIA

DECEMBER 1985

DEDICATION

This work is dedicated to Veronica and to Akosua, who, through no fault of theirs, had to endure the hardship of my absence while I pursued my studies in the United States. To them I say, "Kafra, kafra."

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Nazir Warsi, for suggesting this topic and for overseeing its' successful completion to the fullest extent possible. I appreciate very much his profound interest in the whole project. Thanks also go to the Chairperson of the Department of Mathematical and Computer Sciences, Dr. Benjamin J. Martin, for establishing a most serene and academic atmosphere in the department to stimulate us, the students, to pursue our courses to our satisfaction. To Dr. Bennett Setzer a big thank you is due for the various engaging courses he taught me. His patience and time for grading all those papers promptly was most admired and appreciated. I also want to thank all those classmates, most of whom have long completed their programs, for their moral support and encouragement.

This list of thank you's will not be complete without thanking the United States Department of Energy as well as the National Science Foundation for providing the scholarship for this program. The Department of Physics of the University, Professor Ronald Mickens, and all the academic staff, deserve special commendation for serving as the channel through which all these funds eventually reached me.

Lastly, I thank my brother Kofi Bota and sister-in-law, Marian, for accommodating me these past three years, while I enriched my academic portfolio. To my dear nephew, Master Danqua, a big thanks for providing a belly full of laughter at times of low tide.

ABSTRACT

BIMPONG-BOTA, KWABENA M.S., ATLANTA UNIVERSITY, 1984

IMPLEMENTING DEADLOCK DETECTION IN DISTRIBUTED PROCESSING

Thesis Advisor: Professor Nazir Warsi

Thesis dated: December 1985

Many protocols have been published on the topic of deadlock detection in distributed processing. A survey has been made on four of these varied schemes. The method of "distributed locking and distributed deadlock detection protocol" has been selected for implementation.

Data structures and various routines for this particular protocol have been fully developed, and the resulting program exhaustively tested. In all test cases, results were positive and based on these results, we believe we have succeeded in implementing a deadlock detecting scheme for distributed processing.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Introduction to Deadlock Detection

Many protocols have been published for deadlock detection in distributed systems in the recent past, mostly because of the renewed interest in distributed processing. With most of these published protocols, implementation is almost impossible simply because first, the communication system, as we know it today, is not able to handle the volume of traffic fast enough, thus resulting in late updates of multi-site, geographically, spatially connected system data. Second, most of the algorithms are such that implementation will involve using complex data structures which will be difficult to tract.

Purpose of the Thesis

The purpose of this thesis is to select, among these varied and known protocols, an algorithm which can be implemented in terms of simplicity in data structures and less complexity in specification. The ultimate goal of this work is to have a deadlock detection scheme for distributed systems available and pending future improvements in communication technology. A side goal is to have a deadlock detecting scheme for a local network with various nodes which need to work together. An example of usage of this scheme will be a network of all local banking institutions in metropolitan Atlanta.

Major Conclusion

In this work, we have made a survey of some of the available protocols for detecting deadlock in distributed systems. In Chapter Two, the centralized approach, the decentralized approach, the hierarchical approach, and the distributed locking and distributed deadlock detection protocols are discussed. In all these schemes, the major disadvantage was the question of communication delays. Apart from this phenomenon, it is found that some of these protocols are more complex than others in terms of protocol definition and data structure representation. We selected among these the distributed locking and deadlock protocol.

In Chapter Three, the data structures for the selected protocol are fully discussed and various routines are developed. The resulting program has been tested with four different sets of chain of events, with varying degrees of complexity. In each test case, the outcome was positive. In view of the success of these tests, we believe we have succeeded in the pursuit of our objective as outlined in the Purpose of the Thesis section.

CHAPTER TWO

SURVEY OF DEADLOCK DETECTION SCHEMES FOR DISTRIBUTED SYSTEMS

Introduction and Definition of Deadlock

Deadlock is a state of a connected system as follows: "A set of processes is in deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set."[1]

Necessary Conditions for a Deadlock State

The necessary conditions giving rise to a deadlock is described in the literature. Basically, there are two methods for dealing with the deadlock state. A protocol could be used to ensure that the system never enters a deadlock state; alternatively, the system is allowed to enter a deadlock state and then try to recover. In this thesis, we pursue the latter protocol which involves, first, a deadlock detection scheme, followed by a recovery scheme. We shall not explore the recovery scheme to any detail.

Deadlock Detection

To detect a deadlock state, an algorithm must be invoked periodically or continuously to determine whether a deadlock has occurred.

---

[1]J. Peterson, and A. Silberschatz, Operating System Concepts (New York: Addison Wesley Publishing, 1983), p. 260.

In order to do so the system must:

a. update and maintain information about transactions;

b. provide an algorithm that uses the above information to determine whether the system has entered a deadlock state.[2]

In this chapter we describe a few of the deadlock detection schemes that have been proposed in the literature. We place emphasis on the Gligor-Shattuck-Menasce-Muntz protocol, which we propose to implement.

The backbone of all the schemes is the construction and maintenance of a Wait-for (WF) graph, which is a condensed form of the traditional resource allocation graph. Figure 1 shows the difference between a resource allocation graph and an equivalent Wait-for graph.

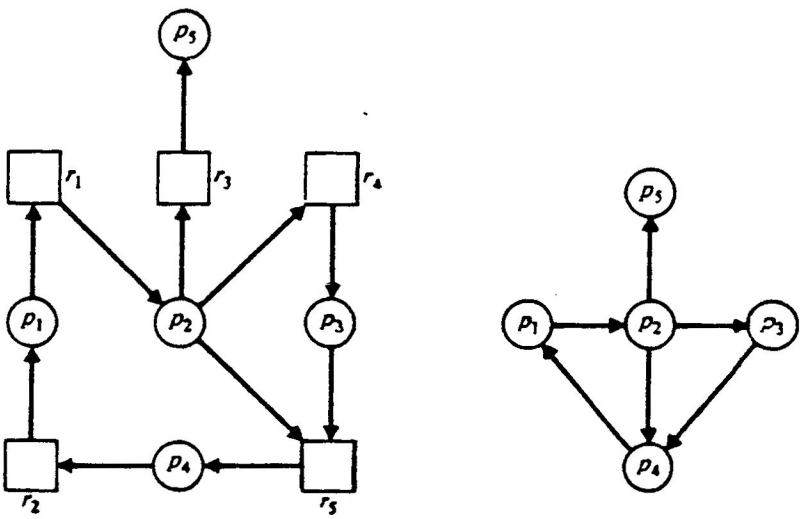

Fig. 1. Resource allocation graph and equivalent Wait-for graph.

[2]Ibid., p. 260.

Deadlock Problem Statement and Assumptions

Assumption

Assume only a single resource of each type.

Deadlock Problem

A. Each site (Si) has a resource (Ri).

B. Processes/transactions originating at any site (Si) request a resource (Ri). In general many processes may have been granted the right to use the resource (Ri).

C. Two types of deadlocks can occur:

1. Local deadlock at a site: Transactions T1, T2,...Tn have been granted permission for Ri at Si. If T requests the use of Ri and T holds a resource (say R), which is required by any Ti to complete its duty, then, there is a deadlock.

2. Global deadlock involving two or more sites: Suppose T1 has a resource R1 and requests R2 which is held by T2. Now T2 requests R1. Then this is a global deadlock involving S1 and S2.

Potential deadlocks arise when a transaction is blocked.[3] To detect any deadlock, a WF graph is created at each site. In general a cycle in a WF graph suggests a deadlock state, but not all cycles result in real deadlocks. Sometimes, there are false deadlocks, therefore, a decision as to whether a deadlock is real or not must be built into the protocol for deadlock detection.

---

[3]D. Menasce, and R. Muntz, "Locking and Deadlock Detection in Distributed Data Base," in IEEE Transaction on Software Engineering vol. SE-5 (May, 1979), pp. 195-202.

There are a number of different methods for organizing the WF graph. This fact has given rise to a number of different approaches to tackle deadlock detection problems. A few of these available protocols are described in the following sections.

Centralized Approach to Deadlock Detection

Description

In the centralized approach, a global WF graph is constructed and maintained by a single deadlock detector/coordinator. The central deadlock coordinator builds the global WF graph for information received from all sites in the network. Whenever a transaction (T) either inserts or removes an edge in its local graph, it must notify the detector of this modification. Upon receipt of such a message, the detector updates its global graph. Figure 2 exemplifies two local networks with a corresponding view from the central global graph.

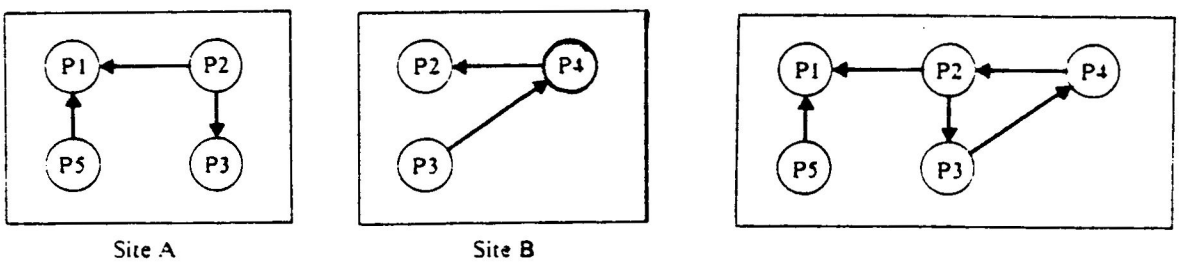

Fig. 2. Local Wait-for graph with corresponding global Wait-for graph.

Invocation of Deadlock Detecting Algorithm

When the deadlock detecting algorithm is invoked, the detector searches its global graph. If a cycle is found, a victim is selected and

rolled back. The detector notifies all the sites that a particular transaction has been selected as a victim. The sites, in turn, roll back the victimized transaction.

Problems with the Centralized Approach

The first problem with the centralized scheme is the unnecessary roll backs resulting as a consequence of two situations:

a. false cycles may exist in the global WF graph;

b. unnecessary roll backs resulting when a deadlock has indeed occurred and a victim has been picked, while at the same time one of the processes was aborted for reasons unrelated to the deadlock (such as a transaction exceeding its allocated time quantum).

Secondly, even though the centralized method may be practical and efficient for local networks, high cost factors, arising from the interconnection of fairly large spatially connected network systems, does go against it. For example, we certainly do not want deadlocks which involve only resources located at southern California to be detected at the East Coast.

Thirdly, we have the problem of robustness (the vulnerability of the central site to various types of hardware failure) and thus an additional overhead associated with having to monitor hardware failures.

Decentralized Approach to Deadlock Detection

Description

In the "decentralized approach" to deadlock detection, there does

not exist a central detector/coordinator. All nodes in the network share in the responsibility for detecting deadlocks.[4]

An important component for the decentralized scheme of deadlock detection is the process management module (PMM) which is located at each site. All resource requests and releases go through the PMM.

