8009

DEADLOCK DETECTION PROBLEM IN COMPUTING SYSTEMS :

A Simulation Approach Using a Priority Based

Deadlock Detection Algorithm

by

Sema F. Akgün

B.S. in CMPE., Boğaziçi University, 1987

Submitted to the Institute for Graduate Studies in

Science and Engineering in partial fulfillment of

the requirements for the degree of

Master of Science

in

Computer Engineering

Boğaziçi University

1989

DEADLOCK DETECTION PROBLEM IN COMPUTING SYSTEMS : A Simulation Approach Using a Priority Based Deadlock Detection Algorithm

APPROVED BY

Doç. Dr. Oğuz Tosun (Thesis Supervisor)	[illegible]
Doç. Dr. Selahattin Kuru	[illegible]
Doç. Dr. Ali Rıza Kaylan	[illegible]

DATE OF APPROVAL Oct. 4, 1989

ACKNOWLEDGEMENTS

I am grateful to Dr. Oğuz Tosun for his guidance and help. I would like to thank Dr. Ufuk Çağlayan who proposed the topic of the thesis. Thanks to Dr. Selahattin Kuru, Ali Rıza Kaylan for their comments. Valuable editorial improvements and help of Fedon Kadifeli are gratefully acknowledged. Thanks to A. C. Cem Say, my ideas are deeply influenced by discussions with him. Special thanks go to Gürol Oktuğ.

Sema F. Akgün

ABSTRACT

In this thesis, the deadlock detection problem in computing systems is examined in detail. Deadlock models and some published algorithms on deadlock detection are discussed. A modified priority based algorithm is introduced and some more modifications are offered to make the algorithm correct and more efficient. The final version of the algorithm is simulated for a single-site system. To show the effects of these modifications, the simulation results obtained with modifications are compared with the results obtained without them. It is observed that after the modifications, the system performed better. For further simulation studies, a distributed system model is offered.

ÖZET

Bu tezde bilgisayar sistemlerindeki kilitlenme yakalama sorunu incelenmektedir. Kilitlenme modelleri ve konu hakkında yayınlanmış bazı algoritmalar tanıtılmaktadır. Daha önce düzeltilmiş "Önceliğe Dayalı Bir Kilitlenme Yakalama Algoritması" açıklanmaktadır. Algoritmayı daha verimli ve doğru yapmak için bazı değişiklikler önerilmektedir. Algoritmanın son hali tek işlemcili bir sistem için benzetimlenmektedir. Değişikliklerin etkisini göstermek amacı ile, alınan sonuçlar değişiklikler yapılmadan alınan sonuçlarla karşılaştırılmaktadır. Yapılan değişikliklerle sistemin daha başarılı olduğu gözlenmektedir. Ayrıca ileride yapılacak olan benzetim çalışmaları için dağıtılmış bir sistem modeli önerilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

C_i	current value of local clock at site i
c_i	controller at site i
$DM(R_i)$	data manager of R_i
E	set of events in a system
e_i	event i
F_c	future of cut c in a distributed system
$Id(T_i)$	identity of T_i
P_c	past of cut c in a distributed system
p_i	process i in a single-site system
p_{ij}	process i in site j of a distributed system
R_i	resource of type i
r_i	current number of available units of R_i
seq	sequence of processes
s_j	site j in a distributed system
S_t	global state at time t
T_i	transaction i at a single-site system
t_i	total number of units of R_i
t_{ij}	transaction agent i in site j of a distributed system
$TM(T_i)$	transaction manager of T_i
WFG	wait for graph of a single-site system
WFG_i	wait for graph of site i in a distributed system

Π	set of processes in a system
ρ	set of resources in a system
ρ_c	set of consumable resources in a system
ρ_r	set of reusable resources in a system
Σ	all possible allocation states of all system resources
$\emptyset$	null set
$\preceq$	happened before relation
$\vdash$	reachability relation

I. INTRODUCTION

Deadlock detection is an important problem of multiprogramming environments, in which several processes may compete for a finite number of resources. A process requests resources, and if the resources are not available at that time, it enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other processes which are directly or transitively waiting for them. This situation is called a *deadlock*. In other words, it is a permanent blocking of a set of processes that either compete for system resources or communicate with each other.

If deadlock is ignored in the design state, it must be detected later by some means, and then a process must be terminated and restarted to recover from it. It is also possible that a deadlock, containing all the processes in the system, occurs.

Methods for coping with deadlock fall into three categories. The first policy is *detection and recovery*. Here no action is taken to prevent deadlock from occurring. When a group of deadlocked processes is identified, some of them are terminated in order to break the deadlock. In connection with this policy the selection of the so-called victim is another interesting component. The second policy is *prevention*. It is accomplished by preventing at least one of the conditions, all of which are necessary for deadlock to happen. And the third policy is *avoidance*. This refers to methods that rely on some knowledge of future process behavior to constrain the pattern of resource allocation.

The topic of the thesis is deadlock detection and resolution. First the problem is introduced. Then, survey analysis is done and deadlock detection algorithms in distributed systems are grouped according to the methods they use. A single-site system uses basic and simple methods for deadlock detection because there is no communication problem within a single-site system. After this general study, a deadlock detection algorithm is studied. A situation under which the algorithm is unable to resolve the deadlock is found. Some structural modifications are suggested to make the algorithm more efficient and correct. To show the performance of the algorithm, it is simulated on a single-site system. The simulation results are discussed. Finally a distributed system model is introduced for further simulation study.

The thesis consists of eleven parts (including this part). In the next three parts, the deadlock problem is explained in detail. The fifth part introduces deadlock detection algorithms in distributed systems. In the remaining parts, a priority based deadlock detection algorithm is introduced, its structure is changed, and simulated for a single-site system. The aim of the simulation is to show that the algorithm works under deadlock conditions.

Part II introduces the deadlock problem, gives examples of deadlocks, describes what a general resource graph is, explains the necessary and sufficient conditions for deadlock to happen, and gives information about deadlock policies.

Part III discusses models of deadlock according to resource request model of the processes.

Part IV explains the methods of deadlock detection in distributed systems. It is known that deadlock is important problem in distributed systems too.

Part V makes a classification of distributed deadlock detection algorithms and explains each class. It also gives example algorithms for some classes.

Part VI explains an edge-chasing algorithm that is developed by Sinha and Natarajan [1]*, discusses the errors and the deficiencies of the algorithm detected by Choudhary *et al*[2]. Also some modifications are proposed to improve the algorithm.

Part VII introduces a single-site system simulation using the algorithm explained in Part VI both to test the algorithm and to show the effect of the some system parameters on the performance of the system. For the future studies, it offers a distributed system simulation model.

Part VIII contains conclusion of the thesis.

Appendix A explains two-phase locking protocol in concurrency control.

Appendix B gives the deadlock detection algorithm modified by Choudhary *et al* [2].

---

* References enclosed in brackets refer to the bibliography.

Appendix C contains the listing of the simulation program which is introduced in Part VII.

Bibliography gives a list of references used in this study and cited in the text of the thesis. References not cited are listed separately.

II. DEADLOCK

Deadlock is the *permanent* blocking of a set of processes either competing for system resources or communicating with each other. The problem of deadlock is not unique to the operating system environment. Generalizing our interpretation of resources and processes, we can see that the deadlock problem may be a part of our daily environment.

In this part, first, the problem is introduced. Examples of deadlock are given. Secondly, the general model of a system is explained. Methods of dealing with deadlock are introduced.

2.1. The Deadlock Problem

In this section, we define some terms used in the rest of the chapter, introduce the problem, give examples of deadlock, and explain the characteristics of resource types.

2.1.1. Definition of Deadlock

A computer system may be abstractly represented by a pair of sets (Σ, Π), where

Σ = { All possible allocation states of all system resources }

Π = { Processes }

Each element in Σ represents one possible state in the distribution of the resources. Each process in Π is a function that, for each system state in Σ, maps to another set of states. That state, possibly, can be empty.

For example, let $\Sigma = \{S, T, U, V\}$ and $\Pi = \{P_1, P_2\}$. In this system there are only four possible system states. Suppose the possible actions by the two processes are:

$P_1(S) = \{T, U\}$	$P_2(S) = \{U\}$
$P_1(T) = \varnothing$	$P_2(T) = \{S, V\}$
$P_1(U) = \{V\}$	$P_2(U) = \varnothing$
$P_1(V) = \{U\}$	$P_2(V) = \varnothing$

where, for example, $P_1(S) = \{T, U\}$ means that when P_1 is in state S, it may operate changing system state to T or U. When the range is $\varnothing$ (null set), the process may not operate to change the system state from the given state. A system can be shown graphically, by using nodes for the possible states and arcs for the possible state changes. The above example can be defined by Figure 2.1.

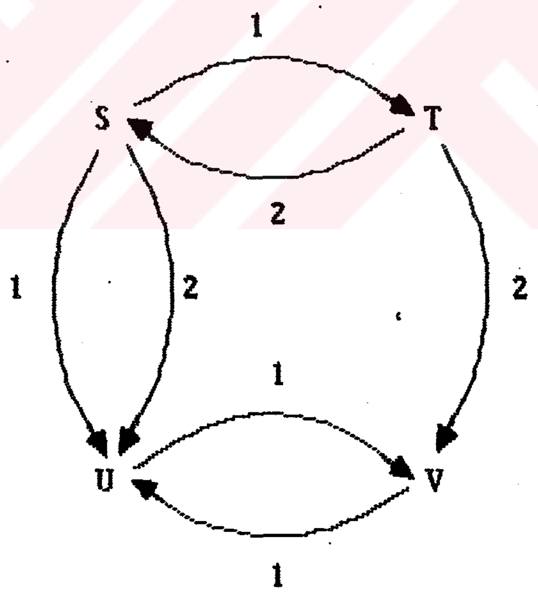


FIGURE 2.1 System states of the example

An operation by process i changes the system state from, say, S to T. It is abbreviated by writing $S \rightarrow i \rightarrow T$. If a sequence of operations by processes i,j,. . .,k is possible ($S \rightarrow i \rightarrow T, T \rightarrow j \rightarrow U, \ldots, V \rightarrow k \rightarrow W$), the sequence is abbreviated by $S \rightarrow * \rightarrow W$.

With these settings, some terms related to deadlock can be defined. A process P_i is ***blocked in state S*** if there exists no T so that $S \rightarrow i \rightarrow T$. In the figure, P_1 is blocked in

With these settings, some terms related to deadlock can be defined. A process P_i is *blocked in state S* if there exists no T so that S→i→T. In the figure, P_1 is blocked in state T because there is no arc labelled 1 starting at node T. Process P_i is *deadlocked in state S* if P_i is blocked in S and, for all states T with S→i→T, P_i is blocked in T. No matter how other processes can change the system state, there will be no opportunity for P_i to perform an operation. In the figure, P_2 is deadlocked in the states U and V. P_1 is not deadlocked in T because T→2→S unblocks P_1. If there is a process P_i deadlocked in S, then S is a *deadlock state*. In the figure U and V are deadlock states. If all processes P_i are deadlocked in S, then S is a *total deadlock state*. There is no total deadlock state in the figure. State S is *secure* (safe state) if it is not a deadlock state and, for any state T reachable from S, T is not a deadlock state.

2.1.2. Examples of Deadlock

Deadlocks can be different from each other. Depending on the number of resources and processes, they can be simple or complicated.

As a first example, consider the two processes competing for disk file D and tape drive T. Deadlock occurs if each process holds one resource and requests the other. Strategies to deal with this kind of problem include imposing constraints on system design so that certain resources are requested in particular order.

As a second example, suppose the main memory space required for activation records of processes is dynamically allocated. Suppose total space consists of 20K bytes and two processes require memory in the following way:

P_1:	P_2:
Request 8K bytes	Requests 7K bytes
Requests 6K bytes	Requests 8K bytes

As in the previous example, if both processes progress their second requests, deadlock occurs. Strategies to cope with such deadlocks include preemption of main memory through paging or requiring processes to declare maximum amount of memory space required in advance.

Consider two communicating processes having the following structure is another example.

```
P1:                          P2:

...                          ...

Receive (P2, M)              Receive (P1, M)

...                          ...

Send (P2, M')                Send (P1, M')
```

Design errors such as these may occur at isolated places in very large programs and may be difficult to detect.

For a fourth example, consider that we have two processes sharing resource R. After some period of time both processes want to hold the resource R exclusively by upgrading their locks. Each process begins to wait for the other one to release the lock.

```
P1:                              P2:

...                              ...

req-shared-lock (R)              req-shared-lock (R)

acquire-shared-lock (R)          acquire-shared-lock (R)

...                              ...

req-exclusive-lock (R)           req-exclusive-lock (R)
```

A solution for this problem is that if there are more than one users of a resource, processes are not allowed to upgrade their locks without releasing the resource first.

In each of these examples, deadlock occurs because processes request resources held by other processes and, at the same time, those processes wait for the resources held by former processes. This is the fundamental characteristics of deadlock.

Deadlock is similar to *starvation*, since each of these involves one or more processes that are permanently blocked and waiting for the availability of the resource. The two, however, are distinctly different phenomena. A deadlocked process waits for resources that will never be released. Starvation occurs when some process waits for a resource that periodically became available, but it is never allocated to that process.

2.1.3. Resource Types

There are two types of resources: *reusable* and *consumable*. Each class has distinct properties that are reflected in the various strategies designed to deal with the deadlock problem.

Reusable resources have fixed total inventory. Additional units are neither created nor destroyed. Units are requested and acquired by processes from a pool of available units and, after use, they are returned to the pool. Examples of reusable resources are processors, I/O channels, main and secondary memory, devices, busses, and information such as files, databases and mutual exclusion semaphores. In the first two and the fourth examples, processes use reusable resources.

Consumable resources have no fixed number of units. Units may be created (produced) or acquired (consumed) by processes. An unblocked producer of a resource may release any number of units. These units immediately become available to the consumer of the resource. An acquired unit ceases to exit. Examples of consumable resources are interrupts, signals, messages and information in I/O buffers.

In general, deadlock may involve any combination of classes of resources. The classes of resources present in any system or subsystem affect the manner in which deadlock problem can be handled.

2.2. The General Model

A general system consists of nonempty sets of processes, Π, and resources, ρ.

$\Pi = \{P_1, \ldots, P_n\}$

$\rho = \{R_1, \ldots, R_m\}$

The set ρ is partitioned into two disjoint sets which are ρ_c and ρ_r, representing consumable and reusable resources. For each resource R_i, the current number of available units of R_i is greater than or equal to zero ($r_i \geq 0$). The total number of each reusable unit is greater than zero ($t_i > 0$). For each consumable resource, there is a nonempty set of processes which produce units for that resource.

2.2.1. General Resource Graph

A particular state of the general resource system model is described by the number of units of each resource that each process requests, the number of units of each reusable resource held by each process, and current available inventory of each process. Each state can be explained by a bipartite digraph (directed-graph).

Nodes of the system are resources and transactions. To distinguish them, square boxes, □, are used to represent processes and circles, 0, to represent resources. For reusable resources, the inventory of the resource is represented by placing small tokens into the circle of the resource. For consumable resources, the tokens represent the current number of available units.

There are three types of edges in the system. *Request edges* (P_i, R_j) are used to connect processes to resources and represent the requests which are not granted yet. *Assignment edges* (R_j, P_i) connect resources to processes indicating that the resource is allocated to the corresponding process. *Producer edges* (R_j, P_i) connect consumable resources to processes that produce them. This edge is the permanent identifier of the producer. In Figure 2.2, the producer edge is shown using a dashed line.

There are some restrictions which a general resource system model should obey. For reusable resources:

(a) The number of assignment edges directed from R_j cannot exceed t_j(total number of resources type j).

(b) At any time the number of available units is $r_j = t_j$ - (number of edges directed from R_j).

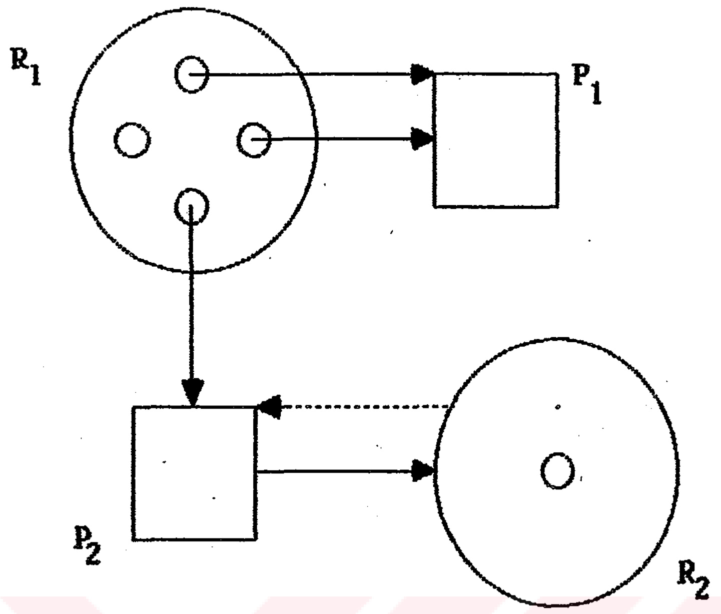


FIGURE 2.2 A general resource graph

(c) For each process P_i, [number of request edges (P_i, R_j)] + [number of assignment edges (R_j, P_i)] $\le t_j$.

For consumable resources:

(a) Edge (R_j, P_i) exists if and only if P_i produces R_j.

(b) The inventory of r_j at any time is constrained only to be nonnegative. This means that systems containing consumable resources may have infinite number of states.

2.2.2. Operations on Resources

In this part, operations which are performed by processes on resources are explained. These operations are *request*, *acquisition*, and *release*.

If process P_i is executable, then it may request any number of resources R_j, R_k, . . . For each request, an edge is inserted, e.g. (P_i, R_j), (P_i, R_k), . . .

If process P_i has a request, for the resource R_j and the number of requested units are not more than current inventory r_j then P_i may acquire the resource. As a result of this, the graph must be modified. Request edge

(P_i, R_j) for a reusable resource becomes (R_j, P_i) indicating an allocation. Each request edge to a consumable resource disappears, simulating the consumption of units by P_i.

Process P_i may release any subset of resource it is holding or produce any number of units of consumable resource. Assignment edges disappear from the graph, but producer edges are permanent. When new units of R_j are produced or released, current inventory of the resource is increased by that amount.

2.2.3. Existence of a Deadlock in System

To check the existence of deadlock in a graph, the graph reduction method can be used. In particular a reduction by a process P_i simulates the acquisition of any outstanding request, the return of any allocated units of a reusable resource, and if P_i is a producer of a consumable resource, the production of sufficient number of units to satisfy all subsequent requests by consumers.

Formally, a graph can be reduced by a nonisolated node, representing an unblocked process, in the following way:

(a) For each resource R_j, delete all edges (P_i, R_j) and if R_j is consumable decrement r_j by the number of deleted request edges,

(b) For each resource R_j, delete all edges (R_j, P_i). If R_j is reusable, then increment r_j by the number of deleted edges. If R_j is consumable, then set r_j to infinity.

A reduction of a graph by a process node P_i may led to the unblocking of another process node P_j, making P_j a candidate for the next reduction. A graph is completely reducible if there exists a sequence of graph reductions that reduces the graph to a set of isolated nodes.

A process P_i is not deadlocked in state S, if there exists a sequence of reductions in the corresponding graph that leaves P_i unblocked.

Another method of deadlock checking is searching for the existence of cycles in the graph. Cycles show that there are some processes waiting for some resources. If deadlock happens there must be a cycle. On the other hand, if there is a cycle the system may or may not be in a deadlock state, depending on the resource request model of the processes. Some resources have multiple instances. A resource with multiple instances is involved in deadlock, iff all the instances of it are involved in cycles.

2.3. Deadlock Policies

Methods of dealing with deadlock fall into three categories. These are detection and recovery, prevention, and avoidance. Each policy has its advantages and disadvantages, and also they are used under different conditions.

2.3.1. Deadlock Detection and Recovery

If a system does not employ a protocol to *prevent* deadlocks, then it needs a *detection* and *recovery* scheme. When a group of deadlocked processes is identified some of them must be terminated (aborted) to resolve the deadlock. Either a deadlock detection algorithm examines the state of the system periodically, or system events may trigger the execution of the algorithm. The process which is selected to be aborted is called victim. The algorithm should select the one whose termination costs the least. Factors that are commonly used to make this determination include:

(a) The amount of effort that has already been invested in the process. This effort will be lost if the transaction aborted.

(b) The cost of aborting the process. This cost generally depends on the number of updates the process has already performed.

(c) The amount of effort it will take to finish executing the process. The scheduler wants to avoid aborting a process that is almost finished. To do this, it must be able to predict the future behavior of processes.

(d) The number of cycles that contain the process. Since aborting a process breaks all cycles that contain it, it is best to abort processes that are part of more than one cycle.

A process can be repeatedly involved in deadlock. In each deadlock, the same process is selected as the victim. It aborts and restarts its execution, only to become a part of deadlock again. To avoid such *cyclic restarts*, the victim selection algorithm should also consider the number of times a process aborted due to deadlock. If it has been aborted too many times, then it should not be a candidate for victim selection, unless all processes involved in deadlock have reached this state.

The thesis is on deadlock detection and resolution. So, this topic is examined in detail in the following parts.

