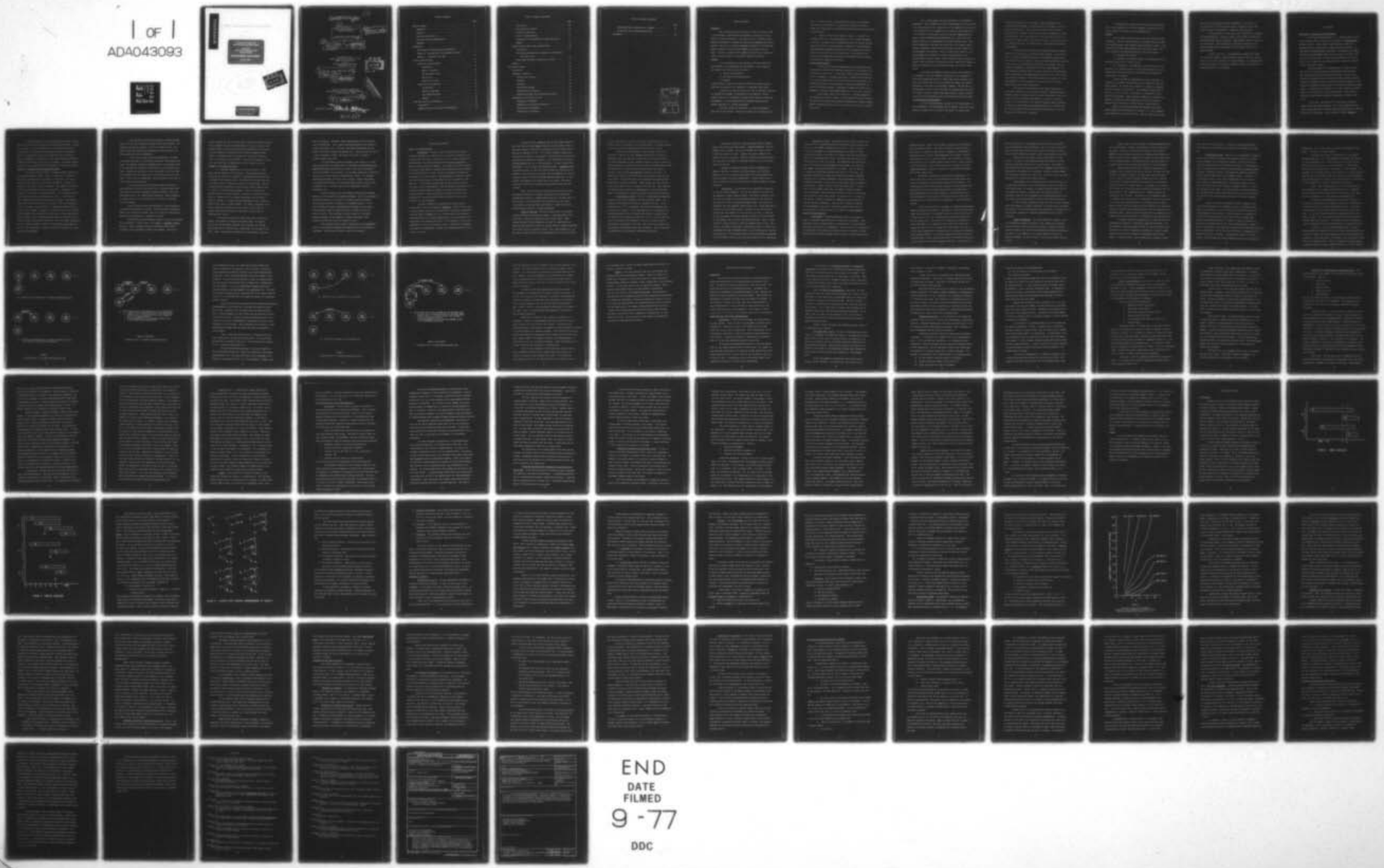
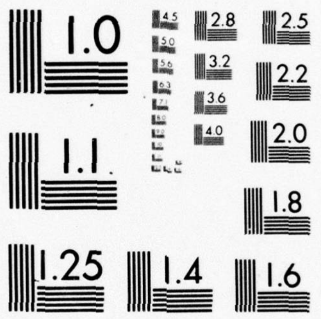
AD-A043 093 ILLINOIS UNIV AT URBANA-CHAMPAIGN CENTER FOR ADVANCED--ETC F/G 9/2
RESEARCH IN NETWORK DATA MANAGEMENT AND RESOURCE SHARING. SYNCH--ETC(U)
MAR 76 P A ALSBERG, G G BELFORD, S R BUNCH DCA100-75-C-0021
UNCLASSIFIED UIUC-CAC-TN-76-185 CCTC-WAD-6503 NL

1 OF 1
ADA043093



END
DATE
FILMED
9-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

Center for Advanced Computation

CAC Document Number 185
CCTC-WAD Document Number 6503

Research in Network Data Management and Resource Sharing

Synchronization and Deadlock

March 1, 1976

(2)

(18) CAC Document Number 185
CCTC-WAD Document Number 6503 (19)

(6)

Research in Network Data Management and Resource Sharing.

Synchronization and Deadlock.