When a PMM finds out that a local resource can not be currently allocated to a process that is requesting it, it initiates and creates an ordered blocked process list (OBPL) with a process entry for the blocked process. This list is built up or expanded until:

1. a deadlock is detected;
2. it is ascertained that there is no deadlock; or
3. it is determined that there is not enough information to continue expansion.

The detection of deadlock amounts to a repetition of a process name in an OBPL. If no deadlock is detected, then the OBPL is aborted.

Problems with the Decentralized Approach

A major drawback to Goldman's protocol is the factor involved in repeated construction of the OBPL, some of which are eventually aborted. Secondly, abortions of OBPLs are sources of false deadlock detection; the reason for false deadlocks is that abortions at nonlocal sites might not be reflected soon enough in other copies of OBPLs. Thirdly, even though repeated checking of the last process status minimizes false deadlocks, multiple copies of an OBPL is an undesirable waste of resources.

---

[4]B. Goldman, Deadlock Detection in Computer Networks (Cambridge: MIT Cambridge Technical Rep. MIT-LCS TR-185, [1977]), p.

Hierarchical Approach to Deadlock Detection

General Description

The hierarchical deadlock detection algorithm is a distributed algorithm. Just like the centralized and decentralized approaches, each site maintains its own local graph to take care of local transactions involving local resources. In contrast, however, the WF graph is distributed over a number of detectors or controllers. These controllers are organized into a tree, where each leaf controller is assigned the responsibility of overseeing each local subdata base WF graph. Each nonleaf controller maintains a WF graph which contains relevant information from the graphs of the controllers in the subtree below it.

Symbolic Description of Hierarchical Approach

The whole system, say database system, (DBS) is viewed as the union of a set of subsystems DBSi's such that DBSi<>DBSj for j<>i. The hierarchy is such that at the bottomost level we have the leaves of a tree called Leaf Controllers (LKs) and Nonleaf Controllers (NLKs).

1. Every subdata base DBSi is assigned a leaf controller (LKi) which maintains a WF graph WF(LKi). This graph contains all the nodes of the global WF associated with a transaction incarnation (a collection of resources acting on behalf of a transaction). Leaf controllers have additional nodes called input-ports and output-ports. Input-ports are associated with incident directed arcs and output-ports are associated with outgoing arcs. They are denoted by I(LKi,T), and O(LKi,T) respectively.

2. Nonleaf controllers maintain a graph called an input-output-ports (IOP) graph. Nodes of an IOP are associated with input (i-node) ports and output (o-nodes) ports of leaf controllers. A hierarchical WF graph is shown in Figure 3.

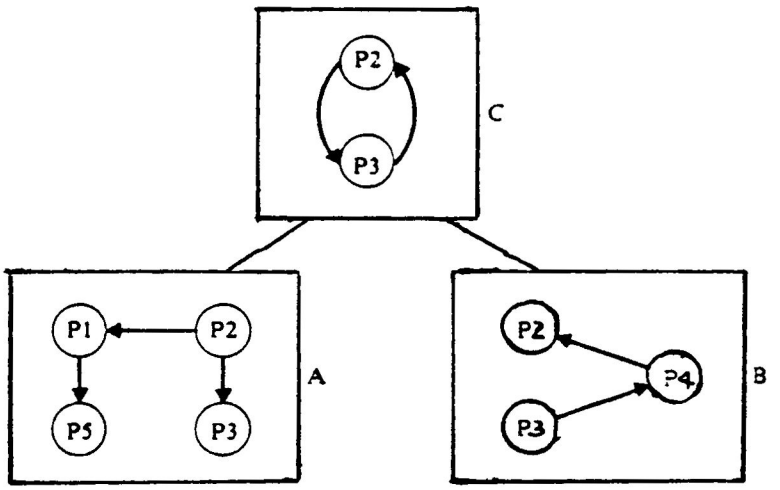

Fig. 3. Hierarchical Wait-for graph.

In Figure 3, A, B and C are controllers such that C is a nonleaf, lowest common ancestor of A and B which are leaves.

Protocol Description

Suppose that node Ti appears in the local WF graph of controllers A and B. Then Ti must also appear in the local WF graph of:

1. controller C
2. every controller in the path from C to A
3. every controller in the path from C to B.

Additionally, if Ti and Tj appear in the WF graph of controller D, and

there is a path from Ti to Tj in the WF graphs of one of the sons of D, then an edge <Ti,Tj> must be in the WF graph of D.

Properties of Hierarchical Protocol

1. Deadlocks which involve resources of a single subdata base DBSi are detected by the formation of a cycle in the WF graph of the leaf controller associated to DBSi.
2. Deadlocks which involve resources controlled by the leaf controllers (i.e., resources spanning different controllers) are detected by the formation of a cycle in the IOP graph of the nonleaf controller, which is the lowest common ancestor of the leaf controllers involved.

Reflection of Lock Releases in Graphs

An unlock operation will cause an arc (may be more) to be deleted from a WF graph of a leaf controller. All the i-o paths (if any) which contained this arc are broken. A report is made to the father of the LK. There, other arcs are broken in the nonleaf controllers which send messages to their fathers. This propagation continues up in the hierarchy until the deletion of an arc from a graph does not cause any i-o path to be broken.

The above description requires continuous updates of nonleaf controllers. A variation will be a mixture of continuous and periodic updates, depending on the volume of transactions. A varied approach can be used when appropriate. As an example, the information concerning connections between input- and output-ports can be sent periodically. This strategy would reduce the amount of traffic generated but is likely to result in a deadlock existing for far too long a time. One other method intermedi-

ate to continuous and periodic deadlock detection is to report connnections between input- and output-ports after they have persisted longer than some threshold. A careful choice of this threshold can result in less traffic being generated than with periodic checking.

Deadlock Resolution

Whenever an i-o arc is received by an NLK, the name of the controller which generated the arc will be stored with the arc. Then, when an NLK detects a deadlock cycle, it can send down the tree to its appropriate sons a message which will continue to propagate down (through the appropriate son) until it reaches the leaves of the tree. At this point the LKs can report directly to the NLK, which detected the deadlock, all the necessary information to implement the desired policy of deadlock resolution. The criteria involved in optimal transaction victimization is a policy issue and will be pursued in this thesis.

An alternative, which might result in a nonoptimal choice of victim, is to select the transaction to be pre-empted from those which appear in the IOP graph only; in this instance no additional messages are propagated.

Hierarchy Establishment

The performance of the hierarchical protocol, in terms of the overhead of message traffic cost, can be minimized if the hierarchy is appropriately chosen. Given a set of leaf controllers assigned to the nodes of a computer network, given the DBS traffic pattern, and given the cost of sending messages between every pair of nodes in the network, the problem becomes finding a hierarchy which minimizes the total cost incurred

in using the protocol. The general optimization problem is the subject of current research effort.

Problems with Hierarchical Protocol

Though the concept involved in this scheme is simple, the problem of keeping tract of parts of a graph at different places is a nontrivial task. Communication limitations are also a problem.

Distributed Locking and Deadlock Approach[5-6]

Description

This approach differs from the already described approaches in the following way: There exists a distributed control mechanism as well as a distributed deadlock detection mechanism.

Fully distributed protocol for deadlock detection requires that each site of the distributed system construct and maintain a graph for deadlock detection. To ensure that each site maintains as accurate a view of the process and resource status as possible, per-site concurrency controller exchange messages that cause additions and deletions of graph nodes and arcs.

Definitions for Protocol

To make the description of this protocol more meaningful, some definition of terms are in order.

---

[5]D. Menasce, and R. Muntz, "Locking and Deadlock Detection in Distributed Data Base," in IEEE Transaction on Software Engineering, vol. SE-5 (May, 1979), pp. 195-202.

[6]V.D. Gligor, and S.H. Shattuck, "On Deadlock Detection in Distributed Systems, in IEEE Transaction on Software Engineering vol. SE-6, 5 (September, 1980), pp. 435-39.

Controllers - a deadlock detection mechanism associated with a resource; it also builds and maintains a WF graph.

Transaction (T) - a sequence of actions which can be either read, write, lock, unlock operations; transactions request resources by sending requests to the controller of the resource.

Blocking Set(T) [BS(T)] - the set of all unblocked transactions which can be reached by following a directed path in the WF graph starting at the node associated with transaction T. This is the set of transactions which are ultimately blocking transaction T.

Potential Blocking Set(T) [PBS(T)] - the set of waiting transactions reachable from T.

Site-of-origin [OS(T)] - node at which T is started.

Waiting Transaction [W(T)] - a transaction T requesting a nonlocal resource.

Transaction Execution

Transaction execution can be described as follows. A transaction starts at a site of origin, where it enters the system. The transaction starts running at this site, performing local operations until nonlocal data are necessary. Then a lock request in the appropriate mode is built and sent to the controller for the requested resource. This controller will either accept or reject the lock, sending the reply to the site of origin of the transaction. If there are multiple copies of data,

lock requests have to be sent to all controllers which keep a copy of the data.

Distributed Protocol Description

All controller's decisions to accept or reject lock requests are based solely on information kept at the controller's site. For detection of deadlocks, a WF graph is maintained at each site by the controllers. The following definitions are relevant to the protocol description that follows:

1. A directed arc (T', T") in WF graph at site S for resource R indicates T' is blocked by T".
2. Unblocked T is a vertex with outdegree zero - i.e., a node with no outgoing arcs.
3. The blocking set of T, BS(T)= T'|T' is unblocked from T .
4. The site-of-origin (T) is denoted by OS(T) and WF(R) is the WF graph for R at its site.
5. A transaction T requesting a nonlocal resources is a waiting transaction.
6. Potential-blocking set for T - PBS(T)= T'|T' is waiting and is reachable from T .
7. S(R) denotes site of R.

Protocol Rules

Assume T1, T2,...Tk hold R and T requests R

Rule 0: T is marked WAITING if R is a nonlocal resource

Rule 1: (1) Add an arc (T,Ti) for i=1,2,...,k in WF(R)

(2) If cycle detected in WF(R) then deadlock detected. Take appropriate measure.

(3) For each T' belonging to BS(T) send (T,T') to OS(T) if OS(T)<>S(R)

(4) For each T' belonging to BS(T) send (T,T') to OS(T') if OS(T')<>S(R)

(5) Create PBS(T)

Rule 2: (1) For each pair (T,T') received at S(R), add arcs (T,T') to WF(R)

(2) If cycle is detected then deadlock has occurred; take appropriate measure.

(3) If T' is blocked and OS(T)<>S(R) then for each T" belonging to BS(T) at this site (R), send (T,T") if OS(T")<>S(R).

(4) If T is waiting and OS(T)=S(R) then for each T" belonging to PBS(T) send (T",T') to OS(T")<>S(R)

(5) Discard potential blocking pair (T,T") and mark T nonwaiting.