2.3.2. Deadlock Prevention

A second class of deadlock policy is *prevention*. Here the system design prevents entry into a state which leads to deadlock. This is accomplished by denying at least one of the four conditions which are necessary for deadlock to happen:

(a) *Mutual Exclusion*: Processes hold resources exclusively, making them unavailable to other processes.

(b) *Nonpreemption*: Resources are not taken away from a process holding them; only processes can release resources they hold.

(c) *Resource Waiting*: Processes that request unavailable units of resources block until they become available.

(d) *Partial Allocation*: Processes may hold some resources when they are waiting for other resources.

Deadlock is prevented by designing the resource management section of an operating system so that one of the conditions cannot occur. Denying any condition inevitably degrades utilization of the system resources, but it is appropriate in the systems for which deadlock carries a heavy penalty (real-time systems controlling chemical or nuclear processes).

2.3.3. Deadlock Avoidance

Avoidance refers to methods that rely on some knowledge of future process behavior to constrain pattern of resource allocation. Once again degradation in the resource utilization is inevitable. Often, a subset of resources for which deadlock is especially expensive is managed with an avoidance policy.

There are various algorithms which differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information for each process, it is possible to construct an algorithm that ensures, the system will never enter a deadlock state. A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there will never be a circular-wait condition.

Given the concept of a safe state, we can define avoidance algorithms which ensure that the system will never enter an unsafe state . The idea is that the system will always remain in the safe state. Initially the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be immediately granted or not. The request is granted only if it leaves the system in a safe state. As long as the state is safe, the operating system can avoid unsafe states. In an unsafe state, processes are not prevented from requesting resources in such a way that a deadlock occurs.

Note that in this scheme, if a process requests a resource which is currently available, it may still have to wait.

III. MODELS OF DEADLOCK

Depending on the application, resource systems allow a number of different kinds of resource requests. For example, a process may need to access combination of some resources, such as resource A and resource B, resource A or resource B, etc. This part introduces a hierarchy of request models used in the literature, starting from very restricted forms and going to models with no restriction. We also mention about some algorithms which are designed for those models.

In Part II the deadlock problem is studied in detail. The theory can be directly applied to all system resources. Not to complicate the system, we can consider all resources as "single unit reusable" resources. As a consequence, each resource is either not in use or allocated to a single user . This shows that resource nodes are redundant and can be eliminated from reusable resource graph.

In such a graph, edges are different than the ones we used. If R is a resource and P and Q are processes such that (P, R) and (R, Q) are edges in the graph, then the transformed graph will have an edge (P, Q). This new graph is called *wait-for-graph (WFG)* because an edge represents one process waiting for another to release the resource. Since all resources are single unit, a cycle in WFG is the necessary and sufficient condition for a deadlock to exist. For this reason, deadlock detection algorithms are based on finding a cycle in the WFG.

3.1. One-Resource Model

It is the simplest model, in which a process can only make one outstanding request at a time. Finding a deadlock in such a WFG corresponds to finding a cycle in the graph. The outdegree of a node, which is the number of edges leaving a process node, is not more than one.

A very simple algorithm for deadlock detection in the one-resource model appears in Sinha and Natarajan [1]. It is an edge-chasing algorithm in which probes are sent in the direction of the edges of WFG. In the simplest case, a probe consist of a natural number which is unique to the nodes in the graph, node id, and the id of the node which will be the victim in case of deadlock.

The algorithm has very nice features. It is very simple. Exactly one process in the cycle detects the deadlock and simply informs the victim to take necessary actions before being aborted. Spontaneous aborts are allowed and it does not detect phantom deadlocks.

Although the algorithm seems to be correct Choudhary *et al.* [2] show the missing parts of the algorithm and improve some of them without changing the main structure of the algorithm. This is explained in detail in Part VI.

3.2. AND Model

In this model, processes are allowed to request more than one resource at a time and then, they wait all requests to be granted. The nodes of such a system can have outdegree greater than one. The problem of finding the deadlocks is equivalent to finding cycles in the WFG.

Consider the WFG given in Figure 3.1. Node p_{11} has two outstanding resource requests. Because the system is an AND model system, both of the requests must be satisfied.

We define deadlock in the AND model, using the lines of Chandy and Misra [5]. A process p_i is said to be dependent on process p_j if there is a sequence seq = $p_i, p_k, \ldots, p_j$ of processes such that each process in seq is idle and each except the first holds a resource for which the previous process in seq is waiting. We define p_i to be locally dependent on p_j if all the processes in seq belong to the same controller. p_i is deadlocked if it is dependent on itself or a process that is dependent on itself.

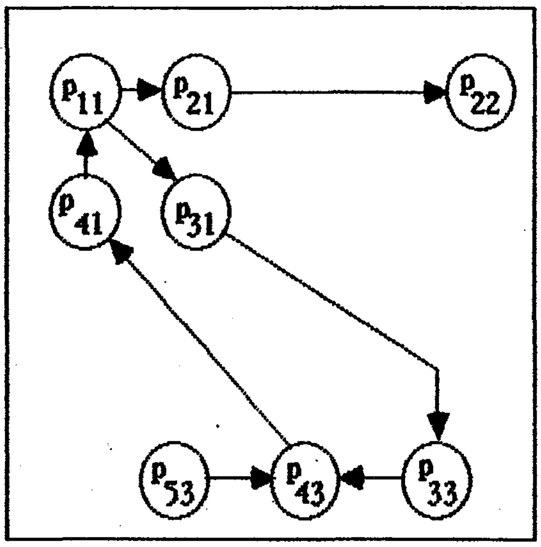


FIGURE 3.1 WFG of an example system

Deadlock detection algorithms for the AND model declare that deadlock exists only if cycles exist. Generally, you cannot say that p_i is deadlocked, if it is not involved in the cycle but waiting p_j which is a part of the cycle. As you can see in Figure 3.1 node p_{53} is not part of a cycle, but because it is waiting for node p_{43} which is part of a cycle, it is also deadlocked. Deadlock in the one-resource system can be defined in the same way, with additional restriction that a transaction can have at most one outstanding request at a time. It is seen that the AND model is a more general form of the one-resource model.

3.3. OR Model

Another model of resource request is the OR model. A request for many resources is satisfied by granting any requested resource. An example of this model can be a read request for a replicated data item. It can be satisfied by reading any copy of it. In the OR model, detection of a cycle is insufficient for deadlock detection. For example in Figure 3.1, there is a cycle in the WFG, but we cannot say that there is a deadlock, because node p_{11} is transitively waiting for node p_{22} which is an active node. We can say that it is a deadlock situation, if all the edges leaving from p_{11} are involved in cycles.

In the OR Model, a *knot* in the WFG indicates existence of deadlock. By definition, a vertex v is in a knot if ∀ w such that w is reachable from v → v is reachable from w. So, no path originating from a knot has "dead ends."

We define a deadlock in an OR model as follows: A process is blocked if no one of its outstanding requests is granted. Each *blocked* process has a set of processes, called its *dependent set*. A set S of processes is deadlocked if all processes in S are permanently blocked. More clearly, a set S of processes is deadlocked if

(a) all processes in S are blocked,

(b) the dependent set of every process in S is a subset of S, and

(c) there are no grant messages in transit between processes in S.

Presence of a deadlocked set of processes is equivalent to the existence of a knot in the WFG. Therefore, deadlock detection in the OR model can be reduced to detecting knots in the WFG. A blocked process p is deadlocked if p is in a knot or p can reach only deadlocked processes.

AND model deadlock detection can be simulated by repeated applications of the OR model deadlock computations, where each invocation operates on a subgraph of the AND model WFG according to Knapp [6]. But it becomes a very inefficient method to handle deadlocks.

3.4. AND-OR Model

As the name implies, it is the generalization of the previous two models. Requests can be combinations of the ones in AND and OR models. For example, (a and (b or c)) may be a request of this model. For this model we can use repeated application of the OR model deadlock computation as explained in the previous section. As explained before, using the algorithm in this way is very inefficient.

A better deadlock detection method for this model is developed by Hermann and Chandy [7]. The algorithm is explained in Section 5.3.

3.5. $\binom{n}{k}$ Model

The $\binom{n}{k}$ model allows the specification of requests to obtain any k available resources out of a pool of size n. It is the generalization of the AND-OR model. So every request in the $\binom{n}{k}$ model can be expressed using the AND-OR model.

To find a deadlock in such a model, the requesting process should be checked. If out of k requests, more than (n-k) are involved in cycles, it is said, the process is deadlocked.

An example algorithm for this model is Bracha and Toueg's Algorithm [8]. A transaction can have as a request an arbitrary and-or combination of $\binom{n}{k}$ requests.

A process becomes blocked, when it issues an $\binom{n}{k}$ request. It does so by sending out n request messages. It becomes executing again when it receives k grant messages. In this case, it sends *relinquish* messages to the remaining (n-k) processes, informing them that the edge created by sending the request message no longer exists.

Gafni [9] suggests improvements to this algorithm, without giving any correctness proof or simulation results.

3.6. Unrestricted Model

Initially no resource request structure is assumed. Instead the stability of the deadlock is the only assumption made meaning that deadlocks cannot go away by themselves; we must detect and resolve them.

The advantage of this model is that it works under every resource request model. But, because it is designed considering all resource request structures, it has a lot of overheads. So, it is preferred just for the systems in which resource requests do not have a general structure.

However, in the context of deadlock detection in computing systems, these algorithms seem to be of more theoretical value. Since the fact that no further assumptions are made about the underlying structure of the system, computation leads to a great deal of overhead that can be avoided in algorithms for the simpler models.

IV. DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

In general, a distributed system consists of a number of sites, each of which is actually a centralized system. This brings additional problems to the system such as dealing with replicated data, single process executing in parallel at different sites, etc. As it can be imagined, it is more difficult to detect the deadlock in a distributed system. This is because each site has only a local view of the whole system.

Both resource (in this case, we refer to a device such as disk, tape, etc.) and communication deadlocks can be distributed. In distributed systems, processes that access nonlocal data, migrate to other sites creating a subprocess at that site. Subprocesses may run concurrently with each other. The originating process is blocked until all subprocesses terminate. A communication deadlock can occur, if a process in replicated database requests the value of some nonlocal data item and is blocked until one of the sites that hold a copy of this data responds.

4.1. A Brief Introduction to Concurrency Problem

A nice place to see a group of resources is a database. In a database, we can call each data item a resource. When concurrency or multiprogramming is allowed, a mechanism must be developed to control the access of processes to data items. This mechanism is called *concurrency control* mechanism. The proper definition of concurrency control is given using the terms of Bernstein *et al.* [3]. Concurrency control deals with the problem of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other.

The main component of the systems that offer concurrent processing is the *transaction*. A transaction is defined as a process that accesses a shared database.

When two or more transactions execute concurrently, their operations on the resources in database are performed in interleaved fashion. Such an interleaving can cause the resources to be in an inconsistent state. So, not to face with such situations, resource requests of processes working concurrently are controlled, before granting them.

When a transaction (process) successfully terminates, the transaction is said to be *committed.* Successful termination means that the process acquired all the resources it requested and finished. A process is *aborted,* if its execution is terminated by the operating system, before it completes.

The main difficulty in deadlock detection in distributed systems lies in the efficient construction of the global WFG. Construction of the global WFG is required to detect global deadlocks. Even though each WFG_i is acyclic, the global WFG may contain a cycle. To discover such deadlocks, all sites must put their local WFG's together.

In distributed systems, if a process requests a resource at a remote site, a *remote agent* is created at the remote site to implement the actual request, access, and release of the resource.

In distributed deadlock detection algorithms, usually database objects are used as resources.

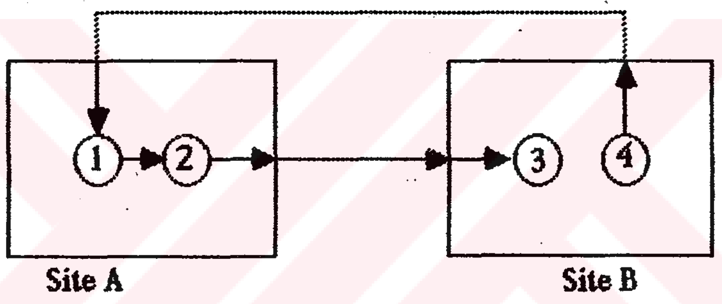
4.2. Centralized Deadlock Detection

In centralized deadlock detection, one site, the central detector, is responsible for the detection of global deadlocks. There are two basic approaches. In a *periodic* deadlock detection, as the name implies, various sites are polled periodically to check the occurrence of any deadlock. In the *continuous* case, each local site informs the central detector when an edge inserted to or deleted from local WFG.

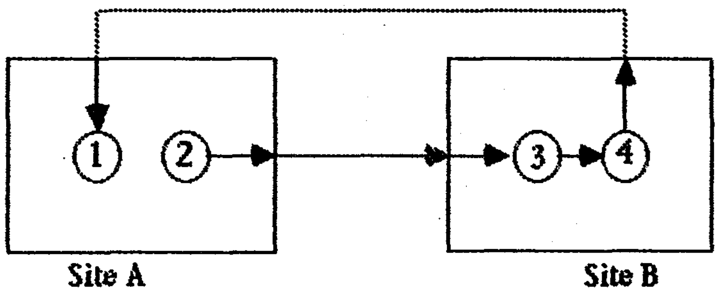
Once the centralized detector finds a deadlock, it selects a victim by using the rules explained in Section 2.3.1 . Therefore in addition to WFG's, the centralized detector needs some information about transactions to make a good victim selection.

Transferring this information creates more message traffic in the system (makes the traffic heavier).

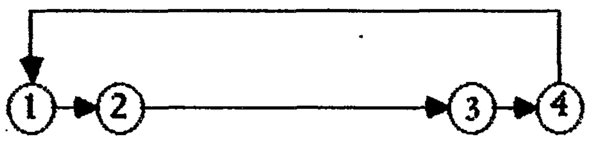
Although centralized deadlock detection is conceptually simple, there are some problems involved in the approach. The first problem is ***phantom (false) deadlocks.*** Assume a continuous deadlock detection and suppose initial states of site A and site B are as shown in the Figure 4.1(a). Initially, process 1 requests a resource held by process 2, process 2 and process 4 respectively wait for process 3 and process 1.After a while process 3 requests the resource held by process 4. The local WFG's of site A and site B are sent to the central detector–Figure 4.1(c). At that moment process1 releases its resource request–Figure 4.1(b). But before the new WFG is sent, the central detector detects a deadlock, but it is a false deadlock.



(a) Initial system state



(b) Subsequent system state



(c) Phantom deadlock

FIGURE 4.1 Formation of phantom deadlock

To prevent the detection of such a false deadlock, different concurrency control algorithms such as two-phase locking can be used. For more information about two phase locking refer to Appendix A. In case of two-phase locking, phantom deadlocks again occur when a process that was involved in deadlock, spontaneously aborts.

A second problem with centralized deadlock detection is related to the high volume of message traffic between the local sites and the central site. The lines leading towards the central detector can be bottlenecked. If continuous checking is used, too much overhead is encountered. The tradeoff is between rapid detection of deadlocks and reduced message traffic. There are several variations of periodic deadlock detection to reduce the number of messages required for deadlock detection.

A third problem is their vulnerability to failure of the central site, causing failure of the entire system. Such faults result in long delays until a new central agent is determined and supplied with up-to-date WFG information. One method for the solution of this problem is to provide a backup central site. But this solution brings other side problems with it, such as its cost, need for backup time, etc.

One reason for the popularity of the centralized deadlock detection methods is its conceptual simplicity. Moreover, some practical problems, such as removal of false deadlocks, are easily solved. For example, whenever a global deadlock is detected, the central controller can reconstruct the deadlock cycle using new information received from local controllers. If the deadlock cycle remains, then the deadlock is a genuine deadlock.

4.3. Hierarchical Deadlock Detection

The centralized deadlock detection requires that all information to construct the global WFG must be requested by one site and kept in that site. Hierarchical deadlock detection is in between centralized and distributed deadlock detection.

As in the centralized approach each site maintains its local WFG. In contrast to the centralized approach, the global WFG is distributed over a number of different deadlock detectors. These controllers are organized in a tree, where each leaf contains the local WFG of a single site. A site is reported to its parent deadlock detector. Each parent deadlock detector is in charge of detecting and resolving any deadlock that is local to itself and the set of its descendent sites. The process terminates at some central deadlock detector.

Hierarchical deadlock detection partially solves the problem of high cost of constructing the global WFG. But it is still vulnerable to failure of central deadlock detectors and the phantom deadlock problem is not eliminated.

4.4. Distributed Deadlock Detection

Using this method, any site can detect the deadlock given enough information. Distributed deadlock detection has been the subject of intensive research in recent years and a lot of algorithms have been published on the subject. In this part, we will see why distributed deadlock detection is needed.

Most WFG cycles are of length two. Let's see why it is so. Suppose we start with all active processes, so the WFG has no edges. When processes become blocked, edges are added to the graph. Early in the execution, more processes are not blocked, so new added edges are from waiting processes to the ones which are actually holding the resource (an unblocked process). As more processes become blocked, there is more chance that a process P_i will be blocked by a lock owned by process P_j which is also blocked, creating a path of length two.

Suppose all processes access to the same number of data items with equal probability. Then on the average, blocked and unblocked processes hold the same number of locks. All processes are equally likely to block an unblocked process. Then the probability that an edge creates a path of length two (three, four, etc) is proportional to the number of processes that are on the ends of paths of length one (two, three, etc). Because initially there is no edge, short paths must dominate. So an edge that completes a cycle has higher chance to wait a process which is on the end of a shorter path. Therefore most WFG cycles are of the length two.

Because most WFG cycles are of length two, many times only two sites will be involved in deadlock. So we do not need to construct the global WFG to see the existence of a deadlock. In this case trying to construct a global WFG, such as in centralized deadlock detection, will be both useless, and time consuming. Communication of only those sites which are involved in the deadlock is enough to detect the deadlock. In this way, deadlock is detected faster without causing unnecessary communication.

4.5. Lock Granularity

The choice of granularity of the database (when data files are used as resources), represents a tradeoff between increased concurrency and system overhead. Finer granularity, at the record or field level, provides more opportunity for concurrency, but more locks to be dealt with. It may be desirable to allow different objects with different granularities, such as a record, a disk page, or an entire file, making the system more complex. This method usually called ***multigranularity locking***.

In multigranularity locking protocols, deadlock can occur for more than one reason. First, a transaction that obtains too many locks on data items of small granularity wants to increase the granularity of its subsequent lock requests. Another problem arises when granularity of the resources is organized as a tree. In this case a locking protocol may require a transaction that wants to lock some set of granules to lock a majority of parents of these granules first. If two transactions happen to try locking the same set of granules, they may reach to a level where both hold locks on exactly half of the parents of the set, so none of them succeed.

4.6. The Resource Model

To study deadlock detection algorithms for distributed systems, the model of Menasce and Muntz [4] is going to be used. According to the model, a distributed system consists of a collection of N sites, $s_1, s_2, \ldots, s_N$, connected by a communication network. The network is assumed to be fully connected. It is also said that the communication network is connected using star topology. Each site is a centralized system that stores some portion of the resources. Data objects in the databases are accepted as resources in the distributed system. There are M transactions, $T_1, T_2, \ldots, T_M$ running on distributed data. A transaction sends *resource requests* to a transaction manager (TM). There is one controller (c_i) for each site (s_i). A transaction is blocked from the time it requests a resource until the acquisition of the resource. A transaction can request a resource which is residing at a remote site. A distributed transaction T_i implemented by *transaction agents* t_{ij}, each of which is the local agent for transaction T_i at site s_j. In case transaction agent t_{ij} requests a resource which is controlled by controller c_m, controller c_j transmits the request to agent t_{im} via controller c_m. When t_{im} acquires the resource it sends a message to t_{ij} via c_m. As it can be seen, intersite requests are always between two agents of the same transaction.

When agents in transaction T_i no longer need a resource controlled by c_m, they communicate with agent t_{im}, which is responsible for releasing the resource. It is assumed that messages sent between two sites arrive sequentially and in a finite time. And also it is assumed that if a single transaction runs by itself in the distributed system, it will terminate in a finite time and deadlock does not arise.

A transaction agent is said to be *idle* if it is waiting to acquire a resource, otherwise, it is *executing*. If an agent never acquires a requested resource, it is permanently idle.

In part V, this model is taken into account when explaining some algorithms.

V. DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

The distributed deadlock detection algorithms in the literature come from four different classes which are path-pushing, edge-chasing, diffusing computations, and global state detection. Each class has its advantages and drawbacks.

The correctness of a deadlock detection algorithm depends on two conditions without looking at the class which it belongs to. First, every deadlock must be detected eventually. Second, if a deadlock is detected it must exist. This condition means that there should not be incorrectly detected deadlocks (phantom deadlocks) because of out-of-date information. But in case of spontaneous aborts no algorithm can guarantee to detect only genuine deadlocks.

5.1. Path-Pushing Algorithms

The basic idea under this class of algorithms is to build some simplified form of global WFG at each site. For this purpose, it is allowed that all sites can exchange deadlock information without causing too much message traffic. Using path-pushing, each site looks for cycles in its local WFG and lists all paths in the graph. It selectively sends some portions of paths to other sites that may need them to find cycles. When a site receives a path, it adds the edges of the received path to its local WFG, and checks for cycles. If there is no cycle, then the paths that neither the sender nor the receiver have seen before are listed and sent to other sites which may have more edges to add to these paths. The deadlock is detected by the site that adds the final edge to the path making it a cycle. And then, it is reported to the other sites involved in the cycle. If cycle lengths are short, this method is better than centralized deadlock detection.

Path-pushing is a nice method, if every site knows where to send its paths. The best method is sending them to all sites, if you want deadlock to be detected faster.

Although it makes detection faster, it causes heavy traffic in the system. Using this approach, every site will end up detecting the deadlock, which is more than necessary. And also, two or more sites that detect the deadlock might choose different victims.

To reduce the traffic and still make enough detections another method is developed. Suppose that

(a) each transaction, T_i, has a unique identification, *$Id(T_i)$*, and

(b) Ids are totally ordered.

In every cycle, at least one path $T_i \rightarrow \ldots \rightarrow T_j$ has $Id(T_i) < Id(T_j)$. If we only send around the paths that have this property, we still find every cycle by reducing the number of transferred path in the system. So, when a site produces new paths, it sends them to sites having this property.

One important point on the subject is that many path-pushing algorithms are found to be incorrect. For example Gligor and Shattuck [10] show that the algorithm developed by Menasce and Muntz [4] is incorrect. Another example is Obermarck's algorithm [11]. It is thought that one reason of such incorrect algorithm development is that at that time the notion of snapshots and consistent global sites in asynchronous systems were not well understood.

5.2. Edge-Chasing Algorithms

The existence of a cycle in a distributed WFG can be checked by sending special messages called *probes* along the edges of the graph. Probes are distinct from resource request and grant messages and are only used for the detection of a cycle in the system. When the initiator of a probe receives that probe, it knows that there is a cycle in the system and it is on the cycle. Then deadlock resolution is initiated.

Only blocked processes propagate the probe along their outgoing edges. Executing processes simply discard probes or do not put the probes into operation, depending on the algorithm.

An edge-chasing algorithm which is developed by Sinha and Natarajan [1] and modified by Choudhary *et al.* [2], is examined and improved in the following parts.

5.3. Diffusing Computations

In this class, the basic idea is a *diffusing computation* which is activated by a transaction manager that suspects a deadlock. If this computation terminates, the initiator declares deadlock. The characteristic feature of the superimposed computation in the case of distributed deadlock detection is that the global WFG is implicitly reflected in the computation. The actual WFG is never built explicitly. The diffusing computation expands by sending *query* messages and shrinks by receiving *replies*. These messages are distinct from request and grant messages. When a diffusing computation shrinks back to its originator it terminates.

Nodes different from the root are called *internal* nodes. Each node in the diffusing computation has an initial state called the *neutral* state. The root sends queries to its successors to start diffusing computation. After receiving the first query, a node leaves the neutral state and becomes active. This query is called the *engaging query* for that node. The process that sent the engaging query is called the *engager* of that node. The edge along which the query is sent is called the *engagement edge* of the node.

After receiving engaging query, an internal node can send queries to its successors, and also send replies to its predecessors and receive replies from its successors. Queries travel in the direction of edges and replies travel in the opposite direction.

The difference between the number of queries and replies sent over an edge is called the *deficit* of this edge. The deficit of an edge is greater than or equal to zero.

Now we can define that the neutral state of a node is the state in which the deficits of all edges are zero. The diffusing computation terminates when the root returns to its neutral state. A node sends back its engaging reply only after it has received replies from each query it has sent.

We say that diffusing computation has terminated if and only if all internal nodes are in their neutral state, and also the root returns to its neutral state.

In general this approach results in shorter messages and less deadlock detection overhead as compared to path-pushing algorithms.

If we examine Hermann and Chandy's AND-OR model algorithm [7] which uses diffusing computations, we see that it is a *tree* computation. A tree computation consists of a hierarchy of diffusing computations. Transaction agents are mapped to the processes in the following manner: A process may have an AND request or an OR request; an AND-OR request issued by some transaction agent is mapped to a tree of processes. The mapping is a representation of the AND-OR request in a regular form. Figure 5.1 shows an example of this mapping. Processes like p'_1 are AND processes, and the others are OR processes.

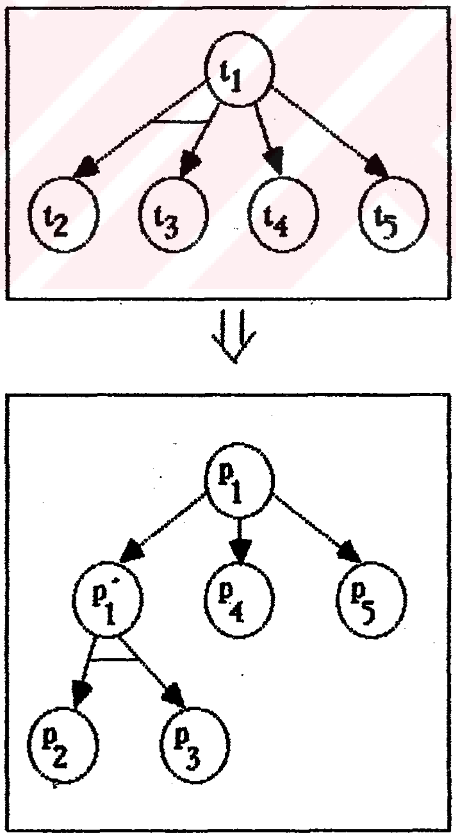


FIGURE 5.1 Mapping transaction agents to processes

When a grant message is received an edge in the WFG disappears. For the receiving blocked process:

(a) Either no outgoing edges remain, and the process becomes active,

(b) or if outgoing edges remain, there are two possibilities: If it is an AND process it remains blocked. If it is an OR request, all outgoing edges disappear and the process becomes active.

The main idea is that any time a diffusing computation reaches a blocked OR process, the diffusing computation is propagated to the dependent set of this process; if the engaged process is blocked AND process, it initiates a separate tree computation for each outgoing edge. In order to start a deadlock computation, an initiating process sends a query to the process that is suspected of deadlock. A tree computation terminates when its initiator receives a reply from the suspected process.

According to the definition, a blocked process p is deadlocked if:

(a) Either p is an AND process and will never receive a grant for at least one of the requested resources,

(b) or p is an OR process, and will never receive a grant message.

Queries have the form *query(seq, k)* where seq is the sequence of processes and k is the sender of the query. If an engaging query(seq, m) arrives at a blocked AND process p_k, a new set of computations is initiated by p_k. And, after appending its outgoing process p_k sends to all outgoing edges of the WFG. If a blocked OR process receives an engaging query, it propagates the query to all processes in its dependent set. These actions are referred as *extension.*

If query(seq, m) is not engaging and receiving process p_k has been blocked, a reply(seq, k) is sent to the sender process. This action is called *reflection.*

When a reply(seq, m) is received by an AND process, it sends its engaging reply back, if it has been continuously blocked from the time it received the engaging query. An OR process sends back its engaging reply when it receives replies from all the elements of its dependent set, and if it has not been executed since it received the engaging query. These actions are called *collision.*

To keep track of the queries sent and the replies received by each process, two different message lists are used. These are incoming query list *IQ-list* and outgoing query list *OQ-list*. Those lists are updated when a new query is received or sent. The important point is the receipt of a grant message.

A deadlock computation is started by some controller, creating a process called *initiator*. Initiator sends a query to the process which is checked for deadlock. And a tree computation starts. A tree computation terminates iff for every i and j, query(seq, i) is sent to p_j, and reply(seq,j) arrives at p_i with no grants within this time interval.

5.4. Global State Detection

An important point here is having a consistent global state without freezing the *underlying computations*. Underlying computations can be considered as the system, processes, transaction agents, and transaction managers.

Events in the system are sending and receipt of messages. The set of events in the system is denoted by *E*. The *local state* of a process p consists of the history of all events occurred on p. Using Lamport's lines [12], Knapp [6] makes a definition of partial ordering. Let e_1 and $e_2 \in E$. Then $e_1 \le e_2$ (e_1 happened before e_2) if either

(a) e_1 and e_2 are both on the same process p, and e_1 occurred earlier in p than e_2;

(b) e_1 is a send event and e_2 is the corresponding receive;

(c) $(\exists e' : e' \in E : e_1 \le e' \wedge e' \le e_2)$.

The first condition says that events in a single process are totally ordered. The second condition implies that each message is received after it is sent. And according to the third condition, we can say that ordering is transitive. Since an event cannot occur before itself, partial ordering is irreflexive.

We can represent the history of a system and its happened-before relation by a diagram in Figure 5.2. The dots represent the events and the horizontal lines are the time axes of the processes.

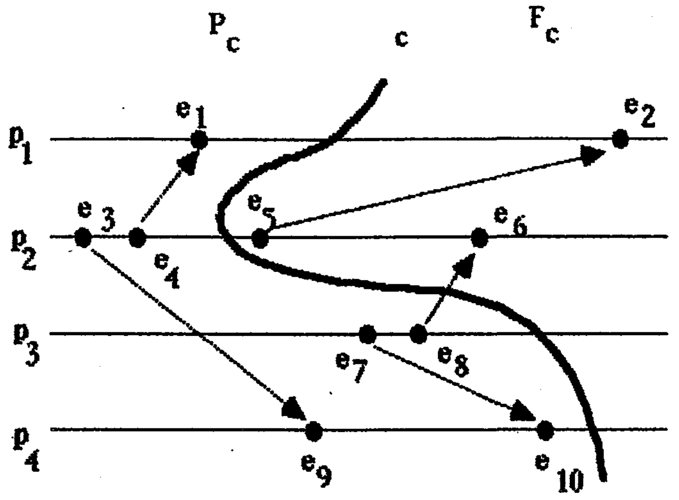


FIGURE 5.2 A cut of a distributed system

The following formalization is from Chandy and Lamport [13]. A *cut* c of E is a partition of E into two sets P_c and F_c, meaning the past and future of c. A cut is *consistent* if F_c is closed under $\leq$. A consistent cut defines a consistent state. It can be said that consistent cuts are the ones that do not contain a send event in the future with the corresponding receive event in the past.

A special type of consistent state is S_t which is the global state at time t. S_t is a purely theoretical construct that cannot be observed, because it is impossible. In contrast, consistent states can be obtained within the system. We can extent the relation $\leq$ to consistent states as follows: Let S_1, S_2 be consistent states. Then $S_1 \leq S_2$, if the past of S_1 is a subset of the past of S_2.

A reachability relation, $\vdash$, is defined between the states. Let S be a consistent state and $e \in E$, such that $P_s \cap \{e\}$ defines a consistent state S'. Then $S \vdash^{e} S'$ denotes that S' is reachable from S. If there is a sequence of events, σ, and if we can reach from state S to S' by following those events , we can write $S \vdash^{\sigma} S'$. Chandy and Lamport show that S $\leq$ S' implies ($\exists$ *schedule* $\sigma :: S \vdash^{\sigma} S'$).

In deadlock detection, the state of a system can be identified by the WFG, and for schedules we can consider sequences of information. A transaction is deadlocked, if it is deadlocked in WFG_t, WFG at time t. And we can also say that if a transaction is deadlocked in WFG, it is also deadlocked in WFG', under the condition that WFG $\leq$ WFG'. This is the main point on which deadlock detection algorithms can be based.

Chandy and Lamport [13] show how to obtain a consistent global state of a distributed system. A consistent global state in this way is called a *snapshot* of the system.

VI. STUDY ON AN EDGE-CHASING ALGORITHM

In this part, first we examine the original algorithm of Sinha and Natarajan [1]. Then, the modified version of the algorithm by Choudhary *et al.* [2] is discussed. Finally, new modifications and structural changes are offered to make the algorithm more efficient and correct. In this part, terms "transaction" and "process" are used interchangeably.

6.1. A Priority Based Distributed Deadlock Detection Algorithm by Sinha and Natarajan

The deadlock detection scheme presented by Sinha and Natarajan does not construct any WFG, but follows the edges of the graph to search for a cycle. It is assumed that each transaction is assigned a priority in such a way that according to priorities all transactions are totally ordered. When a transaction waits for a data item which is locked by a lower priority transaction, an *antagonistic* conflict occurs. If an antagonistic conflict occurs for a data item, the waiting transaction suspects from deadlock and initiates a message to find cycles. If the message comes back to the initiator, a deadlock cycle is detected.

6.1.1. The Distributed Resource Model

In the distributed system, each site has a system-wide unique identifier, called *site_id* in short. And each site communicates through messages.

It is assumed that all messages sent arrive at their destinations in finite time and also messages are error-free. The site-to-site communication is pipelined, meaning that messages arrive in the order they are sent.

Within a site, there are several processes and resources (*data items*). Every process has a system-wide unique name, called *process_id*. To access one or more *data items* (resources), which may be distributed over several sites, a user creates a transaction process at the local site. A transaction process coordinates actions on all data items participating in the transaction and preserves the consistency of the resources.

Data items are passive entities that represent some accessible piece of information. Each data item is controlled by a *data manager*. If a transaction wants to operate on a data item, it must send a request to the corresponding data manager. Locking and unlocking of data items are performed via data managers. A data item can be in one of two modes: *free* or *exclusive* (no shared access is allowed). Data manager grants the data item to requesting transaction, if the corresponding data item is free. Otherwise, the lock request is inserted in a queue, called *request_Q*, and a transaction in the request_Q is called the *requester* of the data item. A transaction which has locked a data item is called the *holder* of the data item.

Transactions can be in one of two states: *active* or *waiting*. If a transaction is in a request_Q of a data manager, it is in waiting state, otherwise it is active. The state of a transaction changes from waiting to active when the data manager schedules its pending lock request. In case of state changes, data managers inform the transactions about the changes.

Each transaction is assigned a priority in such a way that priorities of all transactions are *totally ordered*. To assign priorities to transactions, *timestamp* mechanism is used. When a transaction is initiated, it is assigned a unique timestamp. So the transaction with the least timestamp value has the highest priority. This condition implies that the oldest transaction within a site has the highest priority. If transactions are created at different sites, first, the priorities of corresponding sites are compared to decide which transaction has higher priority.

A timestamp generated by a site whose site-id is i for a transaction is a pair (C, i) where C is the current value of the local clock. Greater than, >, and less than, <, relations for timestamps are defined as follows:

Let $t_1 = (C_1, i1)$ and $t_2 = (C_2, i2)$ be two timestamps. Then

$t_1 > t_2$ iff $C_1 > C_2$ or ($C_1 = C_2$ and $i1 > i2$);

$t_1 < t_2$ iff $C_1 < C_2$ or ($C_1 = C_2$ and $i1 < i2$).

Transactions use two-phase locking protocol, while making their resource requests.

6.1.2. Distributed Deadlock Detection

Deadlock is detected by circulating a message, called *probe*, through the deadlock cycle. The occurrence of an antagonistic conflict for a data item triggers the initiation of a probe. A probe is an ordered pair (*initiator, junior*), where the initiator denotes the requester which is confronted with the antagonistic conflict. A junior denotes the transaction whose priority is the least among transactions on the cycle.

A data manager sends the probe only to the holder of the data item. Transactions send probes only to resources they are waiting for. Transactions and data managers cannot communicate among themselves for the purpose of deadlock detection.

The basic detection algorithm has three steps.

(a) A data manager initiates a probe in the following two situations: The first situation is when the data item is locked by a transaction and there is an antagonistic conflict. The second situation is when a holder releases the data item, the manager schedules a waiting lock request and there are other lock requests for which the priority of requester>the priority of new_holder.

When a data manager initiates a probe it sets

initiator := requester;
junior := holder.

(b) Each transaction maintains a queue, called *probe_Q*, where it stores all the probes it has received. So, the probe_Q of a transaction contains the information on the transactions that are directly of transitively waiting for it.

When transactions enter the second phase of two phase locking protocol they are never involved in deadlock so they can discard their probe_Q and ignore any probe or message which is related to deadlock detection.

When a transaction requests a data item and waits for it to be granted, it goes from active to wait state and transmits a copy of each probe stored in its probe_Q to the corresponding data manager.

When a transaction T receives probe(initiator,junior), it performs the following.

```
if junior>T
then
 junior :=T;
save the probe in the probe_Q;
if T is in the waiting state
then
  transmit a copy of the probe to the data manager where it is waiting;
```

(c) When a data manager receives probe(initiator,junior) from one of its requesters, it performs the following.

```
if holder>initiator
then
  discard the probe
else
  if holder<initiator
  then
    propagate the probe to the holder
  else declare deadlock and initiate deadlock resolution;
```

When the deadlock is detected, the detecting data manager has the identities of the highest and the lowest priority transactions. Junior (lowest priority transaction) is chosen to be aborted.

6.1.3. Deadlock Resolution

Resolution consists of three steps.

(a) The data manager sends an abort message to the victim which is junior of the probe. The identity of the initiator is also sent in the message.

On receiving an abort message, the victim initiates a message, *clean(victim,initiator)*, sends it to the data manager of the resource that it is waiting for and enters the *abort* phase. Sinha and Natarajan thinks that probe_Q's of transactions, from initiator to victim in the direction of probe traversal, will not contain any probe having victim either junior or initiator. So, there is no need for the clean message to traverse in that part.

In the abort phase, the victim releases all the locks it owns, and withdraws its pending lock requests, and aborts. During this phase, it discards any message it receives.

(b) When a data manager receives a clean message, the message is propagated to the holder of the resource.

(c) On receiving clean(victim,initiator), transaction T performs the following operations.

```
purge from the probe_Q every probe that has the victim as its junior
or initiator;
if T is in waiting state
then
  if T=initiator
  then
    discard the clean message
  else
    propagate the clean message to the data manager where it is waiting
else
  discard the clean message;
```

After cleaning up their probe_Q's, transactions on the broken cycle keep the remaining probes in their probe_Q for the detection of later deadlocks.

6.2. Errors and Deficiencies Detected by Choudhary *et al.*

It is detected by Choudhary *et al.* [2] that the algorithm of Sinha and Natarajan [1] either fails to detect deadlocks or report deadlocks which do not exist in many situations. They proposed a modified version of the algorithm. Their full algorithm can be found in Appendix B. In the following subsections these errors and deficiencies are discussed.

6.2.1. Undetected Deadlocks

Consider the situation shown in Figure 6.1(a). Assume that DM(X_1) initiated a probe (T_1, T_5) and propagated to T_5, then to T_4 and finally to T_3. These transactions keep the probe in their probe_Q's. Now suppose T_3 commits and releases its locks. If T_2 is first in the request_Q of DM(X_3), it will acquire the lock. This situation is shown in Figure 6.1(b). Now, if T_2 requests a resource which is held by T_1, as shown in Figure 6.1(c) using a bold line, a deadlock cycle will be formed. Using the original algorithm this cycle cannot be detected, because the only probe that can detect the cycle is probe (T_1, T_5) which will never be propagated to T_1.

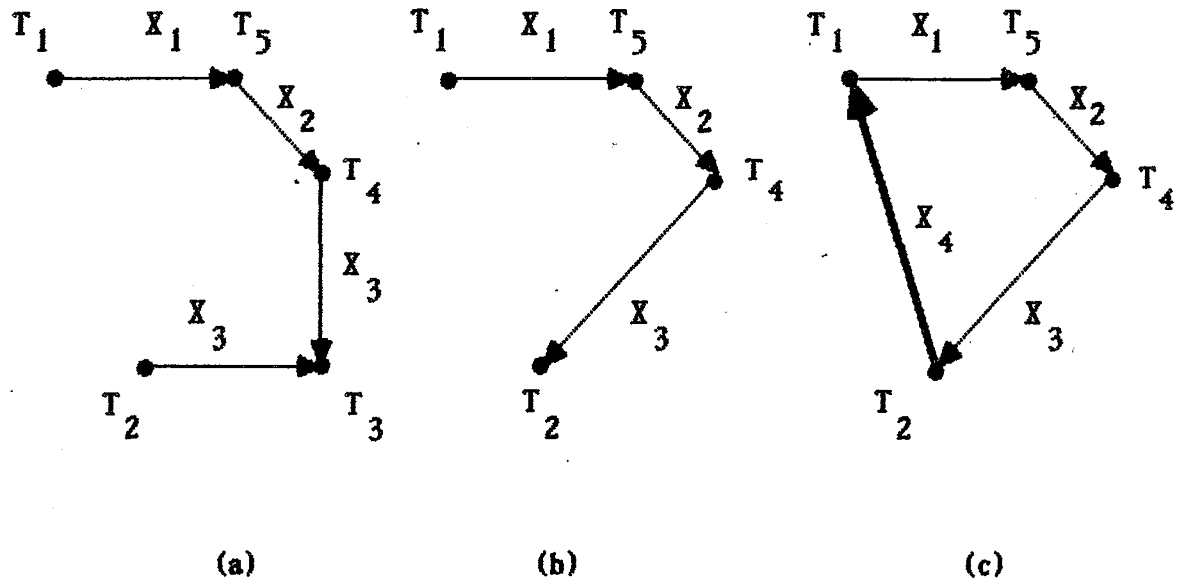


FIGURE 6.1 An undetected deadlock

According to Choudhary *et al.* [2], the following extension must be added to the original algorithm. When a transaction completes or aborts, it releases the resources that it is holding. The data manager associated with each released resource assigns the resource to some transaction waiting in the request_Q (if one exists). Each data manager then requests from all remaining transactions waiting in the request_Q to transmit their complete probe_Q's to itself. The data manager forwards each received probe to the new_holder of the resource for which initiator exceeds the priority of the new_holder.