Problems

This approach of deadlock suffers from delays in message delivery just like in the other approaches of deadlock detection. A second problem is that the "condensed" form of the WF graph does not make it possible to reflect, adequately, the relative positions of waiting transactions. Third, if a transaction in the middle of a chain requests a release of a resource, for example through abortion or through victimization in a deadlock recovery scheme, a cascading effect of arc changes generate a flood of messages that can severely impair the performance of the whole system.

CHAPTER THREE

IMPLEMENTATION OF A DEADLOCK DETECTION SCHEME

Gligor-Shattuck-Menasce-Muntz Protocol

The modified Menasce-Muntz approach, proposed by Gligor-Shattuck and outlined in the last section of Chapter Two is a good candidate for implementation of deadlock detection. Even though it has a problem with data flow in the communication system, which limitation is shared by all discussed protocols, we feel it has the potential of being developed into an excellent deadlock detecting protocol. The specification of this particular protocol is such that most of its demands can easily be simulated on the digital computer. We also feel that it will provide a neat method for deadlock detection pending future improvements in data transmission technology.

The method is chosen because of its neat data structure representation.

Discussion of Data Structures

Resources

Every resource in the network has a definite site at which it is located; when a transaction requests a resource, the resource could be locked by another transaction or it could be available for locking by the requesting transaction. We therefore select a record representation for resources with three attributes: SITE, LOCK, and LOCKEDBY.

Transactions

Similar to resources, every transaction has a SITE. When a transaction requests for a nonlocal resource it is marked as WAITING. A transaction is blocked and must wait for another transaction to release a lock; this condition requires that the OUTDEGREE of the requesting transaction be imcremented by one and the INDEGREE of the waited for transaction likewise incremented by one. If it happens that the transaction being waited for is nonblocked then a BLOCKINGSET must be built for the requesting transaction. On the otherhand, if the transaction being waited for is itself in a waiting state then a POTENTIAL-BLOCKINGSET must be built for the requesting transaction. Associated with the blockingset and potential-blockingset are counters which we call BLOCKSETCOUNT and PBLOCKSETCOUNT. All transactions that wait for another transaction to release a lock must be maintained as a set which we call INNODES. Lastly, since the whole process of transaction actions are dynamic we need a pointer, LINK, to show the position of a transaction being waited for. A record structure is selected to represent all these varied attributes of a transaction.

Blockingset and Potential-Blockingset

Sets are conceptually easy to deal with; however, because their manipulations are hard to physically output, we choose arrays as the structure for implementation purposes. Dynamic arrays could have been chosen, but it will merely have increased the complexity of the overall data structure of the algorithm.

Buffer

In the algorithm for deadlock detection, messages are normally sent over to different sites. There are physical constraints in the real system, for example, delays in communication. To simulate this constraint, a general purpose (that is global) storage is introduced by way of what we call BUFFER. The first field, PENDING, indicates the fact that the contents of that entry as a set is not transmitted yet. The fields NODE1 and NODE2 stand for a pair that need to be transmitted to a designated site stored in the SITE field.

Wait-for (WF) Graph

The Wait-for graph at a site is structured as an array of transactions. Every transaction in the WF graph is designed as a headnode. Through this headnode, a field, LINK, points to a chain of transactions connected to this headnode. Thus, the overall WF graph at a site is an adjacency linked list of records.

Pascal Language Implementation of Algorithm

The deadlock detecting algorithm is implemented with the following list of routines. The routines are fully documented in the program listing under Appendix A, however, we give a brief description of what they do.

Procedures and Functions Used

INITIALIZE - For establishing the layout of resources and transactions in the system.

REQUEST - This is invoked when a transaction requests a resource. A side effect is triggering of the state GRANTED to be true or not.

ADDARC - Used in adding a pair of transactions <T1, T2> to the WF graph at a site.

DEADLDETECT - When invoked, this routine checks to see if a deadlock state exists at a given site.

CREATEBLOCKSET - Used in adding a member to the blockingset of a transaction.

BUFFERTRIPLE - This routine adds a triple <T1, T2, SITE> to the Buffer. At the same time the state of the entry is set as PENDING, meaning that it is unused.

SENDPAIR - This routine works on the blockingset of a transaction. It makes the choice of the triple to be sent over to the buffer through invocation of the BUFFER-TRIPLE routine.

CREATEPBLOCKSET - Used in building potential-blockingset of a transaction.

RECEIVE - Used in fetching the contents of the BUFFER. Data fetched is sent to appropriate site.

DEADLDETECT is invoked and if not true other actions like checking waiting state of transactions, deleting potential blocking pairs and removal of waiting marks are performed.

MANIPULATE - This routine invokes other routines when a resource is not granted to a transaction.

SHOWGRAPH - Displays WF graph at a given site.

SHOWMESSAGES - Displays the contents of the buffer any time it is invoked.

Performance Test on Algorithm

The deadlock detecting program has been tested with varied sets of chain of events. For all the test cases, resources Ri and R(i+10), and Transactions Ti and T(i+10), are preassigned to site Si, where i starts from 1 and ends at 10. Full documentation of the test runs are listed under Appendix B.

Test Sample I

Small scale initial local request are made followed by nonlocal requests of resources. The sequence of events are as follows:

1: Transactions Ti request for Ri; (1 = < i < = 3)

2: T1 requests for R2

3: T2 requests for R3

4: T3 requests for R1

A graphical representation of this chain of events is shown in Figure 4.

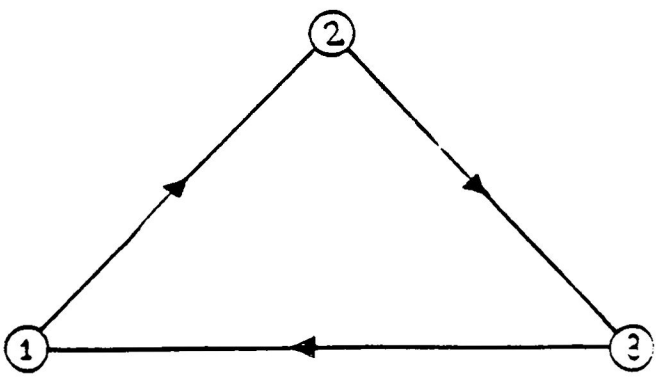

Fig. 4. Wait-for graph for Sample Run I.

As shown in WF graph a cycle exists in this graph. This conclusion is confirmed by the sample run.

Test Sample II

A larger chain of events similar to Sample Run I but with Transactions making multiple resources requests.

1: Transaction Ti request for Ri; (1 = , i < = 10)

2: T1 requests for R3

3: T1 requests for R4

4: T2 requests for R1

5: T2 requests for R3

6: T2 requests for R5

7: T3 requests for R5

8: T4 requests for R2

9: T4 requests for R4

10: T5 requests for R4

11: T5 requests for R1

Resulting WF graph is shown in Figure 5.

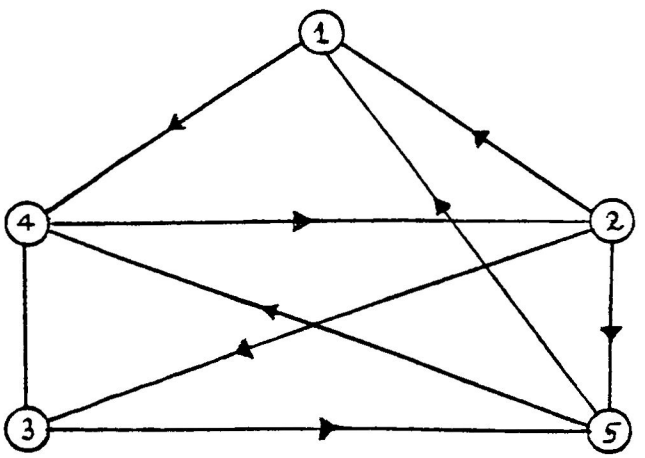

Fig. 5. WF graph for Sample Run II.

There are several cycles in the WF graph; two of these involve nodes 3, 4, 5, and nodes 1, 4, 3, 5. The program again confirms the fact that there is a cycle in the graph.

Test Sample III

This involves a mixed initial request involving local and nonlocal resources. The sequence of events are as follows:

1: T1 requests for R2

2: T2 requests for F3

3: T3 requests for R4

4: T4 requests for R7

5: T5 requests for R5

6: T6 requests for R1

7: T7 requests for R6

8: T8 requests for R8

9: T9 requests for R10

10: T10 requests for R9

11: T2 requests for R2

12: T7 requests for R3

13: T1 requests for R6

14: T3 requests for R7

15: T4 requests for R8

16: T8 requests for R9

17: T10 requests for R5

18: T5 requests for R10

19: T9 requests for R4

Graphically, we have the situation in Figure 6.

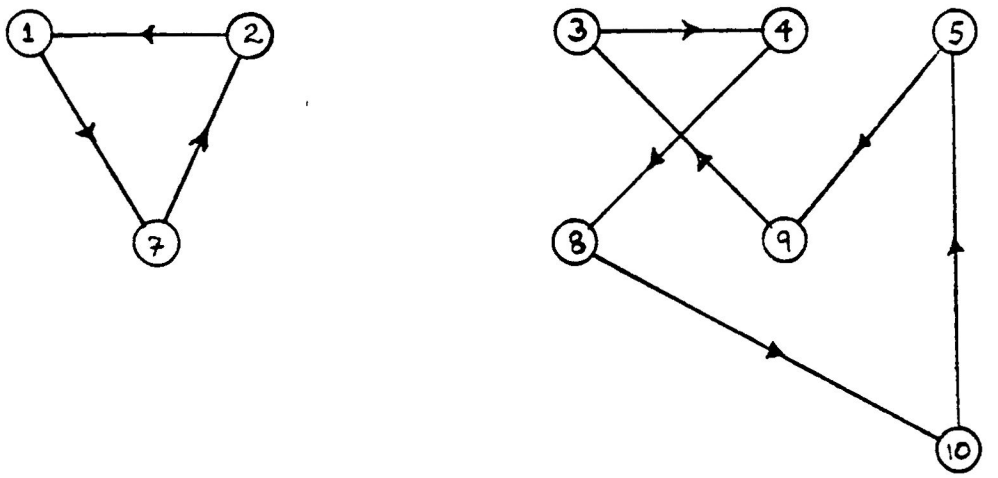

Fig. 6. Graph of Sample Run III.

There are cycles in both subgraphs and the program again confirms this fact.

Test Sample Run IV

For this test multiple requests are made by transactions in an arbitrary manner. The resources are made from both local and nonlocal sources. The sequence of events are as follows and are shown in Figure 7.