If the algorithm is modified as suggested above, when transaction T_2 is granted to resource X_3, transaction T_4 transfers a copy of each probe in its probe_Q to data manager DM(X_3). Because the initiator of the probe (T_1, T_5) has higher priority than transaction T_2, that probe is sent to transaction T_2 by the data manager. When transaction T_2 requests a resource which is held by transaction T_1, the probe (T_1, T_5) id is transferred to transaction T_1, which detects the deadlock cycle.

6.2.2. False Deadlocks

Other missing part of the original algorithm is that it detects false deadlocks. False deadlock detections depend on external probes and old probes left in the probe_Q's. Now, we will examine each situation separately.

(A) *False Deadlock Due to External Probes.* Consider the case shown in Figure 6.2(a). Transactions T_1 and T_2 are holding resources X_1 and X_4 respectively, and T_4 is holding resources X_2 and X_3. In addition T_1, T_2 and T_4 have requested resources X_3, X_2 and X_4, respectively. T_1's probe (T_1, T_4) is stored in the probe_Q's of T_2 and T_4. When the deadlock cycle is detected by T_2, T_4 is selected as victim and aborted. A clean message is initiated in the cycle to remove probes that contain T_4. According to the algorithm the clean message is discarded by the initiator, T_2, considering that T_2 should not have any probe in its probe_Q containing the victim, T_4, as its initiator or junior, because T_2 is the highest priority transaction on the deadlock cycle. This argument is only valid when there is no transaction in the system waiting transitively a transaction on a deadlock cycle. In the example, although transaction T_4 is aborted, probe (T_1, T_4) remains in the probe_Q of T_2. Later, in Figure 6.2(c), when T_2 requests a resource held by T_1, the probes, which come from probe_Q of T_2 with higher initiator priority, are sent to T_1 causing it detect a false deadlock.

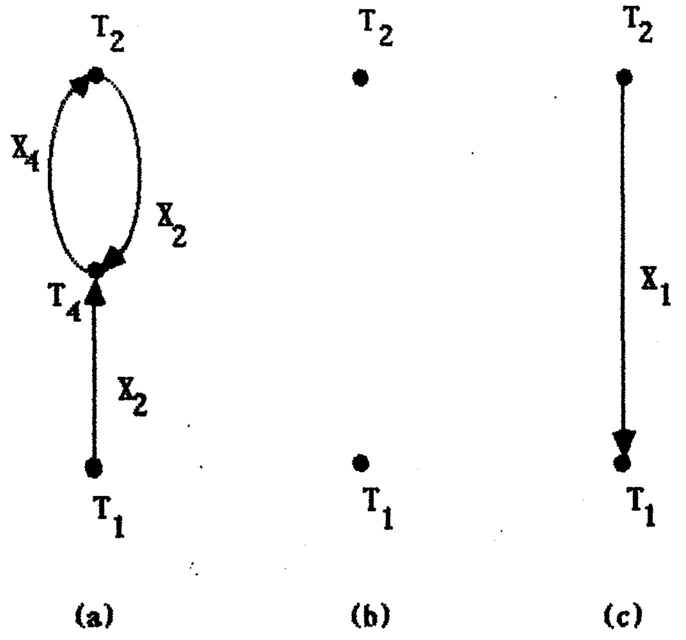


FIGURE 6.2 Example of a false deadlock

The algorithm should be modified such that once a transaction is chosen to abort, it should initiate a clean message which should not be discarded until it returns to the transaction to be aborted. And the information of the clean message should be used by each transaction on the cycle.

(B) *False Deadlocks Due to Old Information.* Now, consider the example in Figure 6.3(a). A deadlock exists between transactions T_2 and T_4. Transactions T_1, T_3, and T_5 are waiting transitively on T_4. The probe (T_1, T_5) is transferred to T_2 via other transactions. When T_4 aborts, the probe (T_1, T_5) remains in the probe_Q of T_2, although there is no wait-for relation between T_1 and T_2. Later when T_2 requests a resource held by T_1, the necessary probes are sent to T_1, including (T_1, T_5). In this situation as shown in Figure 6.3(c), T_1 detects a false deadlock.

To avoid this type of false deadlock, the probe_Q's of all the transactions involved in the deadlock cycle should cleansed of all the probes upon receipt of the clean message. Unfortunately, this cleansing can prevent the detection of some future deadlocks. To avoid this situation, all of the transactions involved in the deadlock cycle or waiting for data items held by the transactions involved in the deadlock cycle should retransmit and/or reinitiate the probes. Another solution to this problem is just ignoring such deadlocks, because they occur very seldom. In the algorithm of Choudhary *et al.* [2], the former method is used. It can be seen easily that in this case, the number of messages in the system increases drastically.

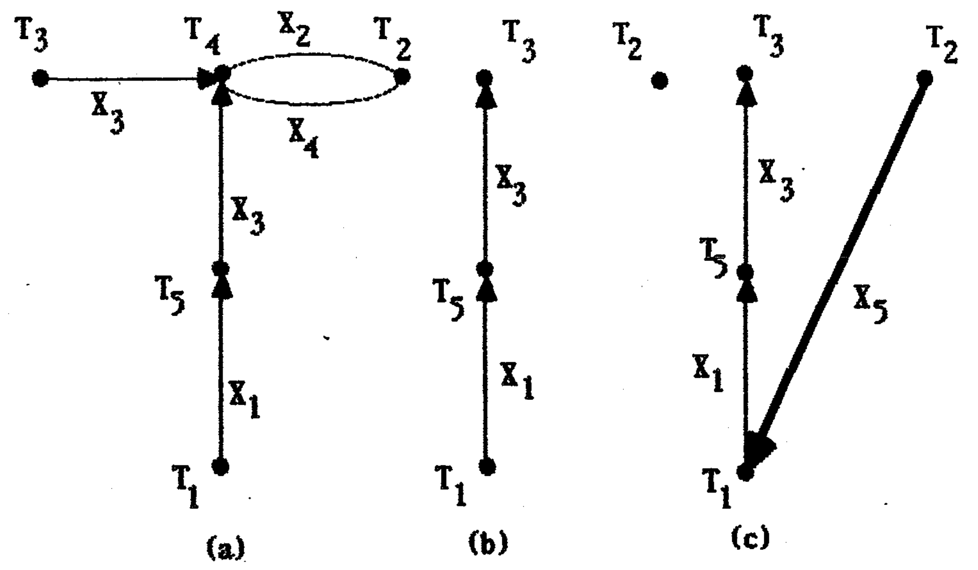


FIGURE 6.3 Another false deadlock example

6.3. Some More Modifications

Choudhary *et al.* [2] proposed a better version of the algorithm of Sinha and Natarajan [1], considering errors and the deficiencies in the algorithm. But there are some more modifications to make the algorithm correct and more efficient. Since some of these are structural modifications, we can call the new algorithm as a new edge-chasing deadlock detection algorithm.

(A) *The Problem of Unresolved Deadlocks.* Consider the situation shown in Figure 6.4(a). Transactions T_1 and T_3 have locked resources X_1 and X_3 respectively. Transaction T_2 has also locked resources X_2 and X_4. Transactions T_1 and T_3 request resources X_2 and X_4 respectively. Because T_1 has higher priority than T_2, DM(X_2) sends probe (1, 2) to T_2. In Figure 6.4(b), T_2 requests resource X_3. Because X_3 has been locked by T_3, and also there is an antagonistic conflict, DM(X_3) sends probe (2, 3) to T_3. T_2 transfers a copy of each probe stored in its probe_Q to the DM(X_3). This indicates that a copy of probe (1, 2) is sent to T_3. Because T_3's priority is lower than the junior of the probe, it assigns itself as the junior of the probe and sends both probes to DM(X_4). When DM(X_4) receives probe (2, 3), it declares deadlock and sends a message to the victim to make it create the corresponding clean message. It also transmits probe (1, 3) to T_2. While probe (1, 3) is moving in the cycle, clean (3, 2) reaches T_2 and it purges every probe from its probe_Q. However probe (1, 3) is ahead of the clean message; it passes T_3 before its abortion and reaches T_2. In Figure 6.4(c), after the abortion of T_3, T_2 requests X_1. Because it is held by T_1, the request of the transaction is put into the wait_Q of the corresponding data manager. Then, T_2 transmits a copy of its probe_Q to DM(X_1). On receiving probe (1,3), the data manager declares a deadlock with a false victim. Although there is a cycle, no information about the real victim is received. Because the identity of the real victim is not known, deadlock cannot be resolved.

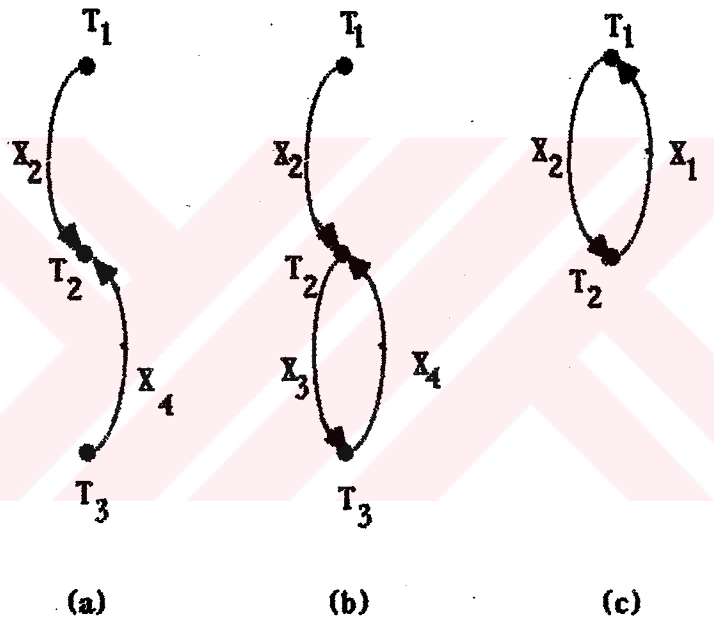


FIGURE 6.4 An unresolved deadlock example

Our modification is that all transactions store the probes received with the identity of the data manager sending it. We need to add a *sender* field into probes. The field contains the identity of the node (data manager or transaction) sending the probe. When a clean message is received, not all probes but the only the ones transmitted by the data manager which is also on the cycle are purged from the probe_Q. The victim discards all the messages received, after initiating a clean message.

(B) *Picking the Highest Priority Transaction from Request_Q.* As it is explained each data manager has a request_Q to hold the information related to transactions which are currently waiting for the corresponding resource. In both the original and modified algorithms, priority of the requesting transactions has no importance on deciding the new_holder of the data item when the data item is released by the current holder. A data manager can assign the lock to any transaction waiting in the request_Q. Then, for each requester in the probe_Q for which requester>new_holder, the data manager initiates a probe and sends it to the new_holder.

Since it is a priority based system, it will be more logical to pick the transaction with the highest priority from the request_Q as the new_holder. Otherwise, there is always a chance for the new_holder to be aborted by a higher priority transaction which is waiting in the request_Q. This modification will reduce the number of probes sent and the number of deadlocks in the system. The only drawback of this modification is the extra computation to find the transaction with the highest priority in the probe_Q. On the other hand, when assigning a resource, creation of probe is not required, because all other transactions waiting for the resource have lower priority.

(C) *A Probe_Q for Data Managers.* In the modified version of the algorithm, after allocating a resource to one of the transactions waiting in the request_Q (if any), data manager sends messages to the remaining transactions in the request_Q to transfer their probe_Q's. When data manager receives those probes, it transfers them to the new_holder, after checking priorities of the initiators. As it can be imagined, the number of messages transmitted increases in such situations.

If each data manager keeps the probes that it receives, there will be no such overhead. When the above condition occurs, instead of sending messages to the transactions and waiting for them to transfer their probe_Q's, data manager picks the necessary probes from its probe_Q and sends a copy of them to the new_holder. A Probe is deleted from the probe_Q of a data manager when its sender, which is also requester, becomes the holder of the data item or when the sender is aborted, or when deadlock happens. In case of deadlock, the probes which are sent by the transaction, and waiting for the corresponding data item and also on the cycle, are purged from the queue. When the holder changes, the probes which are sent by new_holder are deleted from the queue.

This modification also reduces the number of messages and probes sent. But, there is always a tradeoff; In this case, it increases the amount of space consumed. Additional space is required for the probe_Q of each data manager.

It can be easily seen that the aim of the last two modifications is to reduce the number of messages and probes sent. Since this is an algorithm for distributed systems, the amount of messages transmitted is very important. It has a negative effect on the completion time of transactions and it creates heavy traffic in the system. For the first modification, we can say correction, instead of modification, of the algorithm.

6.3.1 Modified Algorithm

In this subsection, we will introduce a new version of the algorithm that contains the improvements explained in the previous section. The structure of probes is changed. A new field *sender* is added to the fields of the probe–*probe(initiator,junior,sender)*. The sender contains the identity of the unit (data manager or transaction) sending the probe. In the same way, the structure of the clean message is also changed–*clean(victim,initiator,sender)*. The function of the sender is the same as above. Another structural modification is that data managers also have probe_Q's. The probes received by a data manager are kept in their probe_Q's.

(A) ***Modified Deadlock Detection Algorithm***: This part of the algorithm is explained in three steps:

(1) A data manager initiates a probe if there is an antagonistic conflict. It means that the requester of a data item has higher priority than its holder. In such a case, probe(initiator,victim,sender) is created and sent to the holder.

When the holder releases a data item, and if there are some transactions waiting for it, the data item is granted to the one with the highest priority. Because all other waiting transactions have less priority than the new holder, no probe is initiated under such a situation–unlike the previous two algorithms.

When a transaction completes or aborts, it releases its locks. As explained above, the data item is granted to another transaction. The probes which are sent by the new holder are purged from the probe_Q of the data manager–if any. Then the probe_Q of the data manager is checked, a copy of the probes of which initiator is greater than the new holder is sent to the new holder. The sender field of the probe is changed before sending it.

(2) A transaction saves the probes received in its probe_Q's before it enters the second phase of the two-phase locking. After it enters the second phase, all the probes received are discarded.

When a transaction T receives probe(initiator,junior,sender), it performs the following:

```
if (junior > T)
then
  junior := T;
save the probe in the probe_Q;
if T is in wait state
then
  transmit a copy of the saved probe to the data manager where it is waiting,
  after changing the sender field
```

When a transaction is waiting to acquire a data item after changing the sender part, it transmits a copy of each probe received to the data manager where it is waiting.

(3) When a data manager receives a probe(initiator,victim,sender), it performs the following:

```
save the probe in the probe_Q;
if (holder < initiator)
then
  send a copy of the probe to the holder, after changing the sender field
else
  declare deadlock and initiate deadlock resolution;
```

When the holder of a data item changes, the data manager purges all the probes which are sent by the new holder from its probe_Q. Then the probe_Q of the data manager is checked, a copy of the probes of which initiator is greater than the new holder is sent to the new holder. The sender field of the probes is changed before sending them.

(B) *The Deadlock Resolution Algorithm*: This part is also explained in three steps.

(1) When a deadlock is declared, the detecting data manager chooses the junior of the probe as the victim and sends an abort signal to it. The aim of the abort signal is to give necessary information to the victim transaction. This information contains the identity of the initiator. On receiving an abort signal the victim initiates a clean(victim,initiator,sender) message and sends it to the data manager where it is waiting. After initiating an abort message, the victim discards any probe or clean message it receives.

The victim aborts when its abort message returns to itself.

(2) When a data manager receives a clean message, it purges every probe sent by the sender of the clean message from its probe_Q. It propagates the clean message to its holder after changing the sender field.

It reinitiates probes for each requester with a higher priority than the holder. A copy of the remaining probes with an initiator having a higher priority than the holder is sent to the holder.

(3) When a transaction T receives a clean(junior,initiator,sender), it performs the following:

```
if T is in wait state
then
 if T = junior
 then
  enter the abort phase, release all locks and purge every probe from its
  probe_Q
 else
  purge every probe, of which sender is equal to the sender of
  the clean message, from its probe_Q;
  after changing the sender of the clean message, propagate it
  to the data manager where T is waiting
else
 discard the clean message;
```

Modifications in the algorithm are shown using bold characters.

VII. A SIMULATION USING THE NEW PRIORITY BASED PROBE ALGORITHM FOR DEADLOCK DETECTION AND EXTENSION FOR DISTRIBUTED SYSTEMS

The aim of this simulation study is to show the performance of the new algorithm which is explained in Part VI. Listing of the simulation program can be found in Appendix C. In this part, the simulation model of the algorithm is explained. The results are given. The algorithm is not compared with any other deadlock detection algorithm. The simulation results are given to show that it works under deadlock conditions. And as an extension to simulate the algorithm on a distributed system, a distributed system model is introduced.

7.1. Simulation Model of a Single-Site System

A single-system consists of processors, channels, memories, resources, etc. all of which are controlled by a central unit. Before going into the detail of the system model, it is better to give the assumptions we make on the system:

(a) Basic two phase locking is used to solve the problem of synchronizing access to a data item. All locks on a data item are considered to be exclusive locks, no shared access is allowed.

(b) Each transaction may make at most one outstanding request at a time, one resource model. So detecting a cycle in the system is necessary and enough condition for deadlock declaration.

(c) All transactions in the system are assumed to be designed properly, i.e., no transaction in the system contains infinite loops or similar errors.

(d) It is also assumed that there is no memory problem, such as, partitioning files into pages because they do not fit into the main memory. We can put any data into the memory when required.

(e) Each transaction is a batch process. Interactive processes are not used in the model.

For the modelling of a single-site system the approach of Agrawal *et al.* [14] is used. Their approach is simplified in some respects. A great deal of interest is paid for the modelling of concurrency control part.

There are three parts of a concurrency control model: a *database system model*, a *user model*, and a *transaction model*. The database system model captures the relevant characteristics of the system's hardware and software, including physical resources and their associated schedulers, the characteristics of the database, such as its size or granularity, the load control mechanism for controlling the number of active transactions in the system and the concurrency control algorithm. The user model deals with the arrival process for users, assuming either an open system or a closed system with terminals. The type of the processes, batch-style or interactive, is related to this part. The transaction model captures the behavior and processing requirements of the transactions in the workload.

Queuing model of the system is shown in Figure 7.1. There are a fixed number of terminals from which transactions originate. There is a limit to the number of transactions allowed to be active at any time in the system, the multiprogramming level *mpl*. When a new transaction originates, if the system has a full set of active transactions, it enters the *ready queue* where it waits for the currently active transactions to terminate successfully or abort. When there is enough space, transactions move from the ready queue to the *concurrency control queue* (cc queue), begin to execute and make their first requests. These requests are handled by the *concurrency control unit*. When a request is granted after an amount of time, the transaction is placed into the cc queue for new requests. The duration between each data item request is uniformly distributed between one and *max_req_time*. The delay to transfer a transaction from ready queue to cc queue is computed by using the uniform distribution between one and *max_move_time*.

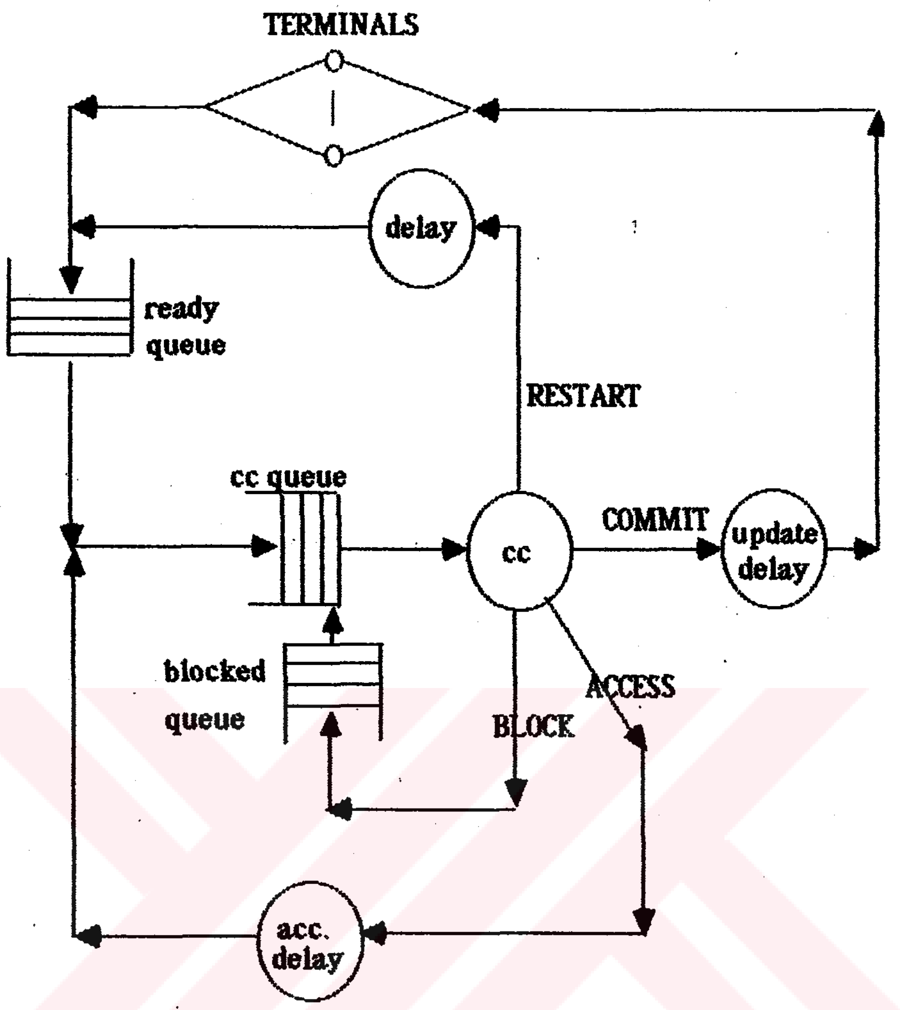


FIGURE 7.1 Logical queuing model for a single-site system

If the result of a request is that the transaction must be blocked, it enters the *blocked queue* until the requesting data item becomes available. If a request leads to a decision to abort a transaction, it goes back to the ready queue, possibly after a randomly determined restart delay period of mean *restart_delay*. During that period, the other transactions which cause the abortion of the transaction leave the system. It then makes all of the same requests again. When a transaction is restarted, its *start_time* does not change. Start_time is the value of the local clock when the transaction is initialized. Since the priorities of the transactions are measured with their start_time's, aborted transactions have higher priorities than the transactions initiated after their abortion. This method is called *gaining priority by ageing*. Restart_delay can be arranged according to *response_time*. Response_time is measured as the difference between when a terminal first submits a new transaction and when the transaction returns to the terminal following its successful completion, including any time spent waiting in the ready queue, time spent before (and during) being restarted, etc. It is better if we make the duration of restart_delay adaptive, depending on the observed average response time. Actually, the importance of restart_delay depends on the load of the system. If the system is heavily loaded, restart_delay loses its importance (a restarted process has to wait, anyway). In lightly loaded systems, restart_delay should be adjusted well. Short restart_delay causes deadlock to happen again. A long restart_delay is waste of time for restarted transactions.

When a transaction completes, all the data items requested by it are updated and the locks are released, and then a new transaction is transferred from the ready queue to cc queue. The size of the ready queue is limited with the number of terminals, because only one job can be sent from a terminal at a time. The size of cc queue is limited with mpl.

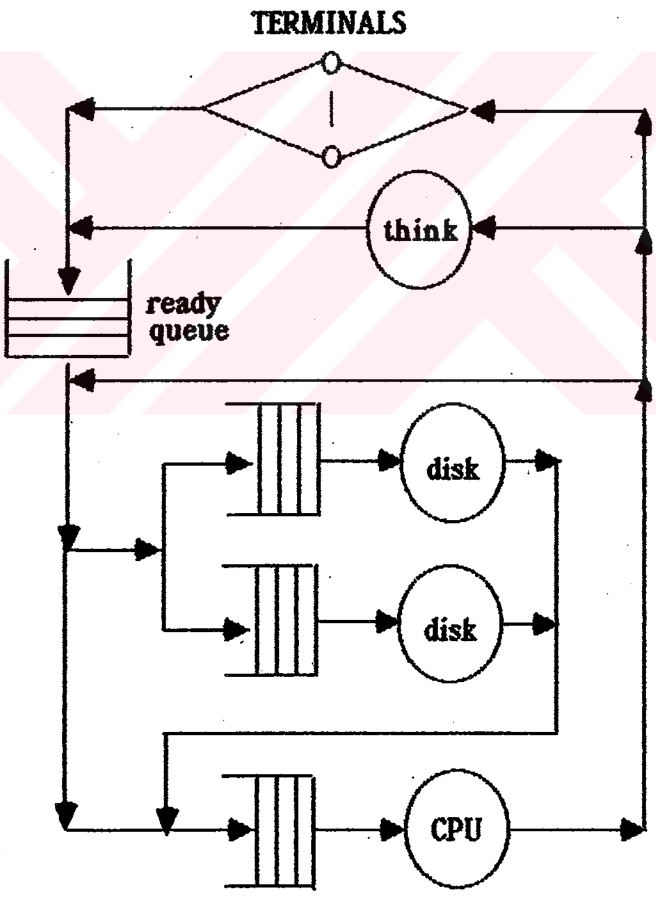


FIGURE 7.2 Physical queuing model for a single-site system

Parameter	Meaning
num_of_res	Number of database objects
num_of_term	Number of terminals
max_res	Maximum number of resources (database objects) requested
min_res	Minimum number of resources (database objects) requested
mpl	Multiprogramming level
context_switch_time	Time required for context switching
max_move_time	Transfer delay from ready queue to cc queue
think_time	Mean time between transactions created from a terminal
min_acc_dur	Minimum disk access time for a database object
max_acc_dur	Maximum disk access time for a database object
max_req_time	Maximum duration between each resource request

TABLE 7.1 Model parameters

CPU and I/O resources underlie the logical model of Figure 7.1. The amounts of I/O and CPU time per logical service are specified as model parameters. To make the model simpler, the number of CPU servers is restricted to one, and there are multiple I/O servers. The physical queuing model is shown in Figure 7.2, and Table 7.1 summarizes the associated model parameters. When a transaction requests CPU, it is put into the CPU queue. Requests in CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. The service discipline for the I/O requests is also FCFS. Another parameter which is needed to define at this point is *context_switch time* which is the amount of time required to save all the registers for the old transaction and to load the registers for the new transaction.

The *think_time* parameter is the mean time delay between the completion of a transaction and the initiation of a new transaction from a terminal. It is assumed that think_time is exponentially distributed.

A transaction is modeled according to the number of data items that it requests. The parameter *tran_size* is the average number of objects requested by a transaction. The amount of data item that a transaction requests is defined by the uniform distribution between *min_size* and *max_size* (inclusive). Because it is a uniform distribution, we do not need the mean value as an input. The data objects are randomly chosen (without replacement) among all of the data items in the database.

7.2. Application of The Algorithm to The Model

Before going into the details of the algorithm, it is better if we make a definition of a transaction for this system. In the system, a transaction can be defined by using the number of data items it requests, the period of time between the requests, and the initialization time. The number of data items is selected using a uniform distribution between the maximum and the minimum number of requested resources. The period of time between each request is also defined by another uniform distribution. In these distributions the upper and the lower bounds can be changed depending on the type of the transactions. In our simulation, all transactions are of the same type. Initialization time (start_time) is the value of the clock when the transaction is initialized. In the system, transactions can be in one of four modes: *active*, *waiting*, *restarted*, *restarted-and-waiting*. Initially all of the transactions are in the active mode. Active means that the transaction is either executing or waiting for the CPU. If a request of a transaction is not allocated, it enters waiting mode. A transaction goes from waiting mode to active mode when it gets the resource that it is waiting for. When an aborted transaction restarts, its mode becomes restarted and does not change until it acquires all the resources that it requested in previous activation. If a restarted transaction waits, its mode is changed to restarted-and-waiting. For both waiting and restarted-and-waiting transactions, we will use the term waiting throughout this part.

The probe sending mechanism works as explained in the previous part. There is a special process (*sp-process*) in the system which wakes up when a transaction is blocked. Because waiting transactions cannot access the CPU, they do not know if the other transactions have sent them any probe or not during their wait periods. The purpose of the sp-process is to check the probe_Q's of such transactions and (if any, and if necessary) to transmit the probes to the necessary data managers. When sp_process wakes up, it checks the probe queues of all the blocked transactions and the probe queues of the data managers for which blocked transactions wait. It, also, performs the probe transfer operations on behalf of blocked transactions. Cycles are detected by this process. Deadlock resolution is also performed by it.

In a multiprogramming environment, every working process is not actually active at any time (even though it seems so). An active transaction can capture the CPU when its turn comes, according to the scheduling algorithm. In our system, a transaction can not know about the probes it received when it does not hold the CPU, for example when waiting in the ready queue to hold the CPU. We can call this period *wait_in_ready_q period.* The probes which are received during wait_in_ready_q period, are kept in *wtg_pr* list. When a transaction captures the CPU, it first checks wtg_pr list and places the probes (if any) into *prlist,* which is the main probe queue of the transaction. Then it continues to its regular work. The transactions which are waiting for data items, also, receive probes during their waiting period. Such transactions cannot hold the CPU, before they acquire their requests. But by some means the probes sent during the wait period must be handled, otherwise deadlocks cannot be detected. sp_process does this job and checks the wtg_pr lists of waiting transactions. It transmits the probes to pr_list and to the data managers they are waiting for.

When a deadlock resolution is initiated, sp_process does not terminate before abortion of the victim. During that period no other process can capture the CPU.

7.3. Data Structures of The Model

There are some basic units which should be implemented using proper data structures in the model mentioned in the previous sections. The structure of those units are explained below.

(A) *Resource Table* : Resource table is an important component which keeps information related to the data items (resources) at a site. It has a linked list structure. For each resource there is a corresponding record in the list. The structure of each record is shown in Figure 7.3.

As it can be understood from the figure, *r_id* contains the identity of the resource, and *t_id* contains the identity of the transaction, currently holding the resource. *Waitq* is also a linked list containing the information about the transactions currently waiting for the resource.

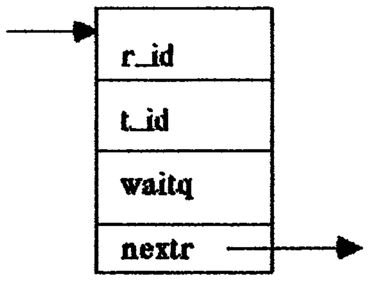


FIGURE 7.3 A record in resource table

(B) *Transaction Table* : Transaction table contains the information about the transactions which can be initialized by the terminals. It has a linked list structure. The structure of a node in the list is shown in Figure 7.4.

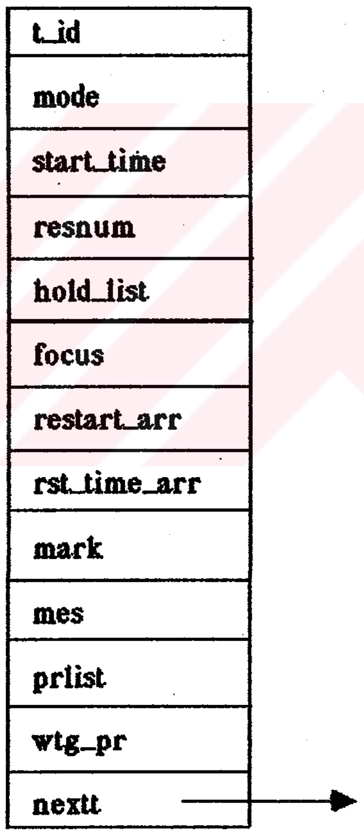


FIGURE 7.4 A record in transaction table

t_id contains the identity of the transaction. *mode* contains the current mode of the transaction. *start_time* is assigned when the transaction is initialized. *res_num* contains the number of data items the corresponding transaction will request. It is a random and predetermined value. *hold_list* is a set containing the identity of data items that the transaction is holding. *restart_arr*, *rst_time_arr*, and *mark* are the parameters used when the transaction restarts and contain the information about requested resources and request times. *prlist* and *wtg_pr* are lists containing the probes received by the transaction. If the probes are sent when the transaction is not active, they are kept in wtg_pr.

(C) *Probe Queues*: Each transaction and data manager has a probe queue in the system, which is actually a linked list. Probe queues are used to keep the probes sent by the transactions that are directly or transitively waiting for that unit. Transactions receive probes from data managers and data managers receive probes from transactions. Probes are kept with the identity of the sender to make the life easier during deadlock resolution activities. The structure is shown in Figure 7.5.

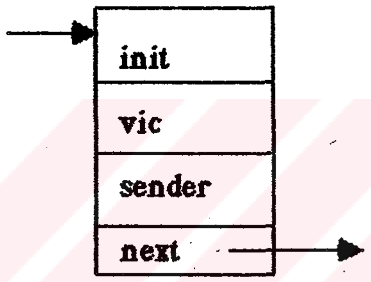


FIGURE 7.5 Structure of a probe queue

Init is the short form of initiator. It shows the initiator of the probe. *Vic* (stands for victim) contains the identity of the transaction that has the lowest priority among the visited transactions by the probe. In a transaction's probe queue *sender* contains the identity of the data manager that has sent the probe. In a data manager's probe queue, on the other hand, *sender* contains the identity of the transaction who has sent it.

(D) *Wait Queue*: Each data manager has a wait queue to store the information related to the transactions requesting the corresponding data item. It has a linked list structure. While assigning a data item to a transaction if there are more than one transaction, the data manager considers the priorities of the requesting transactions. So both the identities and the priorities of the waiting transactions are kept in. Figure 7.6 shows the structure of a wait queue.

Requests in the wait queue are ordered according to the priorities of the requesting transactions. *t_id* contains the identity of the requesting transactions. *Start_time* keeps the initialization time and shows the priority of the transactions.

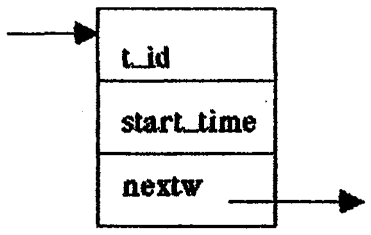


FIGURE 7.6 Structure of wait queue

The queues in the system are simple linked lists just keeping the information required.

7.4. Results Obtained

The main performance metric used in this thesis is *throughput.* Throughput is taken as the number of completed transactions per ten thousand units of time. Number of probes sent per a period of time can be considered as another performance metric. Response_time can be a good performance metric, for the systems in which it has importance. In our simulation throughput, number of deadlocks per ten thousand units of time, number of probes per ten thousand units of time, and response_time are used as performance metrics.

Context_switch_time is accepted as a unit of time in the system. When assigning time values to the parameters, the ratio of the assigned value to context_switch_time is taken into account. Some parameters have fixed values, such as num_of_res, num_of_term, max_res, min_res, context_switch_time, min_acc_dur, max_acc_dur and max_req_time. The value of mpl is changed. Simulation parameter settings are shown in Table 7.2.

To get the simulation results, the program is executed until a thousand transactions complete. With each different setting this execution is repeated.

Parameter	Value
num_of_res	200
num_of_term	50
max_res	8
min_res	2
mpl	2, 5, 7, 10, 15, 30, 50
context_switch_time	1
max_move_time	4
think_time	200
min_acc_dur	15
max_acc_dur	65
max_req_time	25

TABLE 7.2 Simulation parameter settings

The value of mpl is changed from time to time and differences in the results are observed. In these simulations adaptive restart_delay is used. The behavior of the system under different mpl values are shown in Table 7.3.

Think_time = 200;
Restart_delay = adaptive;

mpl	response-time	num.of probes /10000 units	num.of deadlocks /10000 units	throughput
2	7400	1.9	..1	64
5	4359	11.2	.5	107
7	4285	24.7	1.2	109
10	4476	46.3	1.8	104
15	5018	78.0	3.8	93
30	8037	264.0	8.1	53
50	11671	320.2	8.5	34

TABLE 7.3 Simulation results taken with different mpl values

Think_time is set to two hundred. As it can be seen from the table, the best throughput is taken when mpl is equal to seven. After that level because of heavy load in cc queue, throughput decreases gradually and the number of deadlocks begins to increase.

Some simulation results are taken to show the effect of the ordering of requests in the request_Q. We repeat the above simulation with the system for which the priority of requesting transactions has no importance in assigning the resource. The results can be seen in Table 7.4. The effect of the modification is not clearly seen until mpl reaches to fifteen. Because mpl is low, there are not many requests in request_Q's of data managers–so the ordering of requests has no importance on the performance of the system. After mpl reaches to fifteen–we can call it threshold level for this system–, the number of conflicts begin to increase so the length of request_Q's. Then the result of the modification can be seen clearly. If Table 7.3 is compared with Table 7.4, it is seen that the number of deadlocks in Table 7.4 is more than the number of deadlocks in Table 7.3, after the threshold level. Depending on the number of deadlocks, the number of probes sent, and response time in Table 7.4 are greater than the ones in Table 7.3. As a result of greater response time, the throughput in the Table 7.4 is less than the one in Table 7.3.

Think_time = 200;
Restart_delay = adaptive;

mpl	response-time	num.of probes /10000 units	num.of deadlocks /10000 units	throughput
2	7302	2.0	.2	65
5	4379	13.1	.6	106
7	4268	24.1	1.2	109
10	4405	47.1	2.3	104
15	5027	98.2	4.2	91
30	8139	303.2	8.9	50
50	12484	476.1	9.9	26

TABLE 7.4 Simulation results taken with different mpl values when priority has no importance in handling requests

To show the result of the modification which is explained in Section 6.3(C)– Probe_Q for data managers–, the system is simulated using the deadlock detection algorithm with no probe_Q's for data managers. In that case, after the detection of deadlock, data managers on the cycle should send the messages to the transactions waiting for them to reinitiate the probes (as in Choudhary *et al*.'s algorithm). And also after the termination or abortion of a process, the data managers whose holder has been the aborted or terminated transaction perform the same thing. The results are shown in Table 7.5. To see the performance of the modified algorithm, the results can be compared with the ones in Table 7.3. When data managers do not have probe_Q's, the message traffic becomes heavier because they should send a message to each transaction in their request_Q, and wait for the transactions to transmit a copy of their probe_Q's. This process takes time and creates extra delays in the system. Such delays are the cause of an increase in response time and in the number of deadlocks. Depending on these two conditions, throughput of the system decreases.

Think_time = 200;
Restart_delay = adaptive;

mpl	response-time	num.of probes /10000 units	num.of deadlocks /10000 units	throughput
2	7435	2.3	.1	64
5	4450	15.5	1.0	104
7	4342	30.9	1.7	105
10	4512	62.0	3.2	103
15	5141	119.4	5.1	88
30	8637	333.1	9.3	46
50	14740	532.3	10.2	21

TABLE 7.5 Simulation results taken when probe_Q's for data managers are not employed

The effect of think_time is shown in Table 7.6. mpl is set to seven and simulation results are taken for different think_time values. As think_time increases, response_time decreases. When think_time becomes very large, throughput starts to decrease, because CPU stays idle.

mpl = 7;
Restart_delay = adaptive;

think-time	response-time	throughput
200	4285	109
950	3683	107
1500	2975	107
3500	840	108
4000	790	102
4500	666	94
5000	548	90

TABLE 7.6 Simulation results taken with different think-time values

7.5. Extension : General Distributed System Model for The Further Studies

This study can be extended as a simulation of a distributed system using the priority based probe algorithm for deadlock detection. The deadlock detection algorithm remains the same, but some modification is required to adopt the single-site system model to a distributed system model. In this section, these modifications and a general distributed system model are introduced. A simple model of the system is shown in Figure 7.7.

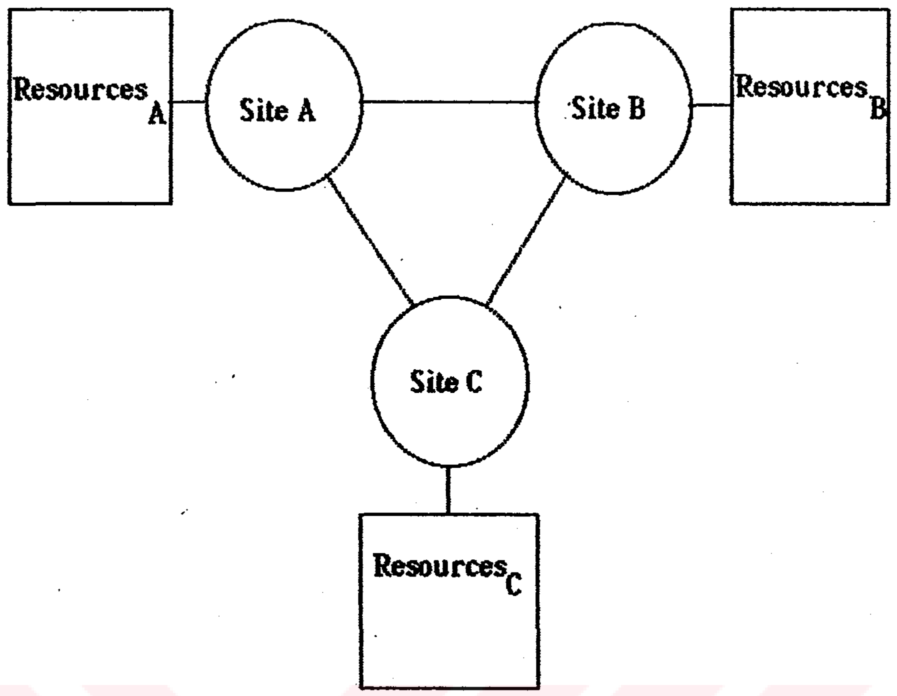


FIGURE 7.7 Simple model of the distributed system

The assumptions made are:

(a) Resources in the system are not replicated. They are single-copy resources. Each site is responsible from its own unique resources. To reach a remote resource, a message indicating the request is sent to the corresponding site. Then, allocation is performed by the site which contains the data item.

(b) Sites are connected using star topology. This implies that a message can be directly sent to any other site within the system. All messages sent arrive at their destination in finite time without any error (error-free system). The delay experienced by a message in a communication channel is constant for each channel.

(c) Site-to-site communication is pipelined, i.e. the receiving site gets the messages in the same order that the sending site has transmitted them.