1: T1 requests for resources R4, R2, R7, R6

2: T2 requests for resources R3, R4, R6

3: T4 requests for R3

4: T5 requests R3, R4

5: T6 requests R5, R3

6: T7 requests R6, R8

7: T8 requests R5, R8

8: T9 requests R4

9: T10 requests R4, R9, R1, R7, R8

The graph shown in Figure 7 shows that there are no cycles. Again the program confirms that there are no cycles in the system.

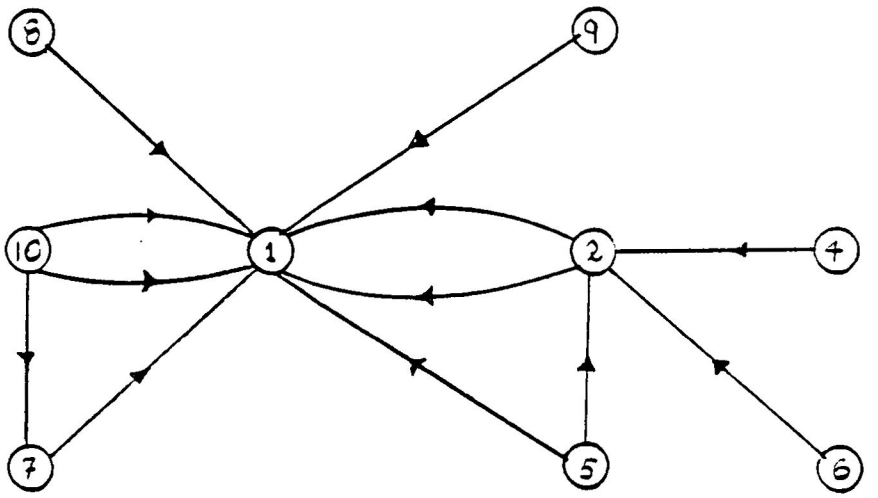

Fig. 7. Graph for Sample Run IV.

The graph shows that there are no cycles. Again the program confirms that there are no cycles in the system.

CHAPTER FOUR

CONCLUSIONS

In this final chapter, we summarize the work we have done on the topic of "Implementation of deadlock detection in distributed systems."

In Chapter One, we stated our objective in pursuing this topic; we also gave a general definition of a deadlock state and cited the conditions leading to a deadlock state. Deadlock resolution was not discussed but mention was made of two methods for avoiding a deadlock state. One of them was to do nothing to avoid deadlocks but rather try to detect and resolve them as they occur.

Chapter Two made a survey of some of the available deadlock detecting protocols. The merits and demerits of the centralized approach, decentralized approach, hierarchical approach and distributed locking and deadlock approach were mentioned. In summary, we describe them as follows:

1. The centralized approach uses a global WF graph with the aid of a single detector at a single site. High cost factor resulting from connecting spatially spaced sites was one of the drawbacks of this protocol. Also a single site as the brain of a whole system will make a system, overall, vulnerable to sudden failure in the event of any component failure at this selected single site.
2. The decentralized protocol builds an OBPL with the aid of a PMM at every site of the network. Detection of deadlock is done by

by the PMM. The major disadvantage of this protocol was that too many OBPLs are built. This proliferation of OBPLs give seed to a lot of false deadlocks.

3. The hierarchical deadlock detection algorithm is a distributed algorithm. The WF graph is spread out over all the sites in a tree structure. Some of the sites perform dual roles in terms of maintaining their local graphs as well as playing a major part in the overall tree structure of the system in a manner to be explained shortly. Coordination of data in this protocol is done by the so-called Leaf and Non-leaf controllers. Non-leaf controllers maintain IOP graphs which detect deadlock involving different sites. Local deadlocks are detected by the Leaf-controllers at the subdata base. The major disadvantages of the hierarchical model of deadlock detection is the large volume of message traffic generated in this system. Optimization of the performance of this protocol is still the subject of current research.

4. In the distributed locking and distributed deadlock approach, there is distributed control as well as distributed locking mechanism. Per-site concurrency controllers exchange messages that cause updates to take place in the WF graph at each site. Detection of deadlocks and decisions to accept or reject requests are based solely on information kept at a controllers' site. A disadvantage of this scheme is the flood of messages in the system. Despite this flaw we selected this protocol for

implementation based on the fact that it is clearly defined and as such capable of implementation with less complexity.

Chapter Three has been used to fully develop the distributed locking and distributed deadlock approach. Data structures were discussed for the scheme and routines written to implement the protocol in a most efficient manner. The resulting program was used to test four artificial but realistic chain of events with varying level of complexity. All the results were positive. Based on these sample tests, we conclude that we have achieved our initial objective of developing an implementable deadlock detecting algorithm which can be useful at the present time in a system whose nodes are not far removed from each other.