(d) When a transaction requests a data item from another site, the corresponding data item is transmitted to the requesting site. After it is released by the requesting transaction, it is transmitted back to its original site. Process migration is not used, assuming that all the sites are heavily loaded.

(e) To construct a global completeness within the system, events are partially ordered using Lamport's approach [12].

(f) And all of the assumptions of the single-site system model are valid.

7.5.1. Simulation Model of a Site in Distributed System

In terms of data access, the main difference between a distributed system and a single-site system is that in the former, transactions may request data items which are residing at remote sites.

The logical queuing model for a distributed system site is shown in Figure 7.8. The previous model is modified according to inter-site requests. When a transaction requests a data item at another site, a timestamped message (for ordering of events among sites) containing the necessary information related to the transaction is sent to the corresponding site and the requesting transaction is put into *remote access block queue*. The transaction stays there until the requested data item is transmitted to its site. A site receiving such a message first arranges the global system clock according to the timestamp of the message (this point is explained in the following subsection in detail). Such messages cause the system to create an agent of the requesting transaction at the requested site. Such operations are called *pre-process* for a remote request by the requested site. Then, this transaction agent is put back of the ready queue in the requested site. When processing time comes to that transaction agent, its request is checked by the concurrency control unit. If the data item is available, it is transmitted to the requesting site. When the requesting site completes its job with this data item, it is sent back to the original site. During this period (transmission and retransmission), all the transactions requested for the corresponding data item are blocked. Transmission and retransmission periods for a file are fixed for each transmission channel. Transactions are not allowed to migrate to remote sites. They only create transaction agents at remote sites when they request data items at those sites.

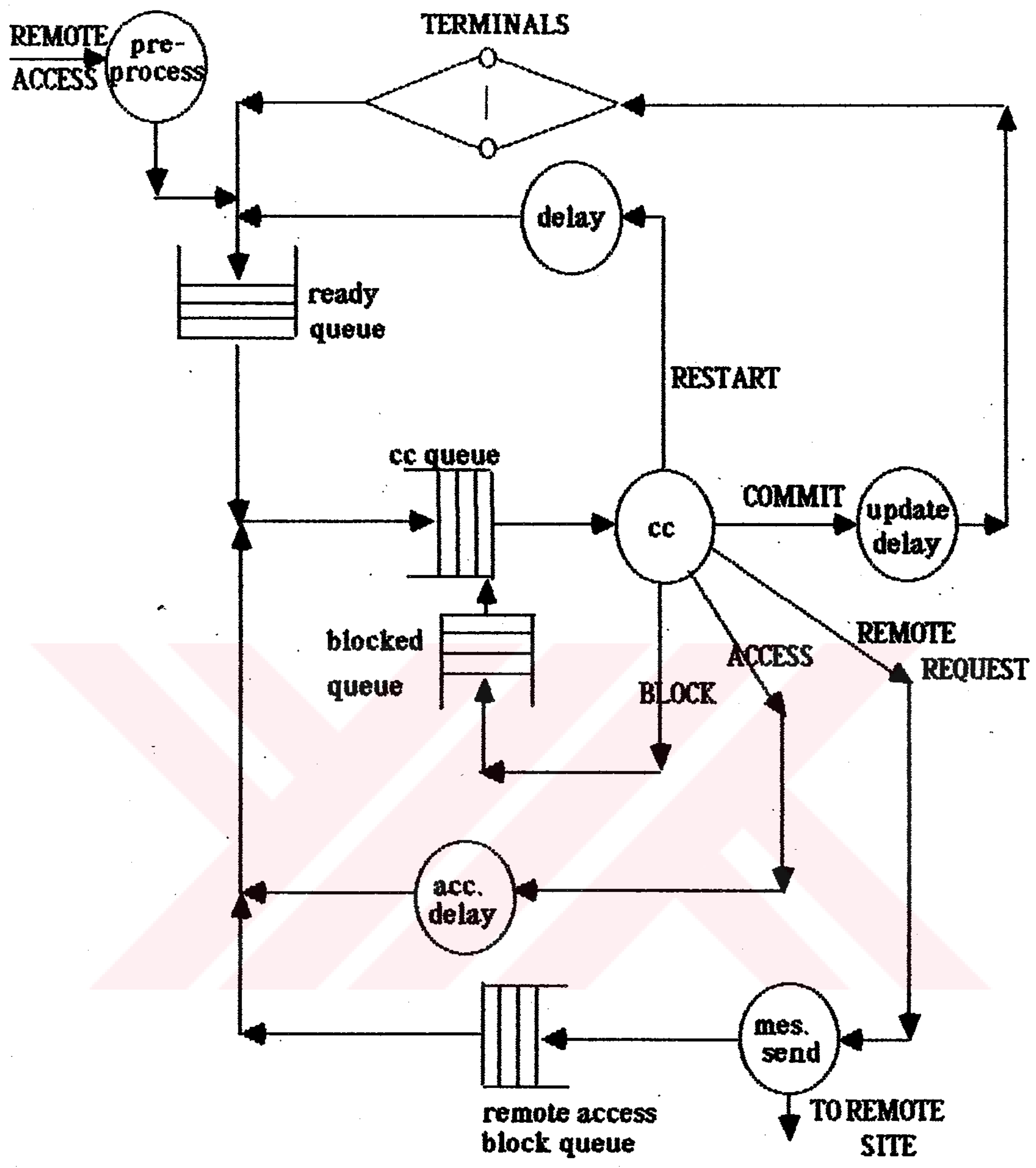


FIGURE 7.8 Logical queuing model for a distributed system site

Physical queuing model is also modified as shown in Figure 7.9.

Some new parameters are added, such as *communication_delay(i,j)*, *remote_request_probability(i,j)*. Communication_delay(i,j) contains the transmission delay experienced by a message while traveling from $site_i$ to $site_j$. Communication_delay(i,j) may be equal to communication_delay(j,i), if both transmission media are the same. For the simplicity of the model all communication_delay's can be equal. Remote_request_probability(i,j) keeps the probability of the request which is made by a transaction residing at $site_i$ for a resource residing at site$_j$. Remote_request_probability can be equal for all sites or change from site to site and also according to requested site.

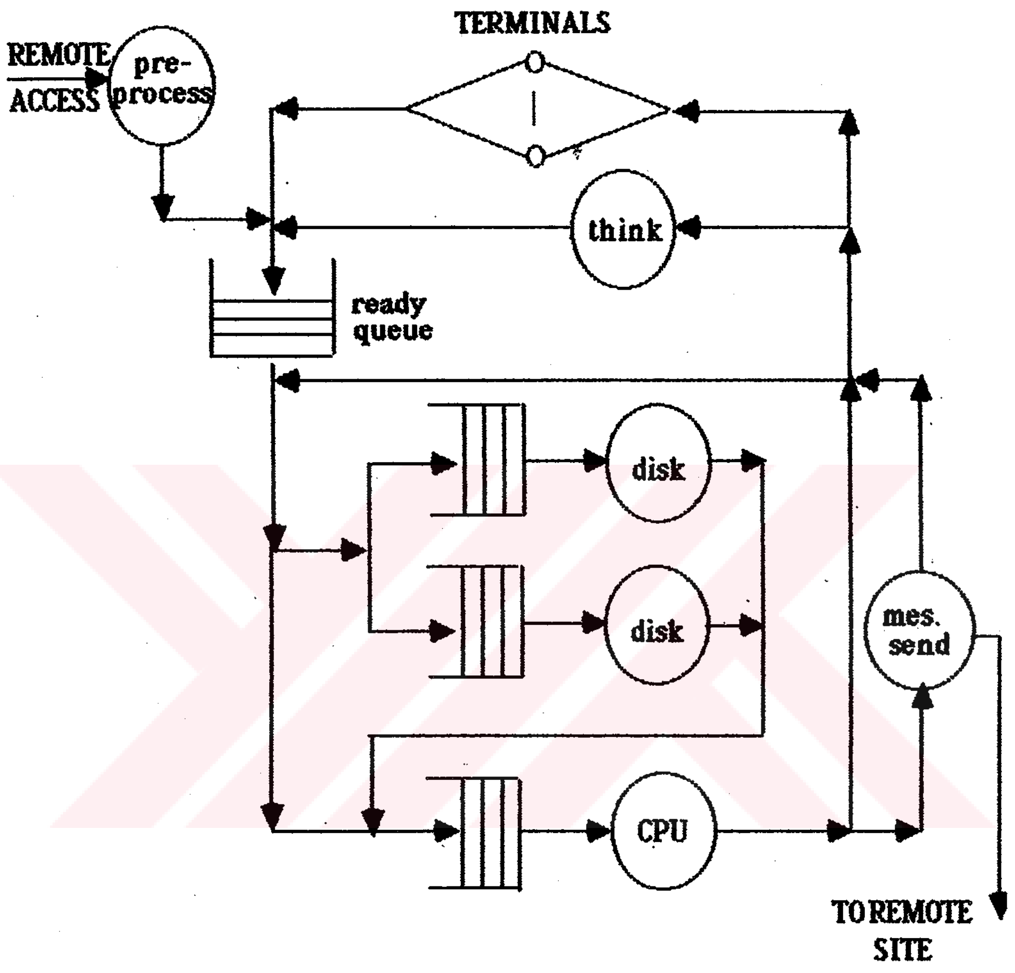


FIGURE 7.9 Physical queuing model for a distributed system site

7.5.2. Ordering of Events in the System

Since it is a distributed system, there is no common clock to order the events within the system. In a centralized system, events are totally ordered according to the system clock. In a distributed system, partial ordering of events according to the messages sent between sites is enough for the consistency of the system. For such ordering Lamport's "happened-before" relation [12] which is explained in Section 5.4 is used. The implementation of the relation to distributed systems is explained in the following paragraph.

Let's give an example of the situation that causes problems. Assume that there are two processes, at different sites, that communicate with each other (process A and process B). Process A sends a message to process B when its local clock is equal to 100. Process B receives this message when its local clock is equal to 50. Because they are at different sites, such a situation usually happens. But this is a contradictory situation. Although the message sending event happened before, it seems as if it has happened after receiving the message according to the local clocks of the sites. To solve this problem, we require the site to advance its local clock when it receives a message of which timestamp is greater than the value of its local clock. In the above example, the local clock of the receiving site becomes 101 when it receives the message. But with this clock, you cannot measure the duration of time between two events at a site correctly.

The relation is simulated in the following way: Each site has a global clock (logical clock) other than its own local clock. The purpose of the global clock is only partial ordering of the events among sites. Global clock is updated by both the ticks of the local clock and the messages sent from other sites. Because of this reason, the global clock is not used for measuring the time between any two events. When a transaction requests a remote data item, the request message is timestamped with the value of the global clock of the requesting site. It can be understood that the global clock of each site is different from each other, but partially ordered according to messages received from other sites.

VIII. CONCLUSION

In this thesis the deadlock problem in computing systems is introduced in detail. The policies used to deal with the deadlock problem are explained. Among these policies, "Deadlock Detection and Resolution" is selected and studied. Deadlocks are modeled according to the resource requirements of processes.

Distributed computing systems are introduced. Advantages and disadvantages of centralized, hierarchical, and distributed deadlock detection in distributed systems are discussed. It is seen that both centralized and hierarchical deadlock detection methods transfer WFG's between sites. The classes of distributed deadlock detection algorithms are presented. These classes employ path-pushing, edge-chasing, diffusing computations, and global state detection methods to detect deadlocks. Some algorithms from different classes are examined.

A priority based deadlock detection algorithm is introduced. The modified version of the algorithm is examined. A situation under which the algorithm cannot resolve a deadlock is found. To solve this problem and to make the algorithm better some structural changes are offered. We called the modified algorithm "the new priority based probe algorithm for deadlock detection."

Lacking of the formal proof, the algorithm is extensively tested through simulation for a single-site system model. To simulate the algorithm, the model which is used by Agrawal *et al.* [14] for the performance analysis of the different concurrecy control algorithms is employed, with a few modifications. Giving different values to the system parameters, the behavior of the system is observed.

To show the effect of modifications, the system is also simulated using the algorithms without modifications. First, the importance of ordering of requests in the request_Q is considered and the system is simulated using the algorithm which does not employ the ordering of events in request_Q. When the results are compared with the ones taken using the new algorithm, it is seen that for all performance metrics, the new algorithm performs better. Secondly, the same system is simulated using the algorithm which does not employ probe_Q's for data managers and the results are also compared with the results obtained using the new algorithm. Again better performance of the new algorithm is observed. The only disadvantage of using probe_Q's for data managers is that extra memory is required for the probe_Q's of data managers.

For further simulation studies, the simulation model is extended for distributed systems. Using this model, the new algorithm can be simulated for distributed systems and performance results are compared with other deadlock detection algorithms for distributed systems.

APPENDIX A. BASIC TWO PHASE LOCKING

To explain the subject Bernstein *et al.* [3] is referred. Throughout the section, we can replace the term transaction with the term process. Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. Each data item has a *lock* associated with it. Before a transaction T_1 may access a data, the scheduler first checks the associated lock. If another transaction T_2 holds the lock, then T_1 has to wait until T_2 releases the lock. The scheduler ensures that only one transaction can hold a lock at a time.

A basic two phase scheduler manages and uses its locks according to the following rules:

(a) When it receives an operation on a data from the transaction manager, the scheduler tests if the requested lock conflicts with the other lock that is already set. If so, it delays the operation, forcing the corresponding transaction to wait until it can set the lock it needs. If not, then scheduler sets the requested lock and sends the operation to the data manager.

(b) Once a scheduler sets a lock for a transaction, it may not release that lock at least until after data manager acknowledges that it has processed the lock's corresponding operation.

(c) Once a scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction.

Rule (a) prevents two transactions from concurrently accessing a data item in conflicting mode. Rule (b) supplements Rule (a) by ensuring that the data manager processes operations on data items in the order that scheduler submits them. Rule (c) called the *two phase rule*, is the source of two phase locking. Each transaction may be divided into two phases: growing phase during which it obtains locks, and shrinking phase during which it releases locks. Its function is to guarantee that all pairs of conflicting operations of two transactions are scheduled in the same order.

An important unfortunate property of two phase locking is that they are subject to *deadlocks*.

APPENDIX B. MODIFIED PROBE ALGORITHM BY CHOUDHARY *et al.*

In this part we represent the modified probe algorithm by Choudhary *et al.* [2]. The algorithm makes no assumption about the scheduling policy of a data manager. When two or more transactions are simultaneously waiting for a data item, the data manager may assign the lock for that data item to any transaction.

(A) *The Revised Basic Deadlock Detection Algorithm*

The basic deadlock detection algorithm now has the following steps.

1) A data manager initiates, propagates, or reinitiates a probe in the following situations.

a) When a data item is locked by a transaction, if a lock request arrives from another transaction, and *requester* > *holder*, the data manager initiates a probe and sends it to the *holder*.

b) When the current *holder* releases a data item, the data manager schedules a "waiting lock request." If there are more lock requests still in the request_Q, then for each lock request for which *requester* > *new holder*, the data manager initiates a probe and sends it to the *new holder*.

When a data manager initiates a probe it sets

initiator := *requester* ;
junior := *holder* ;

c) When a transaction completes or aborts, it releases its locks. The data manager associated with each released data item assigns the lock for the data item to some transaction (heretofore referred to as *new holder*) waiting for that data item (if one exists). Each data manager then requests all remaining transactions waiting on the new lock to transmit their complete probe_Q's to itself. (The identities of these transactions are obtained from the data manager's request_Q.) The data manager forwards each received probe (*initiator,junior*) to *new holder* the lock for which *initiator* › *new holder*.

2) Each transaction maintains a queue, called a probe_Q, where it stores all probes received by it. The probe_Q of a transaction contains information about the transactions which wait for it directly, or transitively. Since a transaction follows two-phase locking, the information contained in the probe_Q of a transaction remains valid until it aborts or commits.

After a transaction enters the second phase of the 2PL, it does not discard the probe_Q. However, during the second phase, any probe received is ignored.

Otherwise, a transaction sends a probe or a copy of its probe_Q to the data manager, where it is waiting in the following three cases.

a) When a transaction *T* receives probe (*initiator,junior*), it performs the following.

```
if (junior › T)
then junior := T;
save the probe in the probe_Q;
if T is in wait state
then transmit a copy of the saved probe to the data manager where it is
    waiting;
```

b) When a transaction issues a lock request to a data manager and waits for the lock to be granted (i.e., it goes from active to wait state), it transmits a copy of each probe stored in its probe_Q to that data manager.

c) If a transaction is waiting and receives a request for its probe_Q from the data manager where it is waiting, it sends a copy of its probe_Q to the data manager.

3) When a data manager receives probe (*initiator,junior*) from one of its requesters, it performs the following.

```
if holder › initiator
then discard the probe
else if holder ‹ initiator
  then propagate the probe to the holder
  else declare deadlock and initiate deadlock resolution;
```

When a deadlock is detected, the detecting data manager has the identities of two members of the cycle, *initiator* and *junior*, i.e., the highest and lowest priority transactions, respectively. The *junior* is chosen as the deadlock *victim*.

(B) *The Deadlock Detection and Post Resolution Computation*

This consists of the following three steps.

1) To abort the *victim*, the data manager that detects the deadlock sends an abort signal to the *victim*. The identity of the initiator is also sent along with the abort signal: abort(*victim,initiator*). Since the *victim* is aborted, it is necessary to discard those probes (from the probe_Q of various transactions) that have the *victim* as their *junior* or *initiator*. Hence, on receiving an abort_signal, the *victim* does the following.

a) It initiates a message, clean(*victim,initiator*), sends it to the data manager where it is waiting.

b) The *victim* enters an abort phase only when its clean message returns to itself. Once it enters the abort phase, the *victim* releases all the locks it held, withdraws its pending request, and aborts. During this phase, it discards any probe or clean message that it receives.

2) When a data manager receives a clean(*victim,initiator*) message, it does the following.

a) It propagates the clean message to its *holder*.

b) It reinitiates the probes for each *requester* for which *requester* > *holder*.

c) It requests each transaction in the request_Q to retransmit its probe_Q.

3) When a transaction *T* receives a clean(*junior,initiator*) message, it acts as follows.

```
purge every probe from its probe_Q;
if T is in wait state
  then if T=junior
    then enter the abort phase and release all locks
    else propagate the clean message to the data manager where T is waiting
  else discard the clean message.
```

APPENDIX C. PROGRAM LISTING

This part contains the listing of the simulation program. The program is written in Pascal. It contains three include files–var_init.pas, initialize.pas, and simu.pas. First the listings of the include files are given according to their places in the main file. To explain the program, comment lines are used.