APPENDIX A

PROGRAM LISTING OF PASCAL IMPLEMENTATION OF DEADLOCK DETECTING ALGORITHM

```
{------------------------------------------------------------------------}
{                                                                        }
{ IMPLEMENTATION OF DEADLOCK DETECTION ALGORITHM USING THE GLIGOR-       }
{ SHATTUCK-MENASCE-MUNTZ PROTOCOL                                        }
{                                                                        }
{ PROGRAMMER: KWABENA BIMPONG-BOTA                                       }
{ ADVISOR   : DR. NAZIR WARSI                                            }
{                                                                        }
{ PROGRAM WRITTEN IN PARTIAL FULFILLMENT OF MASTERS DEGREE REQUIREMENT   }
{ IN COMPUTER SCIENCE  AT ATLANTA UNIVERSITY, ATLANTA                    }
{------------------------------------------------------------------------}
```

```
{ THIS PROGRAM IS USED FOR DEADLOCK DETECTION IN A DISTRIBUTED NETWORK }
{                                                                       }
{-----------------------------------------------------------------------}
{ ALGORITHM                                                             }
{-----------------------------------------------------------------------}
{AO  : WHEN A TRANSACTION T REQUESTS A NONLOCAL RESOURCE, IT IS MARKED  }
{      AS WAITING TRANSACTION                                           }
{                                                                       }
{A1  : THE RESOURCE T CANNOT BE GRANTED TO T  BECAUSE IT IS BEING HELD  }
{      BY TRANSACTIONS T1,T2,...,TK.  ADD AN ARC FROM TRANSACTION T TO  }
{      EACH OF THE TRANSACTIONS IN THE SET [T2,T2,...TK].               }
{      IF THE ADDITION OF THESE ARCS CAUSE A CYCLE TO BE FORMED AT ANY  }
{      OF THESE SITES THEN A DEADLOCK IS DETECTED AND APPROPRIATE STEPS}
{      ARE REQUIRED TO RESOLVE THE DEADLOCK.                            }
{      THE BLOCKING PAIR <T,TP> IS SENT TO OS(T) IF OS(T) IS DIFFERENT  }
{      FROM OS(TP) AND TO OS(TP) IF OS(TP) IS DIFFERENT FROM CURRENT    }
{      SITE.  BUILD A POTENTIAL BLOCKING SET FOR T.                     }
{                                                                       }
{A2.0: A BLOCKING PAIR <T,TP> IS RECEIVED. ADD AN ARC FROM T TO TP AT   }
{      THIS SITE.  IF A CYCLE IS FORMED, THEN A DEADLOCK EXISTS, AND    }
{      MUST BE RESOLVED BY AN APPROPRIATE ACTION                        }
```

```
{A2.1: IF TP IS BLOCKED AND OS(T) IS NOT EQUAL TO CURRENT SITE, THEN   }
{      FOR EACH TRANSACTION TP2 IN THE BLOCKING SET OF T, SEND THE     }
{      BLOCKING PAIR <T,TP2> TO OS(TP2) IF OS(TP2) IS DIFFERENT FROM   }
{      CURRENT SITE                                                    }
{                                                                      }
{A2.2: IF T IS WAITING AND OS(T)=CURRENT SITE THEN, FOR EACH POTENTIAL }
{     BLOCKING PAIR (TP2,T), SEND THE BLOCKING PAIR <TP2,TP> TO OS(TP2)}
{     IF OS(TP2) IS DIFFERENT FROM CURRENT SITE.  THEN DISCARD THE POT-}
{     ENTIAL BLOCKING PAIRS <TP2,T> AND ERASE WAITING MARK OF T        }
{                                                                      }
{----------------------------------------------------------------------}
{DISCUSSION OF DATA STRUCTURES                                         }
{----------------------------------------------------------------------}
{                                                                      }
{THE PROGRAM BUILDS A WAIT-FOR (WF) GRAPH AT DIFFERENT SITES, SI OF THE}
{NETWORK.                                                              }
{THE NUMBER OF TRANSACTIONS IN THE NETWORK IS GIVEN AS N.              }
{                                                                      }
{THE PROGRAM ALSO DETECTS CYCLES AT A SITE.  THE DATA STRUCTURE USED   }
{IN BUILDING THE WF GRAPH IS ADJACENCY LISTS OF GRAPH REPRESENTA-      }
{TION USING LINKED LISTS OF RECORDS WHICH CONTAIN THE FIELDS:          }
{INDEGREE---  INDEGREE OF TRANSACTION                                  }
{INNODES--- THE SET OF TRANSACTIONS ADJACENT TO TRANSACTIONS           }
```

```
{OUTDEGREE--- THE OUTDEGREE OF TRANSACTION                              }
{BLOCKSETCOUNT--NUMBER OF ELEMENTS IN BLOCKSET                          }
{BLOCKINGSET--- BLOCKING SET OF TRANSACTION                             }
{PBLOCKSETCOUNT--NUMBER OF ELEMENTS IN POTENTIAL BLOCKING SET           }
{PBLOCKINGSET-- POTENTIAL BLOCKING SET OF TRANSACTION                   }
{WAITING-- A BOOLEAN FIELD TO INDICATE WAITING STATE OF TRANS           }
{LINK-- A POINTER TO TRANSACTION ADJACENT FROM CURRENT TRANSACTION      }
{WHEN THE PAIR <I,J> IS RECEIVED AT A SITE OR SENT TO A SITE, THE       }
{OUTDEGREE FIELD OF <I> IS INCREMENTED BY ONE AND THE INDEGREE          }
{FIELD OF <J> IS LIKEWISE INCREMENTED BY ONE                            }
{THE LIST OF VERTICES WITH ZERO COUNT IS MAINTAINED AS A STACK          }
{THE STACK IS LINKED THROUGH THE INDEGREE FIELD OF TRANSACTIONS         }
{SINCE THIS FIELD IS OF NO USE AFTER THE COUNT HAS BECOME ZERO          }
{-----------------------------------------------------------------------}

{PROTOCOL IMPLEMENTATION OF ALGORITHM USING PASCAL PROGRAM. LANGUAGE    }
{-----------------------------------------------------------------------}
{                                                                       }
PROGRAM DEADLOCK (NUM,INPUT,OUTFILE,OUTPUT);
CONST SI=10;          {   NUMBER OF SITES IN THE NETWORK, THIS IS FIXED}
      SK=50;          {SIZE OF TRANSACTION/RESOURCE DISTRIBUTION ARRAYS}
    KONS=30;          {                              CIRCULAR BUFFER SIZE}
{-----------------------------------------------------------------------}
```

```
{USER TYPE DECLARATIONS                                                    }
{------------------------------------------------------------------------}
TYPE NEXTNODE=^NODE;
    NODE=RECORD
        VERTEX:INTEGER;          {NAME OF TRANSACTION BEING POINTED TO}
          LINK:NEXTNODE;                 {POINTER TO NEXT TRANSACTION}
       END;

 SETSIZE=ARRAY[1..KONS] OF INTEGER;

 VERTICES=RECORD
           ACTIVE  :BOOLEAN;        {FLAG TO INDICATE INVOLVEMENT OF }
                                    { NODE IN WF GRAPH AT A SITE     }
           INDEGREE:INTEGER;        {INDEGREE OF TRANSACTION}
            INNODES:SETSIZE;        {THE SET OF ALL INCOMMING NODES}
          OUTDEGREE:INTEGER;        {OUTDEGREE OF TRANSACTION}
      BLOCKSETCOUNT:INTEGER;        {BLOCKING SET COUNT OF TRANSACTION}
     PBLOCKSETCOUNT:INTEGER;        {POTENTIAL BLOCKING SET COUNT}
        BLOCKINGSET:SETSIZE;        {BLOCKING SET OF TRANSACTION}
       PBLOCKINGSET:SETSIZE;        {POTENTIAL BLOKING SET OF TRANS.}
            WAITING:BOOLEAN;        {WAITING MARK ON TRANSACTION}
             LINK:NEXTNODE;         {POINTER TO OTHER TRANSACTIONS}
          END;
```

```
BUFF  = RECORD
          PENDING: BOOLEAN;          {FIELD TO INDICATE STATE OF BUFFER}
          NODE1:INTEGER;             {FIRST HALF PAIR OF MESSAGE SENT  }
          NODE2:INTEGER;             {SECOND HALF PAIR OF MESSAGE SENT }
           SITE:INTEGER;             {SITE TO WHICH MESSAGE IS SENT    }
         END;

INFO =RECORD                         {AUXILLIARY RECORD TYPE FOR TRANS.}
       SITE:INTEGER;                 {SITE OF ORIGIN OF TRANSACTIONS   }
      END;

RESOURCES=RECORD
          SITE:INTEGER;              {SITE AT WHICH RESOURCES IS SITUATED}
          LOCK:BOOLEAN;              {A RESOURCE CAN BE LOCKED          }
     LOCKEDBY:INTEGER;               {TRANSACTION LOCKING RESOURCE      }
        END;

TRANSACTION=ARRAY[1..SK] OF VERTICES;

BU=ARRAY[1..KONS] OF BUFF;           {TO STORE MESSAGES TEMPORARILY    }
```

```
{-------------------------------------------------------------------}
{ VARIABLE DECLARATIONS                                             }
{-------------------------------------------------------------------}
VAR I,N,RES,BUFFERCOUNT,THOLD,RR,TR      :INTEGER;
BUFFER                                   :BU;          {FOR STORING MESSAGES}
R                                        :ARRAY[1..SK] OF RESOURCES;
T                                        :ARRAY[1..SK] OF INFO;
WF                                       :ARRAY[1..SI] OF TRANSACTION;
TERMINATE,GRANTED,DEAD                   :BOOLEAN;

{-------------------------------------------------------------------}
{EXTERNAL FILE DECLARATIONS                                         }
NUM,OUTFILE                              :TEXT;

{-------------------------------------------------------------------}
{PROCEDURE INITIALIZE                                               }
{-------------------------------------------------------------------}
{EVERY RESOURCE IN THE SYSTEM BELONGS TO A SITE IN THE NETWORK. LIKEWISE}
{EVERY TRANSACTION IN THE SYSTEM ORIGINATES FROM A SITE.  THIS PROCEDURE}
{IS USED TO INPUT THE LAYOUT OF RESOURCES AND TRANSACTIONS IN THE SYSTEM}
{THE INFORMATION ABOUT RESOURCES AND TRANSACTIONS ARE KEPT IN A FILE CAL}
{LED NUM.DAT. THE PROCEDURE IS ALSO USED TO INITIALIZE THE EXTERNAL FILE}
{OUTFILE.DAT.          }
```

```
PROCEDURE INITIALIZE(VAR RES,TRANS:INTEGER);
VAR  I,J    :INTEGER;
BEGIN
RESET(NUM);
REWRITE(OUTFILE);
RES:=0;          {INITIALIZE COUNTER FOR RESOURCES}
{'ENTER RESOURCE NUMBER AND ITS CORRESP. SITE NUMBER FROM EXTERNAL FILE}
{'ENTER <0 0> WHEN YOU WISH TO TERMINATE INPUT}
READLN(NUM,I,J);
WHILE ((I<>0) AND (J<>0)) DO
 BEGIN
  RES:=RES + 1;
  R[I].SITE:=J;
  {'ENTER NEXT RESOURCE NUMBER AND ITS CORRESP. SITE NUMBER'}
  READLN(NUM,I,J);
 END; { OF WHILE}
TRANS:=0;   {COUNTER FOR TRANSACTION}
{'ENTER TRANSACTION NUMBER AND CORRESP. SITE NUMBER'}
{'ENTER  <0 0> TO TERMINATE'}
 READLN(NUM,I,J);
```

```
 WHILE ((I<>0) AND (J<>0)) DO
  BEGIN
   TRANS:=TRANS + 1;
   T[I].SITE:=J;
   {'ENTER NEXT TRANSACTION NUMBER AND CORRESP. SITE NUMBER'}
   READLN(NUM,I,J);
  END; {OF WHILE}
 { DOCUMENT ENTERED DATA INTO OUTPUT EXTERNAL FILE}
 WRITELN(OUTFILE,'RESOURCE AND TRANSACTION DISTRIBUTION IN NETWORK');
 WRITELN(OUTFILE, 'RESOURCE':9,'SITE':10);
 FOR I:=1 TO RES DO WRITELN(OUTFILE,I:5,R[I].SITE:12);
 WRITELN(OUTFILE);
 WRITELN(OUTFILE,'TRANSACTION':12,'SITE':10);
 FOR I:=1 TO TRANS DO WRITELN(OUTFILE,I:5,T[I].SITE:15);
END;{OF INITIALIZE}
{-------------------------------------------------------------------}
{PROCEDURE REQUEST                                                  }
{-------------------------------------------------------------------}
{A TRANSACTION TR, REQUESTS FOR A RESOURCE, RE.  IF THE RESOURCE IS NOT}
{LOCKED THAT IS, LOCK FIELD IS NOT FALSE, THEN THE RESOURCE IS GRANTED,}
{THE LOCK FIELD OF THE RESOURCE IS SET TO TRUE, AND LOCKEDBY FIELD AS- }
{SIGNED TO TR. IF THE RESOURCE BEING REQUESTED FOR IS AT A DIFFERENT   }
{SITE FROM THE TRANSACTION, THEN THE TRANSACTION TR IS MARKED WAITING  }
{BY SETTING ITS WAITING FIELD TO TRUE.                                 }
```

```
PROCEDURE REQUEST(TR:INTEGER;RE:INTEGER);
VAR I,J,SITE                              :INTEGER;
BEGIN
 SITE:=R[RE].SITE;
 I   :=T[TR].SITE;
 J   :=SITE;
 IF I<>J THEN                          {A REQUEST FOR A NONLOCAL RESOURCE}
  WF[I,TR].WAITING:=TRUE
 ELSE WF[I,TR].WAITING:=FALSE;          {MARK TRANSACTION AS WAITING}
 IF  NOT R[RE].LOCK THEN
BEGIN
 R[RE].LOCK:=TRUE;                       {PUT A LOCK ON RESOURCE}
 GRANTED:=TRUE;        {RESOURCE IS ALLOCATED TO REQUESTING TRANSACTION}
 R[RE].LOCKEDBY:=TR;             {NAME OF TRANSACTION LOCKING RESOURCE}
 WRITELN('TRANSACTION':11,TR:2,'REQUEST FOR RESOURCE':22,RE:2,'GRANTED':8);
 WRITELN(OUTFILE,'TRANSACTION':11,TR:2,'REQUEST FOR RESOURCE':22,RE:2,
 'IS GRANTED':11);
END
ELSE
 BEGIN
  GRANTED:=FALSE;
  WRITELN('REQUEST FOR RESOURCE DENIED');
  WRITELN(OUTFILE,'TRANSACTION':11,TR:2,'REQUEST FOR RESOURCE':22,
  RE:2,'IS DENIED':11);
```

```
  THOLD:=R[RE].LOCKEDBY;
 END;
END; {OF REQUEST}
{-------------------------------------------------------------------}
{PROCEDURE ADDARC                                                   }
{-------------------------------------------------------------------}
{GIVEN A CURRENT SITE AND A PAIR <T1,T2>, THE PROCEDURE ADDS THE PAIR }
{TO THE WAIT-FOR GRAPH AT THAT SITE. ADDITIONALLY, IT ALSO INCREMENTS }
{THE OUTDEGREE OF T1 BY ONE AND THE INDEGREE COUNT OF T2 BY ONE.  THE }
{TRANSACTION T1, INCIDENT TO T2 IS STORED AT THE INNODES FIELD OF T2. }
{IN EFFECT T1 IS WAITING FOR T2.                                    }
{-------------------------------------------------------------------}
PROCEDURE ADDARC(SITE:INTEGER;NODE1,NODE2:INTEGER);
VAR PTR,K:NEXTNODE;
      J:INTEGER;
BEGIN
NEW(PTR);
PTR^.VERTEX:=NODE2;  {NODE2 IS ADJACENT FROM NODE1}
WF[SITE,NODE1].OUTDEGREE :=WF[SITE,NODE1].OUTDEGREE + 1;{OUTDEG INCREASED}
WF[SITE,NODE2].INDEGREE:=WF[SITE,NODE2].INDEGREE + 1; {INDEGREE INCREASED}
WF[SITE,NODE1].ACTIVE:=TRUE;   {NODE1 MARKED AS ACTIVE}
WF[SITE,NODE2].ACTIVE:=TRUE;   {NODE2 MARKED AS ACTIVE}
J:=WF[SITE,NODE2].INDEGREE;
```

```
WF[SITE,NODE2].INNODES[J]:=NODE1; {BUILD A SET OF TRANSACTIONS INCIDENT}
                                  { TO NODE2}
IF WF[SITE,NODE1].LINK=NIL
  THEN WF[SITE,NODE1].LINK:=PTR
ELSE
    BEGIN
     K:=WF[SITE,NODE1].LINK;
     WHILE K^.LINK <> NIL DO K:=K^.LINK;
     K^.LINK:=PTR
    END;{OF ELSE}
END; {OF ADDARC}
{PROCEDURE SHOWMESSAGES                                                    }