Listing of Include File VAR_INIT.PAS :

```
procedure var_init(var data:data_pack_type);
{ initializes all the system parameters of a single-site system }

begin
    data^.numofres:=s1_numofres;
    data^.numofterm:=s1_numofterm;
    data^.maxres:=s1_maxres;
    data^.minres:=s1_minres;
    data^.mpl:=s1_mpl;
    data^.context_switch:=s1_context_switch;
    data^.av_move_time:=s1_max_move_time;
    data^.maxreqtime:=s1_maxreqtime;
    data^.think_time:=s1_think_time;
    data^.minaccdur:=s1_minaccdur;
    data^.maxaccdur:=s1_maxaccdur;
    data^.res_q_ptr:=nil;
    data^.tr_q_ptr:=nil;
    data^.req_q_ptr:=nil;
    data^.ready_q_ptr:=nil;
    data^.cmpl:=0;
    data^.sys_clock:=0;
    data^.clock:=0;
    data^.cumm_tr_dur:=0;
    data^.numofprobes:=0;
    data^.numoftrans:=0;
    data^.numofcomptrans:=0;
    data^.mean_tr_dur:=0;
    data^.numofdeadlock:=0;
end;
```

Listing of Include File INITIALIZE.PAS :

```
procedure initialize(var data:data_pack_type);
{ initializes all the queues used in the simulation of the system
 and creates the first transaction of each terminal }
var
  req_ptr,request:a_req_typet;
  rd_ptr:el_rd_q_t;
  prev:tr_nodet;
  pres:res_nodet;
  last:el_rd_q_t;
  pre:a_req_typet;
  res,i:integer;


procedure find_term(tid:integer; var t:tr_nodet);
{ finds the record of the terminal whose identity is sent as
  parameter }
var
   count:integer;

begin
  if (data^.tr_q_ptr<>nil)
  then
  begin
    t:=data^.tr_q_ptr;
    t:=t^.nextt;
    count:=1;
    while ((t^.t_id<>tid) and (t<>nil)
          and (count<=data^.numofterm)) do
    begin
      t:=t^.nextt;
      count:=count+1;
   end;
   if (count>data^.numofterm)
   then
   begin
      writeln('ERROR - in procedure find_term, initialize');
      halt;
   end;
  end
  else
  begin
    writeln('ERROR -  in procedure find_term, initialize');
    halt;
  end;
end; { find_term }

procedure insert_req(var req:a_req_typet);
{ inserts the received record into cc queue }
var
  pre1,pre2,ptr:a_req_typet;
```

```
begin
  if (data^.req_q_ptr<>nil)
  then
  begin
    req^.next:=nil;
    ptr:=data^.req_q_ptr;
    if (ptr^.next=nil)
    then
      ptr^.next:=req
    else
    begin
      pre2:=data^.req_q_ptr;
      pre1:=pre2^.next;
      while ((pre1<>nil) and (pre1^.req_time<req^.req_time)) do
      begin
        pre2:=pre1;
        pre1:=pre1^.next;
      end;
      pre2^.next:=req;
      req^.next:=pre1;
    end;
  end
  else
    writeln('ERROR - in procedure insert_req, initialize');
end; { insert_req }

procedure transfer_bw_qs;
{ moves transactions from ready queue to cc queue }
var
  fr_rd,rptr:el_rd_q_t;
  fr_req,newone,ptr:a_req_typet;
  tr:tr_nodet;

begin
  if (data^.ready_q_ptr<>nil)
  then
  begin
    fr_rd:=data^.ready_q_ptr;
    fr_rd:=fr_rd^.nextel;
    fr_req:=data^.req_q_ptr;
    fr_req:=fr_req^.next;
    while ((fr_rd^.req_time<fr_req^.req_time) or (fr_req=nil))
           and (data^.cmpl<data^.mpl) and (fr_rd<>nil) do
    begin
      data^.cmpl:=data^.cmpl+1;
      new(newone);
      ptr:=data^.req_q_ptr;
      newone^.next:=ptr^.next;
      ptr^.next:=newone;
      newone^.t_id:=fr_rd^.t_id;
      find_term(newone^.t_id,tr);
      if (tr^.mode=rst)
      then
      begin
        newone^.req_time:=tr^.start_time;
        newone^.mode:=rstcpureq;
      end
      else
      begin
        newone^.req_time:=fr_rd^.req_time;
        newone^.mode:=new;
      end;
      rptr:=data^.ready_q_ptr;
      rptr^.nextel:=fr_rd^.nextel;
      fr_rd^.nextel:=nil;
      dispose(fr_rd);
      fr_rd:=data^.ready_q_ptr;
      fr_rd:=fr_rd^.nextel;
      fr_req:=data^.req_q_ptr;
      fr_req:=fr_req^.next;
      data^.clock:=data^.clock+data^.av_move_time;
    end;
  end
  else
    writeln('ERROR - in transfer_bw_qs, initialize');
end; { transfer_bw_qs }

procedure insert_tr_rd_q(req:el_rd_q_t);
{ inserts a transaction into ready queue }
var
  pre1,pre2:el_rd_q_t;

begin
  if (data^.ready_q_ptr<>nil)
  then
  begin
    pre1:=data^.ready_q_ptr;
    pre1:=pre1^.nextel;
    pre2:=data^.ready_q_ptr;
    while ((pre1<>nil) and (pre1^.req_time<req^.req_time)) do
    begin
      pre2:=pre1;
      pre1:=pre1^.nextel;
    end;
    pre2^.nextel:=req;
    req^.nextel:=pre1;
  end
  else
  begin
    writeln('ERROR - in procedure insert_tr_rd_q, initialize');
    halt;
  end;
end; { insert_tr_rd_q }

procedure job_submission(j:integer;delay:real);
{ initiates a transaction from the corresponding terminal }
var
  rdtrans:el_rd_q_t;

begin
  new(rdtrans);
  rdtrans^.nextel:=nil;
  rdtrans^.t_id:=j;
  rdtrans^.req_time:=data^.clock-delay*ln(1-random);
  insert_tr_rd_q(rdtrans);
end; { job_submission }
procedure init_trans_q;
{ initializes all the terminals in the system }
var
  ptrans,ptr,trans:tr_nodet;
  i,j:integer;

begin
  new(ptr);
  ptr^.nextt:=nil;
  data^.tr_q_ptr:=ptr;
  ptrans:=data^.tr_q_ptr;
  for i:=1 to data^.numofterm do
  begin
    new(trans);
    trans^.t_id:=i;
    trans^.mode:=act;
    trans^.start_time:=0;
    trans^.resnum:=0;
    trans^.hold_list:=[];
    trans^.focus:=0;
    trans^.prlist:=nil;
    trans^.wtg_pr:=nil;
    trans^.nextt:=nil;
    for j:=1 to data^.maxres do
    begin
      trans^.restart_arr[j]:=0;
      trans^.rst_time_arr[j]:=0;
    end;
    trans^.mark:=0;
    ptrans^.nextt:=trans;
    ptrans:=trans;
  end;
end; { init_trans_q }

procedure init_res_q;
{ initializes all the resources in the system }
var
  pres,ptr,resource:res_nodet;
  i:integer;

begin
  new(ptr);
  ptr^.nextr:=nil;
  data^.res_q_ptr:=ptr;
  pres:=data^.res_q_ptr;
  for i:=1 to data^.numofres do
  begin
    new(resource);
    resource^.r_id:=i;
    resource^.t_id:=0;
    resource^.waitq:=nil;
    resource^.probes:=nil;
    resource^.nextr:=nil;
    pres^.nextr:=resource;
    pres:=resource;
  end;
end; { init_res_q }


begin { initialize }
  init_trans_q;
  init_res_q;
  new(rd_ptr);
  rd_ptr^.nextel:=nil;
  data^.ready_q_ptr:=rd_ptr;
  rd_ptr^.t_id:=0;
  for i:=1 to  data^.numofterm  do
    job_submission(i,data^.think_time);
  data^.cmpl:=1;
  new(request);
  rd_ptr:=data^.ready_q_ptr;
  last:=rd_ptr^.nextel;
  rd_ptr^.nextel:=last^.nextel;
  request^.t_id:=last^.t_id;
  request^.req_time:=last^.req_time;
  request^.mode:=new;
  request^.next:=nil;
  last^.nextel:=nil;
  dispose(last);
  new(req_ptr);
  req_ptr^.next:=request;
  data^.req_q_ptr:=req_ptr;
  writeln(' all queues are INITIALIZED');
end; { initialize }
```

Listing of Include File SIMU.PAS :

```
procedure simulate(var data:data_pack_type);
{ simulates a sing-site system }
var
  resource           :res_nodet;
  trans              :tr_nodet;
  request            :a_req_typet;
  call_check_wtg_trs:boolean;


procedure check_clock(var req:a_req_typet);
{ updates the system clock }

begin
  if (req^.mode <> wtgtrck)
  then
    if (data^.clock<req^.req_time)
    then
      data^.clock:=req^.req_time;
end; { check_clock }
```

```
procedure find_term(tid:integer; var t:tr_nodet);
{ finds the terminal whose identity is given }
var
  count:integer;

begin
  if (data^.tr_q_ptr<>nil)
  then
  begin
    t:=data^.tr_q_ptr;
    t:=t^.nextt;
    count:=1;
    while ((t^.t_id<>tid) and (t<>nil)
           and (count<=data^.numofterm)) do
    begin
      t:=t^.nextt;
      count:=count+1;
    end;
    if (count>data^.numofterm)
    then
    begin
      writeln('ERROR - in procedure find_term');
      halt;
    end;
  end
  else
  begin
    writeln('ERROR - in procedure find_term');
    halt;
  end;
end; { find_term }

procedure find_res(rid:integer; var r:res_nodet);
{ finds the resource whose identity is given }
var
  count:integer;

begin
  if (r<>nil)
  then
  begin
    count:=1;
    r:=data^.res_q_ptr;
    r:=r^.nextr;
    while ((r^.r_id<>rid) and (r<>nil)
           and (count<=data^.numofres)) do
    begin
      r:=r^.nextr;
      count:=count+1;
    end;
    if (count>data^.numofres)
    then
    begin
      writeln('ERROR - in procedure find_res');
      halt;
    end;
  end
  else
  begin
    writeln('ERROR - in procedure find_res');
    halt;
  end;
end; { find_res }

procedure insert_tr_rd_q(req:el_rd_q_t);
{ inserts a transaction into ready queue }
var
  pre1,pre2:el_rd_q_t;

begin
  if (data^.ready_q_ptr<>nil)
  then
  begin
    pre1:=data^.ready_q_ptr;
    pre1:=pre1^.nextel;
    pre2:=data^.ready_q_ptr;
    while ((pre1<>nil) and (pre1^.req_time<req^.req_time)) do
    begin
      pre2:=pre1;
      pre1:=pre1^.nextel;
    end;
    pre2^.nextel:=req;
    req^.nextel:=pre1;
  end
  else
  begin
    writeln('ERROR - in procedure insert_tr_rd_q');
    halt;
  end;
end; { insert_tr_rd_q }

procedure insert_req(var req:a_req_typet);
{ inserts a transaction into cc queue }
var
  pre1,pre2,ptr:a_req_typet;

begin
  if (data^.req_q_ptr<>nil)
  then
  begin
    req^.next:=nil;
    ptr:=data^.req_q_ptr;
    if (ptr^.next=nil)
    then
      ptr^.next:=req
    else
    begin
      pre2:=ptr;
      pre1:=pre2^.next;
      while ((pre1<>nil) and (pre1^.req_time<req^.req_time)) do
```

```
      begin
        pre2:=pre1;
        pre1:=pre1^.next;
      pre2^.next:=req;
      req^.next:=pre1;
    end;
  end
  else
    begin
      writeln('req_q_ptr is nil - insert_req');
      halt;
    end;
end; { insert_req }

procedure transfer_bw_qs;
{ transfers transactions from ready queue to cc queue }
var
  fr_rd,rptr:el_rd_q_t;  {from ready_queue}
  fr_req,newone,ptr:a_req_typet;
  tr:tr_nodet;

begin
  if (data^.ready_q_ptr<>nil)
  then
  begin
    fr_rd:=data^.ready_q_ptr;
    fr_rd:=fr_rd^.nextel;
    fr_req:=data^.req_q_ptr;
    fr_req:=fr_req^.next;
    while (((fr_rd^.req_time<fr_req^.req_time) or (fr_req=nil))
           and (data^.cmpl<data^.mpl) and (fr_rd<>nil)) do
    begin
      data^.cmpl:=data^.cmpl+1;
      new(newone);
      ptr:=data^.req_q_ptr;
      newone^.next:=ptr^.next;
      ptr^.next:=newone;
      newone^.t_id:=fr_rd^.t_id;
      find_term(newone^.t_id,tr);
      if (tr^.mode=rst)
      then
      begin
        newone^.req_time:=tr^.start_time;
        newone^.mode:=rstcpureq;
      end
      else
      begin
        newone^.req_time:=fr_rd^.req_time;
        newone^.mode:=new;
      end;
      rptr:=data^.ready_q_ptr;
      rptr^.nextel:=fr_rd^.nextel;
      fr_rd^.nextel:=nil;
      dispose(fr_rd);
      fr_rd:=data^.ready_q_ptr;
      fr_rd:=fr_rd^.nextel;
      fr_req:=data^.req_q_ptr;
      fr_req:=fr_req^.next;
      data^.clock:=data^.clock+data^.av_move_time;
    end;
  end
  else
  begin
    writeln('ready_q_ptr is nil - transfer_bw_qs');
    halt;
  end;
end; { transfer_bw_qs }

procedure job_submission(j:integer;delay:real);
{ initiates a transaction from the given terminal }
var
  rdtrans:el_rd_q_t;

begin
  new(rdtrans);
  rdtrans^.nextel:=nil;
  rdtrans^.t_id:=j;
  rdtrans^.req_time:=data^.clock-delay*ln(1-random);
  insert_tr_rd_q(rdtrans);
end; { job_submission }

procedure initiate_probe(i,v,sndr:integer);
{ initiates the probe whose initiator, junior, and sender are
  given as parameters }
var
  pr:probet;
  tr:tr_nodet;

begin
  find_term(v,tr);
  new(pr);
  pr^.init:=i;
  pr^.vic:=v;
  pr^.sender:=sndr;
  pr^.next:=tr^.wtg_pr;
  tr^.wtg_pr:=pr;
  data^.numofprobes:=data^.numofprobes+1;
  pr:=tr^.wtg_pr;
  while (pr<>nil) do
  begin
    pr:=pr^.next;
  end;
end; { initiate_probe }

procedure dm_to_tran(var pr:probet; var tr:tr_nodet);
{ sends the given probe to the specified transaction }
var
 itra,vtra:tr_nodet;

begin
  find_term(pr^.init,itra);
  find_term(pr^.vic,vtra);
  if ((tr^.start_time > itra^.start_time)
      or (tr^.t_id = pr^.init))
  then
  begin
    if (tr^.start_time > vtra^.start_time)
    then
      pr^.vic:=tr^.t_id;
    pr^.next:=tr^.wtg_pr;
    tr^.wtg_pr:=pr;
    data^.numofprobes:=data^.numofprobes+1;
  end
  else
    dispose(pr);
end; { dm_to_tran }

procedure sendprobe(pr:probet; r:res_nodet);
{ sends the given probe to the specified data manager }
var
  tran:tr_nodet;
  tpr:probet;

begin
  new(tpr);
  tpr^:=pr^;
  tpr^.sender:=r^.r_id;
  tpr^.next:=nil;
  pr^.next:=r^.probes;
  r^.probes:=pr;
  data^.numofprobes:=data^.numofprobes+1;
  find_term(r^.t_id,tran);
  dm_to_tran(tpr,tran);
end; { sendprobe }

procedure insert_into_prlist(pro:probet; var tra:tr_nodet);
{ inserts the probe sent into probe_Q of the specified
  transaction }
var
  prev1,prev2:probet;

begin
  prev1:=tra^.prlist;
  prev2:=tra^.prlist;
  while ((prev1<>nil) and ((prev1^.init<>pro^.init)
          or (prev1^.vic<>pro^.vic))) do
  begin
    prev2:=prev1;
    prev1:=prev1^.next;
  end;
  if (prev1=nil)
  then
  begin
    pro^.next:=tra^.prlist;
    tra^.prlist:=pro;
  end
  else
    if  ((prev1^.init=pro^.init) and (prev1^.vic=pro^.vic))
    then
    begin
      dispose(pro);
    end;
end; { insert_into_prlist }
```

```
procedure transfer_prlist(var tra:tr_nodet; var r:res_nodet);
{ transfers the probes of the transaction to the specified
  data manager }
var
  loc:probet;
  rpr:probet;

begin
  loc:=tra^.prlist;
  while (loc<>nil) do
  begin
    new(rpr);
    rpr^.init:=loc^.init;
    rpr^.vic:=loc^.vic;
    rpr^.sender:=tra^.t_id;
    rpr^.next:=nil;
    sendprobe(rpr,r);
    loc:=loc^.next;
  end;
end; { transfer_prlist }

procedure dispose_list(var l:probet);
{ disposes the given probe_Q }
var
  ptr1,ptr2:probet;

begin
  ptr1:=l;
  l:=nil;
  while (ptr1 <> nil) do
  begin
    ptr2:=ptr1;
    ptr1:=ptr1^.next;
    ptr2^.next:=nil;
    dispose(ptr2);
  end;
end; { dispose_list }

procedure remove_holders_probes(var r:res_nodet; holder:integer);
{ removes the probes which are sent by the holder of the data
  item from probe_Q }
var
  res_pr,prev_res_pr:probet;

begin
  res_pr:=r^.probes;
  prev_res_pr:=res_pr;
  while (res_pr <> nil) do
    if (res_pr^.sender = holder)
    then
      if (res_pr = r^.probes)
      then
      begin
        r^.probes:=res_pr^.next;
        prev_res_pr:=r^.probes;
        res_pr^.next:=nil;
        dispose(res_pr);
        res_pr:=prev_res_pr;
      end
      else
      begin
        prev_res_pr^.next:=res_pr^.next;
        res_pr^.next:=nil;
        dispose(res_pr);
        res_pr:=prev_res_pr;
      end
      else
      begin
        prev_res_pr:=res_pr;
        res_pr:=res_pr^.next;
      end;
end; { remove_holders_probes }

procedure transfer_from_dm(res:res_nodet; var tra:tr_nodet);
{ transfers probes from a data manager to its holder }
var
  probe,pr:probet;
  itra,vtra:tr_nodet;

begin
  probe:=res^.probes;
  while (probe <> nil) do
  begin
    new(pr);
    pr^:=probe^;
    pr^.next:=nil;
    find_term(pr^.init,itra);
    find_term(pr^.vic,vtra);
    if ((tra^.start_time > itra^.start_time)
        or (tra^.t_id = pr^.init))
    then
    begin
      if (tra^.start_time > vtra^.start_time)
      then
        pr^.vic:=tra^.t_id;
      pr^.sender:=res^.r_id;
      pr^.next:=tra^.wtg_pr;
      tra^.wtg_pr:=pr;
      data^.numofprobes:=data^.numofprobes+1;
    end
    else
      dispose(pr);
    probe:=probe^.next;
  end;
end; { transfer_from_dm }

procedure release_all(var tr:tr_nodet);
{ releases all the resources which are held by the specified
  transaction }
var
  res:res_nodet;
  k:integer;
```

```
 procedure release_it(var r:res_nodet);
 var
   temp,wt:w_req_typet;
   act_tr,tra:tr_nodet;
   req:a_req_typet;
   prev_res_pr,res_pr:probet;
   i:integer;

 begin
   if (r^.waitq=nil)
   then
   begin
     r^.t_id:=0;
   end
   else
   begin
     temp:=r^.waitq;
     r^.waitq:=temp^.nextw;
     temp^.nextw:=nil;
     r^.t_id:=temp^.t_id;
     new(req);
     req^.t_id:=temp^.t_id;
     req^.req_time:=data^.clock
       +random(data^.maxaccdur-data^.minaccdur+1)+data^.minaccdur;
     req^.mode:=cpureq;
     req^.next:=nil;
     dispose(temp);
     find_term(req^.t_id,act_tr);
     if (act_tr^.mode=wtg)
     then
       act_tr^.mode:=act;
     if (act_tr^.mode=rstwtg)
     then
       act_tr^.mode:=rst;
     act_tr^.hold_list:=act_tr^.hold_list + [r^.r_id];
     act_tr^.focus:=0;
     remove_holders_probes(r,act_tr^.t_id);
     insert_req(req);
     transfer_from_dm(r,act_tr);
   end;
 end; { release_it }

begin { release_all }
  for k:= 1 to data^.numofres do
  if (k in tr^.hold_list)
  then
  begin
    find_res(k,res);
    release_it(res);
  end;
end; { release_all }
```

```
procedure abort_phase(var tr:tr_nodet; var res:res_nodet);
{ puts the specified transaction into abort phase }
var
  ptr1,ptr2:w_req_typet;
  i:integer;
  rdtrans:el_rd_q_t;

begin
  writeln('   DEADLOCK - transaction ',tr^.t_id:2,' is aborted');
  find_res(tr^.focus,res);
  ptr1:=res^.waitq;
  ptr2:=res^.waitq;
  i:=1;
  while ((ptr1^.t_id <> tr^.t_id) and (ptr1<>nil))do
  begin
    ptr2:=ptr1;
    ptr1:=ptr1^.nextw;
    i:=i+1;
  end;
  if (ptr1=nil)
  then
  begin
    writeln
    ('ERROR - transaction ',tr^.t_id:3,' is not in request_Q');
    halt;
  end
  else
  begin
  if (i = 1)
  then
    res^.waitq:=ptr1^.nextw
  else
    ptr2^.nextw:=ptr1^.nextw;
  ptr1^.nextw:=nil;
  dispose(ptr1);
  tr^.focus:=0;
  release_all(tr);
  tr^.mode:=rst;
  tr^.focus:=0;
  tr^.hold_list:=[];
  dispose_list(tr^.prlist);
  dispose_list(tr^.wtg_pr);
  tr^.mark:=0;
  data^.cmpl:=data^.cmpl-1;
  new(rdtrans);
  rdtrans^.nextel:=nil;
  rdtrans^.t_id:=tr^.t_id;
  rdtrans^.req_time:=data^.clock-data^.mean_tr_dur*ln(1-random);
  insert_tr_rd_q(rdtrans);
  end;
end; { abort_phase }
```

```
procedure dispose_prs_from_cycle(var l:probet; sres:integer);
{ disposes all the probes which has come from the cycle }
var
  ptr1,ptr2:probet;

begin
  ptr1:=l;
  ptr2:=l;
  while (ptr1 <> nil) do
    if (ptr1^.sender = sres)
    then
    begin
      if (l = ptr1)
      then
      begin
        l:=ptr1^.next;
        ptr2:=l;
        ptr1^.next:=nil;
        dispose(ptr1);
        ptr1:=ptr2;
      end
      else
      begin
        ptr2^.next:=ptr1^.next;
        ptr1^.next:=nil;
        dispose(ptr1);
        ptr1:=ptr2^.next;
      end;
    end
    else
    begin
      ptr2:=ptr1;
      ptr1:=ptr1^.next;
    end;
end; { dispose_prs_from_cycle }

procedure reinitiate_probes(var r:res_nodet; var t:tr_nodet;
                            wtr,vic:integer);
{ reinitiates some probes after deadlock }
var
  wreq:w_req_typet;
  pr,tpr:probet;
  iterm,vterm:tr_nodet;

begin
  wreq:=r^.waitq;
  while (wreq <> nil) do
  begin
    if (wreq^.start_time < t^.start_time)
    then
      initiate_probe(wreq^.t_id,t^.t_id,r^.r_id);
    wreq:=wreq^.nextw;
  end;
  dispose_prs_from_cycle(r^.probes,wtr);
  pr:=r^.probes;
  while (pr <> nil) do
  begin
    new(tpr);
    tpr^:=pr^;
    tpr^.next:=nil;
    tpr^.sender:=r^.r_id;
    dm_to_tran(tpr,t);
    pr:=pr^.next;
  end;
end; { reinitiate_probes }

procedure resolution(var tid:integer);
{ starts the resolution of a deadlock }
var
  vic_tr,cycle_tr:tr_nodet;
  res:res_nodet;
  prevtr:integer;
  pr,tpr:probet;

procedure transfer_prlist_to_wtg_pr(var prl,wtgl:probet);
{ transfers probes from wtg_pr list to prlist }
var
 ptr1,ptr2:probet;

begin
  ptr1:=prl;
  while (ptr1 <> nil) do
  begin
    new(ptr2);
    ptr2^:=ptr1^;
    ptr2^.next:=wtgl;
    wtgl:=ptr2;
    ptr1:=ptr1^.next;
  end;
end; { transfer_prlist_to_wtg_pr }

begin  { resolution }
  find_term(tid,vic_tr);
  find_res(vic_tr^.focus,res);
  find_term(res^.t_id,cycle_tr);
  prevtr:=tid;
  while (cycle_tr^.t_id<>vic_tr^.t_id) do
  begin
    data^.clock:=data^.clock+(data^.context_switch*4);
    dispose_prs_from_cycle(cycle_tr^.prlist,res^.r_id);
    dispose_prs_from_cycle(cycle_tr^.wtg_pr,res^.r_id);
    transfer_prlist_to_wtg_pr(cycle_tr^.prlist,
                              cycle_tr^.wtg_pr);
    reinitiate_probes(res,cycle_tr,prevtr,tid);
    prevtr:=cycle_tr^.t_id;
    find_res(cycle_tr^.focus,res);
    find_term(res^.t_id,cycle_tr);
  end;
  data^.clock:=data^.clock+(data^.context_switch*4);
  dispose_prs_from_cycle(res^.probes,prevtr);
  abort_phase(vic_tr,res);
  if (res^.t_id <>0)
  then
  begin
    find_term(res^.t_id,cycle_tr);
    pr:=res^.probes;
    while (pr <> nil) do
    begin
      new(tpr);
      tpr^:=pr^;
      tpr^.next:=nil;
      tpr^.sender:=res^.r_id;
      dm_to_tran(tpr,cycle_tr);
      pr:=pr^.next;
    end;
  end;
end; { resolution }

procedure check_wtg_trs;
{ checks the wtg_pr lists of waiting transactions, transfers
  waiting probes into prlists and transmits a copy of them to
  the necessary data managers }
var
  tr,inittr,victr:tr_nodet;
  res:res_nodet;
  pr,prs:probet;
  req:a_req_typet;
  stop:boolean;

begin
  tr:=data^.tr_q_ptr;
  tr:=tr^.nextt;
  stop:=false;
  while ((tr <> nil) and (not stop)) do
  begin
    if ((tr^.mode = wtg) or (tr^.mode=rstwtg))
    then
    begin
      data^.clock:=data^.clock+(data^.context_switch*2);
      pr:=tr^.wtg_pr;
      while (pr <> nil) do
      begin
        tr^.wtg_pr:=pr^.next;
        pr^.next:=nil;
        find_term(pr^.init,inittr);
        find_term(pr^.vic,victr);
          if ((inittr^.start_time<tr^.start_time) or
              (pr^.init=tr^.t_id))
          then
          begin
            if (victr^.start_time<tr^.start_time)
            then
              pr^.vic:=tr^.t_id;
              if (pr^.init = tr^.t_id)
              then
              begin
                data^.numofdeadlock:=data^.numofdeadlock+1;
                resolution(pr^.vic);
                dispose(pr);
                stop:=true;
              end
              else
              begin
                new(prs);
                prs^.init:=pr^.init;
                prs^.vic:=pr^.vic;
                prs^.sender:=tr^.t_id;
                prs^.next:=nil;
                insert_into_prlist(pr,tr);
                if (tr^.focus<>0)
                then
                begin
                  find_res(tr^.focus,res);
                  sendprobe(prs,res)
                end;
              end;
            end
            else
              dispose(pr);
          pr:=tr^.wtg_pr;
        end;
      end;
      tr:=tr^.nextt;
    end;
  end; { check_wtg_trs }

  procedure check_res(var t:tr_nodet; rqr:integer);
  { checks the specified resource to see if it is available or not }
  var
    res:res_nodet;
    req:a_req_typet;

   procedure allocate;
   { allocates the resource to the transaction }
   begin
     res^.t_id:=t^.t_id;
     new(req);
     req^.t_id:=t^.t_id;
     req^.req_time:=data^.clock+
     (random(data^.maxaccdur-data^.minaccdur+1)+data^.minaccdur);
     req^.mode:=cpureq;
     req^.next:=nil;
     t^.focus:=rqr;
     t^.hold_list:=t^.hold_list + [rqr];
     insert_req(req);
   end; { allocate }
```

```
 procedure put_in_wq;
 { puts the request of the transaction into request_Q of the
   data manager because the resource is held by another
    transaction}
 var
   f1,f2,neww:w_req_typet;
   i:integer;
   notfound:boolean;
   tr1,rtr:tr_nodet;

 begin
   if (t^.mode=act)
   then
     t^.mode:=wtg;
   if (t^.mode=rst)
   then
     t^.mode:=rstwtg;
   t^.focus:=rqr;
   new(neww);
   neww^.t_id:=t^.t_id;
   neww^.start_time:=t^.start_time;
   neww^.nextw:=nil;
   f1:=res^.waitq;
   f2:=f1;
   i:=1;
   notfound:=true;
   while ((f1<>nil) and notfound) do
     if (f1^.start_time<neww^.start_time)
     then
     begin
       f2:=f1;
       f1:=f1^.nextw;
       i:=i+1;
     end
     else
       notfound:=false;
   if (i=1)
     then res^.waitq:=neww
     else f2^.nextw:=neww;
   neww^.nextw:=f1;
   find_term(res^.t_id,tr1);
   if (tr1^.start_time>t^.start_time)
   then
     initiate_probe(neww^.t_id,res^.t_id,res^.r_id);
   transfer_prlist(t,res);
   call_check_wtg_trs:=true;
 end; { put_in_wq }

begin { allocate }
  find_res(rqr,res);
  if (res^.t_id=0)
  then
    allocate
  else
    put_in_wq;
end; { allocate }
```

```
procedure count(holdlist:resset; var result:integer);
{ counts the number of resource that the transaction holds }
var
  i:integer;

begin
  result:=0;
  for i:=1 to data^.numofres do
    if (i in holdlist)
    then
      result:=result+1;
end; { count }

procedure resource_request(var tra:tr_nodet);
{ requests a resource for the specified transaction }
var
  reqtime:real;
  req:a_req_typet;
  reqres,cnt:integer;

begin
  reqtime:=random(data^.maxreqtime)+1;
  data^.clock:=data^.clock+reqtime;
  repeat
    reqres:=random(data^.numofres)+1;
  until not(reqres in tra^.hold_list);
  count(tra^.hold_list,cnt);
  tra^.rst_time_arr[cnt+1]:=reqtime;
  tra^.restart_arr[cnt+1]:=reqres;
  check_res(tra,reqres);
end; { resource_request }


procedure initial_activation(var req:a_req_typet);
{ initial activation of a transaction }
var
  tr:tr_nodet;
  reqres:integer;

begin
  data^.numoftrans:=data^.numoftrans+1;
  find_term(req^.t_id,tr);
  tr^.start_time:=req^.req_time;
  tr^.resnum:=random(data^.maxres-data^.minres+1)+data^.minres;
  resource_request(tr);
end; { initial_activation }

procedure terminate(var trm:tr_nodet);
{ terminates the given transaction }
var
  req:a_req_typet;
  reqtime:real;
  i:integer;

begin
  reqtime:=random(data^.maxreqtime)+1;
  data^.clock:=data^.clock+reqtime;
  release_all(trm);
  data^.cumm_tr_dur:=data^.cumm_tr_dur
                   +(data^.clock-trm^.start_time);
  data^.numofcomptrans:=data^.numofcomptrans+1;
  trm^.mode:=act;
  trm^.start_time:=0;
  trm^.resnum:=0;
  trm^.focus:=0;
  trm^.hold_list:=[];
  trm^.prlist:=nil;
  trm^.wtg_pr:=nil;
  trm^.mark:=0;
  for i:=1 to data^.maxres do
  begin
    trm^.restart_arr[i]:=0;
    trm^.rst_time_arr[i]:=0;
  end;
  data^.cmpl:=data^.cmpl-1;
  job_submission(trm^.t_id,data^.think_time);
end; { terminate }

procedure move_wtg_prs(var tra:tr_nodet);
{ moves some of the probes from wtg_pr list to prlist }
var
  pr:probet;
  ter:tr_nodet;

begin
  pr:=tra^.wtg_pr;
  while (pr<>nil) do
  begin
    tra^.wtg_pr:=pr^.next;
    pr^.next:=nil;
    find_term(pr^.init,ter);
    if (((ter^.mode=wtg) or (ter^.mode=rstwtg)) and
        (ter^.start_time<tra^.start_time))
    then
      insert_into_prlist(pr,tra)
    else
      dispose(pr);
    pr:=tra^.wtg_pr;
  end;
end; { move_wtg_prs }

procedure re_started(var req:a_req_typet);
{ requests resources for restarted transactions }
var
  tr:tr_nodet;
  cnt:integer;
  reqres:1..s1_numofres;

begin
  find_term(req^.t_id,tr);
  if (tr^.mark=0)
  then
  begin
    dispose_list(tr^.prlist);
    dispose_list(tr^.wtg_pr);
  end
  else
    move_wtg_prs(tr);
  tr^.mark:=tr^.mark+1;
  reqres:=tr^.restart_arr[tr^.mark];
  data^.clock:=data^.clock+tr^.rst_time_arr[tr^.mark];
  check_res(tr,reqres);
  if (reqres in tr^.hold_list)
  then
  if (tr^.mark<>data^.maxres)
  then
  begin
    if (tr^.restart_arr[tr^.mark+1]=0)
    then
    begin
      tr^.mark:=0;
      tr^.mode:=act;
    end;
  end
  else
  begin
    tr^.mode:=act;
    tr^.mark:=0;
  end;
end; { re_started }

procedure activate_again(var req:a_req_typet);
{ executes restarted transactions }
var
  t:tr_nodet;
  res:res_nodet;
  cnt:integer;

begin
  find_term(req^.t_id,t);
  if ((t^.mode=rst) and (t^.mark=data^.maxres))
  then
  begin
    t^.mark:=0;
    t^.mode:=act;
  end;
  if ((t^.mode=rst) and (t^.restart_arr[t^.mark+1]<>0))
  then
    re_started(req)
  else
  begin
    t^.mode:=act;
    move_wtg_prs(t);
    count(t^.hold_list,cnt);
    if (cnt=t^.resnum)
    then
      terminate(t)
    else
      resource_request(t);
  end;
end; { activate_again }

procedure take_request(var req:a_req_typet);
{ takes a transaction from cc queue }
var
  ptr:a_req_typet;

begin
  if (data^.req_q_ptr<>nil)
  then
  begin
    ptr:=data^.req_q_ptr;
    if (ptr^.next=nil)
    then
      req:=nil
    else
    begin
      req:=ptr^.next;
      ptr^.next:=req^.next;
      req^.next:=nil;
    end;
  end;
end; { take_request }

begin { simulate }
  take_request(request);
  call_check_wtg_trs:=false;
  while ((request<>nil) and
         (data^.numofcomptrans<stop_simulation)) do
  begin
    check_clock(request);
    write(chr(13),'CLOCK :', sys_1^.clock:6:0);
    case request^.mode of
        new        : initial_activation(request);
        cpureq     : activate_again(request);
        rstcpureq  : re_started(request);
        wtgtrck    : check_wtg_trs;
    end;
    dispose(request);
    if call_check_wtg_trs
    then
    begin
      check_wtg_trs;
      call_check_wtg_trs:=false;
    end;
    transfer_bw_qs;
    take_request(request);
    if request=nil
    then
      while (request=nil) do
      begin
        check_wtg_trs;
        take_request(request);
      end;
    data^.clock:=data^.clock+data^.context_switch;
    if (data^.numofcomptrans<>0)
    then
      data^.mean_tr_dur:=data^.cumm_tr_dur/data^.numofcomptrans;
  end;
end; { simulate }
```

Listing of The Main File :

```
{$U+,R+}
{$G512,P512,D-}
program deadlock;
{
  This program simulates the deadlock detection algorithm
  which is modified by S.F.Akgün for a single-site system.
  All the parameters are given in constant form. It is
  written by S.F.Akgün in Fall,1989.
}
const
    s1_numofres=200;
       { number of resources in the system }
    s1_numofterm=50;
       { number of terminals in the system }
    s1_maxres=8;
       { maximum number of resources requested by a process }
    s1_minres=2;
       { minimum number of resources requested by a process }
    s1_mpl=50;
       { multiprogramming level }
    s1_context_switch=1;
       { context switch time - unit of time in the system }
    s1_max_move_time=4;
       { time required to move objects from ready queue to cc
       queue }
    s1_maxreqtime=25;
       { maximum duration between each resource request made by
        a process }
    s1_think_time=200;
       { mean time between transactions created from a terminal }
    s1_minaccdur=15;
       { minimum disk access time for a database object }
    s1_maxaccdur=65;
       { maximum disk access time for a database object }
    stop_simulation=1000;
       { the program terminates when the number of completed
         transactions reaches the value of stop_simulation }
```

```
type
   probet=^probe;
      { points a probe }
   probe=record
          init,vic,sender:integer;
          next:probet
          end;
      { type of a probe }
   w_req_typet=^w_req_type;
      { points an element in request_Q }
   w_req_type=record
             t_id:integer;
             start_time:real;
             nextw:w_req_typet
             end;
      { type of an element in request_Q }
   reqmode=(new,cpureq,rstcpureq,wtgtrck);
      { modes of the objects in cc queue }
   a_req_typet=^a_req_type;
      { points an element in cc queue }
   a_req_type=record
             t_id:integer;
             req_time:real;
             mode:reqmode;
             next:a_req_typet;
             end;
      { type of an element in cc queue }
   res_nodet=^res_node;
      { points a resource record }
   res_node=record
           r_id,
           t_id:integer;
           waitq:w_req_typet;
           probes:probet;
           nextr:res_nodet
           end;
      { contains information related to a resource }
   resources=1..sl_numofres;
   resset=set of resources;
   modetype=(act,wtg,rst,rstwtg);
      { possible modes of a transaction }
   tr_nodet=^tr_node;
      { points a transaction record }
   tr_node=record
          t_id:integer;
          mode:modetype;
          start_time:real;
          resnum:integer;
          hold_list:resset;
          focus:integer;
          restart_arr:array [1..sl_maxres] of integer;
          rst_time_arr:array [1..sl_maxres] of real;
          mark:integer;
           { restart_arr,rst_time_arr, and mark are used
             in case of restart. Arrays used to store the
             resources requested and their request times }
```

```
         prlist,
          { contains the probes received by the transaction }
         wtg_pr:probet;
          { contains the probes received by the transaction
           when its waiting for CPU }
         nextt:tr_nodet
         end;
     { contains information related to a transaction }
  el_rd_q_t=^el_rd_q;
  el_rd_q=record
          t_id:integer;
          req_time:real;
          nextel:el_rd_q_t;
          end;
     { an element of ready_Q }
 data_pack_type=^data_pack;
 data_pack=record
           numofres,
           numofterm,
           maxres,
           minres,
           mpl,
           cmpl,
           context_switch,
           av_move_time,
           maxreqtime,
           think_time,
           minaccdur,
           maxaccdur : integer;
           sys_clock,
           clock,
           cumm_tr_dur,
           numofprobes,
           numoftrans,
           numofcomptrans,
           mean_tr_dur,
           numofdeadlock : real;
           res_q_ptr :res_nodet;
           tr_q_ptr : tr_nodet;
           req_q_ptr : a_req_typet;
           ready_q_ptr : el_rd_q_t;
           end;
    { contains all the parameters of a modeled single-site system }

var
 sys_1 : data_pack_type;

{$I var_init.pas}
{$I initialize.pas}
{$I simu.pas}


begin { main }
 randomize;
 new(sys_1);
 var_init(sys_1);
 initialize(sys_1);
```

```
  simulate(sys_1);
  writeln;
  writeln('PARAMETERS :');
  writeln('===========');
  writeln('total number of resources :',sys_1^.numofres:4);
  writeln('total number of terminals :',sys_1^.numofterm:4);
  writeln('max. resource request :',sys_1^.maxres:3,
                      min. resource request :',sys_1^.minres:3);
  writeln('multiprogramming level (mpl) :',sys_1^.mpl:3);
  writeln('think_time :',sys_1^.think_time:4);
  writeln('max. disk access time :',sys_1^.maxaccdur:3,
                  min. disk access time :',sys_1^.minaccdur:3);
  writeln;
  writeln('RESULTS OBTAINED :');
  writeln('=================');
  writeln('response_time:',
                 (sys_1^.cumm_tr_dur/sys_1^.numofcomptrans):6:0);
  writeln('number of probes sent/10000 units of time:',
           (sys_1^.numofprobes*10000)/sys_1^.clock:7:0);
  writeln('number of deadlocks happened/10000 units of time :',
           (sys_1^.numofdeadlock*10000)/sys_1^.clock:6:0);
  writeln('THROUGHPUT');
  writeln('(number of transactions completed/10000 units of time):',
          (sys_1^.numofcomptrans*10000)/sys_1^.clock:6:0);
end. { main }
```

BIBLIOGRAPHY

1. Mukul K. Sinha and N. Natarajan, "A priority based distributed deadlock detection algorithm," *IEEE Transaction on Software Engineering*, Vol. SE-11, No. 1, pp. 67-80, January 1985.

2. Alok N. Choudhary, Walter H. Kohler, John A. Stankovic, and Don Towsley, "A modified priority based probe algorithm for distributed deadlock detection and resolution," *IEEE Transaction on Software Engineering*, Vol. 15, No. 1, pp. 10-17, January 1989.

3. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Reading, Massachusetts: Addison-Wesley, 1987.

4. D. Menasce and R. Muntz, "Locking and deadlock detection in distributed databases," *IEEE Transaction on Software Engineering*, Vol. SE-5, No. 3, May 1979.

5. K. Mani Chandy and J. Misra, "A distributed algorithm for detecting resource deadlocks in distributed systems," in *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada), ACM, New York, pp. 157-164, August 1982.

6. Edgar Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys*, Vol. 19, No. 4, pp. 303-328, December 1987.

7. T. Hermann and K. Mani Chandy, "A distributed procedure to detect AND/OR deadlock," Tech. Rep. TR LCS-8301, Dept. of Computer Science, Univ. of Texas, Austin, Tex., 1983.

8. G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection," Tech. Rep. TR 83-558, Cornell Univ., Ithaca, NY., 1983.

9. E. Gafni, "Perspectives on distributed network protocols: A case for building blocks," in *IEEE Military Communications Conference* (Monterey, Calif.), IEEE, New York, pp. 1.1.1-1.1.5, 1986.

10. Virgil D. Gligor and Susan H. Shattuck, "On deadlock detection in distributed systems," *IEEE Transaction on Software Engineering*, Vol. SE-6, No. 5, pp. 435-440, September 1980.

11. Ron Obermarck, "Distributed deadlock detection algorithm," *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 187-208, June 1982.

12. Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol. 21, No. 7, pp. 558-565, July 1978.

13. K. Mani Chandy and Leslie Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63-75, February 1985.

14. Rakesh Agrawal, Michael J. Carel, and Miron Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems*, Vol. 12, No. 4, pp. 609-654, December 1987.

REFERENCES NOT CITED

Agrawal, R., M. J. Carey, and L. W. McVoy, "The performance of alternative strategies for dealing with deadlocks in database management systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, pp. 1348-1363, December 1987.

Blazewicz, J., J. Brzezinski, and G. Gambosi, "Time-stamp approach to store-and-forward deadlock prevention," *IEEE Transactions on Communications*, Vol. COM-35, No. 5, pp.490-495, May 1987.

Bochmann, G. V., "Delay-independent design for distributed systems," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, pp. 1229-1237, August 1988.

Cidon, I., J. M. Jaffe, and M. Sidi, "Local Distributed deadelock detection by cycle detection and clustering," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 3-14, January 1987.

Elmagarmid, A. K., and A. K. Datta, "Two-phase deadlock detection algorithm," *IEEE Transactions on Computers*, Vol. 37, No. 11, pp. 1454-1458, November 1988.

Elmagarmid, A. K., N. Soundararajan, and M. T. Liu, "A distributed deadlock detection and resolution algorithm and its correctness proof," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1443-1452, October 1987.

Hoare, C. A. R., "Communicating sequential processes," *Communications of the ACM*, Vol. 21, No. 8, pp. 3-11, August 1978.

Maekawa, M., Oldehoeft A.E., Oldehoeft R.R. *Operating Systems: Advanced Concept*. The Benjamin/Cummings Pub. Co., 1987.

Peterson, J. L., and Abraham Silberschatz. *Operating System Concepts*.Reading, Massachuttes, Addison-Wesley Pub. Co., 1985.

Wuu, G. T., and A. J. Bernstein, "False deadlock detection in distributed systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 8, pp. 820-821, August 1985.