{--------------------------------------------------------------------------}
{THIS PROCEDURE DISPLAYS THE CONTENTS OF THE CURRENT BUFFER. MESSAGES  }
{WERE PUT THERE TEMP. UNTIL THEY COULD BE ACCEPTED AT THE INTENDED SITE}
{--------------------------------------------------------------------------}
PROCEDURE SHOWMESSAGES(BUFFERCOUNT:INTEGER);
VAR I:INTEGER;
BEGIN
 WRITELN(OUTFILE);
 WRITELN(OUTFILE,'NUMBER','T1':8,'T2':9,'SITE':12);
 FOR I:=1 TO BUFFERCOUNT DO
  WRITELN(OUTFILE,' ':1,I:2,BUFFER[I].NODE1,BUFFER[I].NODE2,BUFFER[I].SITE);
  WRITELN(OUTFILE);WRITELN(OUTFILE);
END;
```

```
{-------------------------------------------------------------------------}
{PROCEDURE SHOWGRAF                                                       }
{-------------------------------------------------------------------------}
{GIVEN A PARTICULAR SITE, THIS PROCEDURE DISPLAYS THE WAIT-FOR GRAPH OF}
{THE SITE. THE PROCEDURE LISTS THE OUTDEGREE, INDEGREE,TRANSACTION NUMB}
{AND ALL OTHER NODES ADJACENT TO ALL ACTIVE NODES                         }
{-------------------------------------------------------------------------}
PROCEDURE SHOWGRAF(SITE:INTEGER);
VAR I:INTEGER;
     K:NEXTNODE;
 BEGIN
 WRITELN(OUTFILE,'ADJACENCY LIST AT SITE':24,SITE:3);
 WRITELN(OUTFILE,'-------- ---- -- ----':24,'--':3);
 WRITELN(OUTFILE);
 WRITELN(OUTFILE,'OUTDEGREE':9,'INDEGREE':12,'TRANSACTION':12,'ADJACENT
 TO...':20);
 WRITELN(OUTFILE);
 FOR I:=1 TO N DO
  IF WF[SITE,I].ACTIVE=TRUE THEN
   BEGIN
    WRITE(OUTFILE,WF[SITE,I].OUTDEGREE:4,WF[SITE,I].INDEGREE:13,I:12);
     K:=WF[SITE,I].LINK;
     IF K<> NIL THEN
```

```
    BEGIN
      WRITE(OUTFILE,'--->':20);
      REPEAT
      WRITE(OUTFILE,K^.VERTEX:4);
      K:=K^.LINK;
     UNTIL K=NIL;
   END;{OF IF}
   WRITELN(OUTFILE);
  END;{OF IF}
END;{OF SHOWGRAF}
{-------------------------------------------------------------------}
{PROCEDURE DEADLHANDLE                                              }
{-------------------------------------------------------------------}
{PRINTS A MEESAGE THAT A DEADLOCK HAS BEEN DETECTED AT A SITE       }
{-------------------------------------------------------------------}
PROCEDURE DEADLHANDLE(SITE:INTEGER);
  BEGIN
   DEAD:=TRUE;                    {FLAG TO INDICATE DEADLOCK EXISTENCE}
   WRITELN('DEADLOCK DETECTED AT SITE:':26,SITE:3);
   WRITELN(OUTFILE);
   WRITELN(OUTFILE);
   WRITELN(OUTFILE,'DEADLOCK DETECTED AT SITE:':26,SITE:3);
   WRITELN(OUTFILE);
   WRITELN(OUTFILE);
  END;
```

```
{FUNCTION DEADLDETECT                                                    }
{------------------------------------------------------------------------}
{THIS BOOLEAN FUNCTION CHECKS IF A GIVEN SITE IS IN A DEADLOCK STATE OR}
{NOT. THE MECHANISM USED IS A STACK TO MONITOR ALL THE TRANSACTIONS AT }
{A SITE WITH ZERO INDEGREE COUNT. AFTER MANIPULATION OF THIS STACK IF IT}
{TURNS OUT THAT THE TOP OF THE STACK IS ZERO THEN THERE IS A DEADLOCK. }
{THE FUNCTION WORKS WITH A COPY OF THE WF-GRAPH OF A SITE              }
{------------------------------------------------------------------------}
FUNCTION DEADLDETECT(SITE:INTEGER):BOOLEAN;
VAR I,J,K,TOP:INTEGER;
    PTR:NEXTNODE;    DONE:BOOLEAN;    TEMP:TRANSACTION;
BEGIN
  TEMP:=WF[SITE];        {COPY GRAPH OF SITE INTO TEMPORARY VARIABLE}
  TOP:=0;                {INITIALIZE STACK}
   FOR I:=1 TO N DO          {CREATE A LINKED STACK OF VERTICES WITH}
     IF TEMP[I].INDEGREE=0     {INDEGREE ZERO}
       THEN
           BEGIN
            TEMP[I].INDEGREE:=TOP;
            TOP:=I;
           END;
   I:=1;
   DONE:=FALSE;
   WHILE((I<=N) AND NOT DONE) DO
```

```
BEGIN
  IF TOP=0
     THEN
         BEGIN
           DEADLDETECT:=TRUE;
           DONE:=TRUE;
         END
  ELSE J:=TOP;
  TOP:=TEMP[TOP].INDEGREE;            {UNSTACK A VERTEX}
  PTR:=TEMP[J].LINK;
  WHILE PTR<> NIL DO
    BEGIN
      {DECREASE THE INDEGREE OF THE VERTEX CONNECTED TO J}
       K:=PTR^.VERTEX; {K IS LINKED TO J}
      TEMP[K].INDEGREE:=TEMP[K].INDEGREE - 1;{DECREASE INDEGREE}
      IF TEMP[K].INDEGREE=0  {ADD VERTEX K TO STACK}
       THEN
         BEGIN
            TEMP[K].INDEGREE:=TOP;
            TOP:=K;
          END; {END OF IF}
      PTR:=PTR^.LINK;
   END; {OF WHILE PTR <> NIL}
  I:=I+1;
```

```
  END; {OF WHILE (I<=N) AND NOT DONE}
 IF DONE=FALSE THEN DEADLDETECT:=FALSE;
END; {OF DEADLDETECT}
{PROCEDURE CREATEBLOCKSET                                                    }
{------------------------------------------------------------------------}
{GIVEN A PAIR <T1,T2> IF T2 IS NONBLOCKED (THAT IS IF OUTDEGREE OF     }
{T2=0), THEN BLOCKINGSETCOUNT OF T1 IS INCREASED BY ONE. T2 IS ADDED TO}
{THE BLOCKING SET T1                                                         }
{------------------------------------------------------------------------}
PROCEDURE CREATEBLOCKSET(SITE,T1,T2:INTEGER);
VAR K      :INTEGER;
BEGIN
 IF WF[SITE,T2].OUTDEGREE =0  {TRANSACTION BLOCKED}
   THEN
      BEGIN {INCREASE BLOCKING SET FOR T1 BY 1}
        K:=WF[SITE,T1].BLOCKSETCOUNT + 1;
        WF[SITE,T1].BLOCKINGSET[K]:=T2;
        WF[SITE,T1].BLOCKSETCOUNT := K;
      END; {OF IF}
END; {OF CREATEBLOCKSET}
```

```
{PROCEDURE BUFFERTRIPLE                                                    }
{--------------------------------------------------------------------------}
{THIS IS AN AUXILLIARY PROCEDURE TO PROCEDURE SENDPAIR.  ALL IT DOES IS}
{TO UPDATE THE BUFFER PARAMETERS GIVEN THE TRANSACTION PAIR SITE TO BE }
{SENT      }
{--------------------------------------------------------------------------}
PROCEDURE BUFFERTRIPLE(T1,T2,SITE:INTEGER);
VAR J          :INTEGER;
BEGIN
  BUFFERCOUNT:=(BUFFERCOUNT + 1) ;
  J:=BUFFERCOUNT;
  BUFFER[J].NODE1:=T1;    {INCOMING T1 STORED IN FIELD NODE1}
  BUFFER[J].NODE2:=T2;    {INCOMING T2 STORED IN FIELD NODE2}
  BUFFER[J].SITE:=SITE;   {INCOMING SITE STORED IN FIELD SITE}
  BUFFER[J].PENDING:=TRUE;
END;
{--------------------------------------------------------------------------}
{PROCEDURE SENDPAIR}
{--------------------------------------------------------------------------}
{GIVEN A SITE AND A TRANSACTION T, THE BLOCKING PAIR (T,T') IS SENT TO}
{THE BUFFER.  IF SITE OF ORIGIN OF T IS NOT EQUAL TO SITE THEN THE    }
{BUFFER.SITE FIELD OF BUFFER IS SET TO SITE OF ORIGIN OF T; OTHERWISE}
{THE FIELD IS SET TO}
{TO SITE OF ORIGIN OF T'                                                   }
```

```
PROCEDURE SENDPAIR(SITE,T1:INTEGER);
VAR I,S,J,K,T2 :INTEGER;
BEGIN
 {FOR ALL ELEMENTS IN THE BLOCKINGSET FOR T1 DO THE FOLLOWING}
 FOR I:=1 TO WF[SITE,T1].BLOCKSETCOUNT DO
   BEGIN
     K:=T[T1].SITE;     {SITE OF ORIGIN OF T1}
     T2:=WF[SITE,T1].BLOCKINGSET[I];
      S:=T[T2].SITE;    {SITE OF ORIGIN OF T2}
     IF K<> SITE THEN BUFFERTRIPLE(T1,T2,K);
     IF S<> SITE THEN BUFFERTRIPLE(T1,T2,S);
    END;
END;  {OF SENDPAIR}
{-------------------------------------------------------------------}
{PROCEDURE CREATEPOTENTIAL BLOCKINGSET                               }
{-------------------------------------------------------------------}
{GIVEN A PAIR <T1,T2> IF T2 IS WAITING, THEN INCREASE POTENTIAL      }
{BLOCKING SET COUNTER OF T1 BY ONE.  T2 IS ADDED TO THE POTENTIAL    }
{BLOCKING SET OF T1       }
```

```
PROCEDURE CREATEPBLOCKSET(SITE,T1,T2:INTEGER);
VAR K    :INTEGER;
BEGIN
 IF WF[SITE,T2].WAITING=TRUE
    THEN
       BEGIN  {INCREASE POTENTIAL BLOCKING SET BY 1}
        K:=(WF[SITE,T1].PBLOCKSETCOUNT +`1) ;
        WF[SITE,T1].PBLOCKINGSET[K]:=T2;
        WF[SITE,T1].PBLOCKSETCOUNT:=K;
       END; {OF IF}
END; {OF CREATEPBLOCKSET}
{PROCEDURE RECEIVE                                                    }
{---------------------------------------------------------------------}
{THIS IS THE IMPLEMENTATION OF THE ALGORITHM A2.0, A2.1, AND A2.2 WITH }
{THE SUPPORT OF THE ROUTINES EXPLAINED ABOVE.                         }
{THIS PROCEDURE WORKS ON THE CONTENTS OF THE BUFFER.  WHEN A TRIPLE    }
{<T1,T2,SITE> IS FETCHED FROM THE BUFFER, THE PAIR <T1,T2> IS ADDED TO }
{THE WF(SITE). A CHECK IS MADE TO SEE IF A DEADLOCK STATE IS ESTABLISHED}
{WITH THE ADDITION OF THIS PAIR.  IF THERE IS A DEADLOCK THE PROCEDURE }
{THAT HANDLES DEADLOCK IS INVOKED OTHERWISE THE FOLLOWING ARE EXECUTED }
{ 1.IF T2 IS BLOCKED AND OS(T1) IS DIFFERENT FROM CURRENT SITE THEN FOR}
{ALL T3 BELONGING TO THE BLOCKING SET OF T1 ADD THE TRIPLE <T1,T3,SITE>}
{TO BUFFER WHERE SITE=OS(T3)                                          }
{ 2. IF T1 IS WAITING AND OS(T1)=CURRENT SITE , THEN, GET ALL TRANSAC- }
{TIONS INCIDENT TO T1--THAT IS GET TRANSACTION NAMES FROM INNODES FIELD}
```

```
{THEN FOR ALL SUCH TRANSACTIONS T3, IF OS(T3) IS NOT EQUAL TO CURRENT  }
{SITE THEN ADD THE TRIPLE <T3,T2,OS(T3)> UNTO BUFFER. T1.WAITING TO    }
{FALSE; AND ERASE POTENTIAL-BLOCKING PAIR (T3,T1).                     }
PROCEDURE RECEIVE(VAR BUFER:BU);{THIS WORKS ON THE  BUFFER CONTENTS}
VAR    MAX,K,T1,T2,T3,I,SITE       :INTEGER;
BEGIN
{WORK ON ALL MESSAGES IN BUFFER}
{CLEAR EACH BUFFER AFTER USING ITS CONTENTS}
 FOR I:=1 TO KONS DO
  IF BUFER[I].PENDING<>FALSE  {IF BUFFER IS EMPTY, IGNORE}
     THEN
      BEGIN
         T1:=BUFER[I].NODE1;
         T2:=BUFER[I].NODE2;
       SITE:=BUFER[I].SITE;
       ADDARC(SITE,T1,T2); {ADD PREVIOUSLY SENT PAIR TO APPROPRIATE SITE}
        BUFER[I].PENDING:=FALSE; {MARK BUFFER CONTENTS  AS RECEIVED}
        IF DEADLDETECT(SITE)
           THEN
              BEGIN
               DEADLHANDLE(SITE);
               I:=KONS;
              END
        ELSE
```

```
BEGIN
{CHECK IF T2 IS BLOCKED AND OS(T1)IS SAME AS CURRENT SITE}
IF ((WF[SITE,T2].OUTDEGREE<> 0) AND (T[T1].SITE<>SITE))
    THEN
     BEGIN
      MAX:=WF[SITE,T1].BLOCKSETCOUNT;  {GET # OF TRANSACTIONS}
                                       { BLOCKING T1}
      FOR K:=1 TO MAX DO
      BEGIN
          T3:=WF[SITE,T1].BLOCKINGSET[K];
          IF T[T3].SITE<> SITE
            THEN BUFFERTRIPLE(T1,T3,T[T3].SITE);
      END;
    END;
    {IF T1 IS WAITING AND OS(T1)=CURRENT SITE DO THE FOLLOWING}
    IF ((WF[SITE,T1].WAITING) AND (T[T1].SITE=SITE))
       THEN
        BEGIN
         MAX:=WF[SITE,T1].INDEGREE;{GET NUMBER OF TRANSACTIONS}
                                   { AJACENT TO T1}
                                   {FOR ALL SUCH TRANSACTIONS,IF }
                                   { OS(T3)<>CURRENT SITE, }
                                   { THEN SEND <T3,T2> TO OS(T3)}
         FOR K:=1 TO MAX DO
         BEGIN
```

```
                T3:=WF[SITE,T1].INNODES[K];     {T3 IS ASJACENT FROM}
                IF T[T3].SITE<> SITE
                THEN BUFFERTRIPLE(T3,T2,T[T3].SITE);
               END;
              {ERASE WAITING MARK ON T}
              {DISCARD POTENTIAL BLOCKING SET}
             WF[SITE,T1].WAITING:=FALSE;
             WF[SITE,T3].PBLOCKSETCOUNT:=0;
          END;
       END;
    END;
END;    {OF RECEIVE}
{-------------------------------------------------------------------}
{PROCEDURE MANIPULATE                                               }
{-------------------------------------------------------------------}
{THIS IS IMPLEMENTATION OF PARTS A0 AND A1 OF ALGORITHM. IF A TRANSACTION}
{REQUESTS FOR A RESOURCE AND ITS NOT GRANTED, THEN IT UPDATES THE WF-GRAPH
{AT THE APPROPRIATE SITE. IT CHECKS IF A DEADLOCK HAS OCCURED WITH THIS}
{UPDATE AT THE SITE. IF NO DEADLOCK HAS OCCURED THEN MESSAGES IN BLOCKING}
{PAIRS ARE SENT OVER TO APPROPRIATE SITES. BLOCKINGSETS AND POTENTIAL }
{BLOCKING SETS FOR TRANSACTIONS ARE BUILT                           }
{-------------------------------------------------------------------}
PROCEDURE MANIPULATE(T1,T2:INTEGER);
VAR SITE:INTEGER;
```

```
BEGIN
 SITE:=T[T2].SITE;
 ADDARC(SITE,T1,T2);
 IF DEADLDETECT(SITE) THEN
   BEGIN
    DEADLHANDLE(SITE);
    TERMINATE:=TRUE;
   END
 ELSE
   BEGIN
   CREATEBLOCKSET(SITE,T1,T2);
   SENDPAIR(SITE,T1);
   CREATEPBLOCKSET(SITE,T1,T2);
   END;
END;{MANIPULATE}
{ MAIN PROGRAM SECTION                                                    }
{-------------------------------------------------------------------------}
BEGIN (*MAIN*)
{INITIALIZE THE SYSTEM BY GETTING DATA FROM EXTERNAL FILE NUM.DAT}
{DATA OBTAINED ARE RESOURCE AND TRANSACTION AT SITES}
INITIALIZE(RES,N);   {RES--# OF RESOURCES IN NETWORK,N--# OF TRANSACTIONS}
WRITELN('PLEASE MAKE REQUESTS BY TYPING IN A TRANSACTION NUMBER AND ');
WRITELN('RESOURCE NUMBER.   TERMINATE REQUESTS BY TYPING < 0 0 >');
```

```
WRITELN;
WRITELN(OUTFILE);
WRITELN(OUTFILE,'SEQUENCE OF TRANSACTIONS AS THEY COME IN AND RESPONSES');
WRITELN(OUTFILE);
WRITELN('TRANSACTION NUMBER?':18,'RESOURCE NUMBER?':18);
READLN(TR,RR);
WHILE (TR<>0) DO
 BEGIN
  REQUEST(TR,RR);                {TRANSACTION TR REQUESTS FOR RESOURCE RR}
  IF NOT GRANTED THEN MANIPULATE(TR,THOLD); {THOLD IS THE TRANSACTION}
                                  {HOLDING THE RESOURCE REQUESTED BY TR}
  {GET NEXT REQUEST}
  WRITELN('TRANSACTION NUMBER?':18,'RESOURCE NUMBER?':18);
  READLN(TR,RR);
 END;
WRITELN(OUTFILE);
WRITELN(OUTFILE,'MESSAGES STORED IN BUFFER BEFORE TRANSMISSION TO SITES');
WRITELN(OUTFILE,'TRANSACTION REQUESTS:');
SHOWMESSAGES(BUFFERCOUNT);
IF (NOT TERMINATE) THEN RECEIVE(BUFFER);
WRITELN(OUTFILE,'TOTAL MESSAGES STORED IN BUFFER DURING RUN');
WRITELN(OUTFILE);
SHOWMESSAGES(BUFFERCOUNT);
```

```
 FOR I:=1 TO SI DO
  BEGIN
   SHOWGRAF(I);
   WRITELN(OUTFILE);
   WRITELN(OUTFILE);
  END;
  IF DEAD<>TRUE THEN WRITELN(OUTFILE,'NO DEADLOCK STATE');
END.
```

APPENDIX B

LISTING OF FOUR SAMPLE RUNS

For all test runs the following assumptions are made for the distribution of resources and transactions at Ten Sites.

RESOURCE	SITE
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	1
12	2
13	3
14	4
15	5
16	6
17	7
18	8
19	9
20	10

TRANSACTION	SITE
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	1
12	2
13	3
14	4
15	5
16	6
17	7
18	8
19	9
20	10

TEST SAMPLE NUMBER ONE

SEQUENCE OF TRANSACTIONS AS THEY COME IN AND RESPONSES

TRANSACTION 1 REQUEST FOR RESOURCE 1 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 2 IS GRANTED

TRANSACTION 3 REQUEST FOR RESOURCE 3 IS GRANTED

TRANSACTION 1 REQUEST FOR RESOURCE 2 IS DENIED

TRANSACTION 2 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 3 REQUEST FOR RESOURCE 1 IS DENIED

MESSAGES STORED IN BUFFER BEFORE TRANSMISSION TO SITES

TRANSACTION REQUESTS:

NUMBER	T1	T2	SITE
1	1	2	1
2	2	3	2
3	3	1	3

DEADLOCK DETECTED AT SITE: 3

OVERALL MESSAGES STORED IN BUFFER BEFORE DEADLOCK DETECTION AT SITE

NUMBER	T1	T2	SITE
1	1	2	1
2	2	3	2
3	3	1	3
4	3	2	3
5	1	3	1
6	2	1	2

ADJACENCY LIST AT SITE 1

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	1	1	---> 2
0	1	2	
1	0	3	---> 1

ADJACENCY LIST AT SITE 2

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	1	---> 2
1	1	2	---> 3
0	1	3	

ADJACENCY LIST AT SITE 3

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
1	1	2	---> 3
2	1	3	---> 1 2

TEST SAMPLE NUBER TWO

SEQUENCE OF TRANSACTIONS AS THEY COME IN AND RESPONSES

TRANSACTION 1 REQUEST FOR RESOURCE 1 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 2 IS GRANTED

TRANSACTION 3 REQUEST FOR RESOURCE 3 IS GRANTED

TRANSACTION 4 REQUEST FOR RESOURCE 4 IS GRANTED

TRANSACTION 5 REQUEST FOR RESOURCE 5 IS GRANTED

TRANSACTION 6 REQUEST FOR RESOURCE 6 IS GRANTED

TRANSACTION 7 REQUEST FOR RESOURCE 7 IS GRANTED

TRANSACTION 8 REQUEST FOR RESOURCE 8 IS GRANTED

TRANSACTION 9 REQUEST FOR RESOURCE 9 IS GRANTED

TRANSACTION10 REQUEST FOR RESOURCE10 IS GRANTED

TRANSACTION 1 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 1 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION 2 REQUEST FOR RESOURCE 1 IS DENIED

TRANSACTION 2 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 2 REQUEST FOR RESOURCE 5 IS DENIED

TRANSACTION 3 REQUEST FOR RESOURCE 5 IS DENIED

TRANSACTION 4 REQUEST FOR RESOURCE 2 IS DENIED

TRANSACTION 4 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 5 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION 5 REQUEST FOR RESOURCE 1 IS DENIED

MESSAGES STORED IN BUFFER BEFORE TRANSMISSION TO SITES

TRANSACTION REQUESTS:

NUMBER	T1	T2	SITE
1	1	3	1
2	1	4	1
3	2	1	2
4	2	3	2
5	2	5	2
6	3	5	3
7	4	2	4
8	4	3	4
9	5	4	5
10	5	1	5

DEADLOCK DETECTED AT SITE: 5

OVERALL MESSAGES STORED IN BUFFER BEFORE DEADLOCK DETECTION

NUMBER	T1	T2	SITE
1	1	3	1
2	1	4	1
3	2	1	2
4	2	3	2
5	2	5	2
6	3	5	3
7	4	2	4
8	4	3	4
9	5	4	5
10	5	1	5
11	2	3	2
12	5	3	5
13	4	1	4
14	1	5	1
15	2	5	2
16	4	5	4
17	1	2	1
18	5	2	5
19	2	4	2
20	3	4	3

ADJACENCY LIST AT SITE 1

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
2	2	1	---> 3 4
1	0	2	---> 1
0	1	3	
0	1	4	
1	0	5	---> 1

ADJACENCY LIST AT SITE 2

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
4	1	2	---> 1 3 5 3
0	2	3	
1	0	4	---> 2
0	1	5	

ADJACENCY LIST AT SITE 3

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	1	---> 3
1	0	2	---> 3
1	3	3	---> 5
1	0	4	---> 3
0	1	5	

ADJACENCY LIST AT SITE 4

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	1	---> 4
0	1	2	
0	1	3	
2	2	4	---> 2 3
1	0	5	---> 4

ADJACENCY LIST AT SITE 5

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
1	0	2	---> 5
1	1	3	---> 5
0	1	4	
3	2	5	---> 4 1 3

TEST SAMPLE THREE

SEQUENCE OF TRANSACTIONS AS THEY COME IN AND RESPONSES

TRANSACTION 1 REQUEST FOR RESOURCE 2 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 3 IS GRANTED

TRANSACTION 3 REQUEST FOR RESOURCE 4 IS GRANTED

TRANSACTION 4 REQUEST FOR RESOURCE 7 IS GRANTED

TRANSACTION 5 REQUEST FOR RESOURCE 5 IS GRANTED

TRANSACTION 6 REQUEST FOR RESOURCE 1 IS GRANTED

TRANSACTION 7 REQUEST FOR RESOURCE 6 IS GRANTED

TRANSACTION 8 REQUEST FOR RESOURCE 8 IS GRANTED

TRANSACTION 9 REQUEST FOR RESOURCE10 IS GRANTED

TRANSACTION10 REQUEST FOR RESOURCE 9 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 2 IS DENIED

TRANSACTION 7 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 1 REQUEST FOR RESOURCE 6 IS DENIED

TRANSACTION 3 REQUEST FOR RESOURCE 7 IS DENIED

TRANSACTION 4 REQUEST FOR RESOURCE 8 IS DENIED

TRANSACTION 8 REQUEST FOR RESOURCE 9 IS DENIED

TRANSACTION10 REQUEST FOR RESOURCE 5 IS DENIED

TRANSACTION 5 REQUEST FOR RESOURCE10 IS DENIED

TRANSACTION 9 REQUEST FOR RESOURCE 4 IS DENIED

MESSAGES STORED IN BUFFER BEFORE TRANSMISSION TO SITES

TRANSACTION REQUESTS:

NUMBER	T1	T2	SITE
1	2	1	2
2	7	2	7
3	1	7	1
4	3	4	3
5	4	8	4
6	8	10	8
7	10	5	10
8	5	9	5
9	9	3	9

DEADLOCK DETECTED AT SITE: 1

OVERALL MESSAGES STORED IN BUFFER BEFORE DEADLOCK DETECTION

NUMBER	T1	T2	SITE
1	2	1	2
2	7	2	7
3	1	7	1
4	3	4	3
5	4	8	4
6	8	10	8
7	10	5	10
8	5	9	5
9	9	3	9
10	1	2	1
11	2	7	2
12	9	4	9
13	3	8	3
14	4	10	4
15	8	5	8
16	10	9	10
17	5	3	5

ADJACENCY LIST AT SITE 1

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
2	1	1	---> 7 2
1	1	2	---> 1
0	1	7	

ADJACENCY LIST AT SITE 2

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
1	1	2	---> 1
1	0	7	---> 2

ADJACENCY LIST AT SITE 3

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	1	3	---> 4
0	1	4	
1	0	9	---> 3

ADJACENCY LIST AT SITE 4

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	3	---> 4
1	1	4	---> 8
0	1	8	

ADJACENCY LIST AT SITE 5

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	1	5	---> 9
0	1	9	
1	0	10	---> 5

ADJACENCY LIST AT SITE 6

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...

ADJACENCY LIST AT SITE 7

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	1	---> 7
0	1	2	
1	1	7	---> 2

ADJACENCY LIST AT SITE 8

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
1	0	4	---> 8
1	1	8	---> 10
0	1	10	

ADJACENCY LIST AT SITE 9

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	3	
1	0	5	---> 9
1	1	9	---> 3

ADJACENCY LIST AT SITE 10

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	5	
1	0	8	---> 10
1	1	10	---> 5

TEST SAMPLE FOUR

SEQUENCE OF TRANSACTIONS AS THEY COME IN AND RESPONSES

TRANSACTION 1 REQUEST FOR RESOURCE 4 IS GRANTED

TRANSACTION 1 REQUEST FOR RESOURCE 2 IS GRANTED

TRANSACTION 1 REQUEST FOR RESOURCE 7 IS GRANTED

TRANSACTION 1 REQUEST FOR RESOURCE 6 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 3 IS GRANTED

TRANSACTION 2 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION 2 REQUEST FOR RESOURCE 6 IS DENIED

TRANSACTION 4 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 5 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 5 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION 6 REQUEST FOR RESOURCE 5 IS GRANTED

TRANSACTION 6 REQUEST FOR RESOURCE 3 IS DENIED

TRANSACTION 7 REQUEST FOR RESOURCE 6 IS DENIED

TRANSACTION 7 REQUEST FOR RESOURCE 8 IS GRANTED

TRANSACTION 8 REQUEST FOR RESOURCE 5 IS DENIED

TRANSACTION 8 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION 9 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION10 REQUEST FOR RESOURCE 4 IS DENIED

TRANSACTION10 REQUEST FOR RESOURCE 9 IS GRANTED

TRANSACTION10 REQUEST FOR RESOURCE 1 IS GRANTED

TRANSACTION10 REQUEST FOR RESOURCE 7 IS DENIED

TRANSACTION10 REQUEST FOR RESOURCE 8 IS DENIED

MESSAGES STORED IN BUFFER BEFORE TRANSMISSION TO SITES

TRANSACTION REQUESTS:

NUMBER	T1	T2	SITE
1	2	1	2
2	2	1	2
3	2	1	2
4	4	2	4
5	5	2	5
6	5	1	5
7	6	2	6
9	8	6	8
10	8	1	8
11	9	1	9
12	10	1	10
13	10	1	10
14	10	1	10
15	10	7	10

OVERALL MESSAGES STORED IN BUFFER BEFORE DEADLOCK DETECTION

NUMBER	T1	T2	SITE
1	2	1	2
2	2	1	2
3	2	1	2
4	4	2	4
5	5	2	5
6	5	1	5
7	6	2	6
8	7	1	7
9	8	6	8
10	8	1	8
11	9	1	9
12	10	1	10
13	10	1	10
14	10	1	10
15	10	7	10
16	4	1	4
17	5	1	5
18	6	1	6
19	8	2	8
20	10	1	10

ADJACENCY LIST AT SITE 1

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	8	1	
2	0	2	---> 1 1
1	0	5	---> 1
1	0	7	---> 1
1	0	8	---> 1
1	0	9	---> 1
2	0	10	---> 1 1

ADJACENCY LIST AT SITE 2

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	3	1	
3	3	2	---> 1 1 1
1	0	4	---> 2
1	0	5	---> 2
1	0	6	---> 2

ADJACENCY LIST AT SITE 3

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...

ADJACENCY LIST AT SITE 4

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
0	1	2	
2	0	4	---> 2 1

ADJACENCY LIST AT SITE 5

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	2	1	
0	1	2	
3	0	5	---> 2 1

ADJACENCY LIST AT SITE 6

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
0	1	2	
2	1	6	---> 2 1
1	0	8	---> 6

ADJACENCY LIST AT SITE 7

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
1	1	7	---> 1
1	0	10	---> 7

ADJACENCY LIST AT SITE 8

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
0	1	2	
0	1	6	
3	0	8	---> 6 1 2

ADJACENCY LIST AT SITE 9

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	1	1	
1	0	9	---> 1

ADJACENCY LIST AT SITE 10

OUTDEGREE	INDEGREE	TRANSACTION	ADJACENT TO...
0	4	1	
0	1	7	
5	0	10	---> 1 1

NO DEADLOCK EXISTS

BIBLIOGRAPHY

Goldman, B. "Deadlock Detection in Computer Networks." M.I.T. Cambridge Tech. Rep., MIT-LCS TR-185, Sept. 1977.

Gligor, V.D.; and Shattuck, S.H. "On Deadlock Detection in Distributed Systems." IEEE Transaction on Software Engineering vol. SE-6,5 (September, 1980): 435-39.

Harrowitz, E.; and Sahni, S. Fundamentals of Data Structures Using Pascal (Rockville: Computer Science Press, 1983): 272-319.

Menasce, D.; and Muntz, R. "Locking and Deadlock Detection in Distributed Data Base." IEEE Transaction on Software Engineering vol. SE-5 (May, 1979): 195-202.

Peterson, J.; A. Silberschatz. Operating System Concepts (Reading: Addison Wesley, 1984): 260.