(9) Research rept.,

(10)

by

Peter A. Alsberg, Geneva G. Belford, Steve R. Bunch, John D. Day, Enrique Grapa, David C. Healy, Edwin J. McCauley, David A. Willcox

Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
of the
Defense Communication Agency
Washington, D.C.

(14) UIUC-CAC-TN-76-185, CAC-185

(15) under contract
DCA100-75-C-0021

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

(11) 1 March 1976

(12) 87p.

Approved for release: [illegible]
Peter A. Alsberg, Principal Investigator

1473
407 227

LB

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Executive Summary

Background

This document presents the results to date of a research study of the problems of synchronization and deadlock, particularly as they arise in a network environment. The study is part of a larger, comprehensive investigation of problems in network data management and resource sharing. The goal is to develop techniques applicable to the World-Wide Military Command and Control System (WWMCCS) Intercomputer Network (WIN). The work is supported by the WWMCCS ADP Directorate, Command and Control Technical Center, of the Defense Communications Agency.

Overview

In the introduction to this document, we briefly define and give examples of both synchronization and deadlock. The *remainder* of the document is divided into three sections:

1) Process Synchronization
2) Data Base Access Synchronization
3) Deadlock

In each section we discuss the technical problems and assess possible solutions, particularly with reference to a distributed environment.

In general, we find that there is no lack of available techniques for handling the problems of synchronization and deadlock. However, previous study of the techniques has largely been limited to single-site environments. We have identified those techniques which seem most readily applicable to a network environment.

Extending single-site techniques to a network necessarily gives rise to new problems. Among these problems are resiliency and the lack of a central control. We have made some progress on developing solutions to these problems. In particular, a promising resiliency scheme and an algorithm for decentralized deadlock detection are presented in this document.

However, further work is needed. Analysis of how well the techniques work (e.g., cost and response) in a distributed environment is required. We expect that the mathematical models we are now developing will be useful in such an analysis. Furthermore, the experimental distributed data management system that we are designing will provide a vehicle for practical tests of the techniques. More definitive conclusions and recommendations on deadlock and synchronization techniques may therefore be expected in the future.

Process Synchronization

The section on process synchronization first reviews the basic techniques for ensuring that processes concurrently in the system do not interfere with one another. In one way or another, these techniques all involve "semaphores" - variables which can be accessed (read and altered in a strictly defined way) by all the processes to be synchronized. By modifying the values of such variables, a process can communicate its state to the other processes. Conversely, by examing the values of the semaphores, a process can determine whether or not it is safe to proceed.

Many schemes exist in the literature for defining semaphores and primitives to operate on them. In this document we briefly describe only a few of the schemes most frequently implemented in computer systems: Dekker's algorithm, Dijkstra's P and V operations, test and set, and wait on an event.

All of these schemes run into difficulties in a distributed environment. First, semaphores reside in shared memory; all the processes to be synchronized must have ready access to the semaphores. Second, even if it is feasible for the semaphores to be stored at one reliable site on the network, the time delays involved in network communications are likely to make synchronization prohibitively slow. Third, the time delays, which may vary considerably, have other adverse affects. They make the problem of races much more severe than it is within a single system. Furthermore, extraordinarily long delays may be virtually indistinguishable from failures. Serious trouble can be caused if a process attempts error recovery on a (remote) process which is still working properly, albeit slowly. Fourth, problems of errors become more severe. Much care needs to be taken to see that communication among processes is resilient to lost and delayed messages, and to other common errors.

We discuss two schemes for multi-processor synchronization in this document. The first, Lamport's algorithm, was not designed for a network environment and appears to suffer from serious resiliency problems. The second, which we call the Alsberg-Day scheme, was recently developed to solve some of the resiliency problems. The key idea is to provide that at least two hosts know what is happening before an action is allowed to proceed. Network time delays however, remain a problem which may become particularly accute in synchronizing data base usage.

Data Base Access Synchronization

As in all synchronization, the goal is to keep processes, or user transactions, from interfering with one another. Each user should have the illusion that he is alone on the system. To achieve this "single-user illusion", it is possible to apply a standard process synchronization technique. Certain questions arise, however, which are specific to data bases. One of these is the data "level" at which exclusive use is assigned. It may be that usage is assigned on a record-by-record basis. Or, at the other end of the scale, only one user at a time may be allowed access to the entire data base. The choice of level is a difficult decision. We discuss some of the tradeoffs which should be considered.

If the data base is distributed throughout a network, network delays affect the decision on synchronization technique. In particular, there is a need to minimize the number of synchronizing messages which must be sent across the network. Looking at current single-site techniques in the light of this requirement, we find that the frequent, explicit setting of semaphores or locks appears to be too time-consuming. The most promising techniques seem to be those in which only the transaction requests themselves, with perhaps some information on the order in which they should be performed, are transmitted across the network. The local data managers are then responsible for setting locks or enforcing a usage protocol that avoids interference.

The question of maintaining an order among transactions is particularly important in a multi-copy environment. Applying updates in different orders to different copies can cause inconsistencies among the copies. One way to maintain the ordering is to attach a "sequence number" to each transaction. The concept of sequence number assignment is specific to distributed data management and hence is relatively new. Two of the strategies which we discuss have been developed at the CAC, and research on this topic is ongoing.

We conclude that at this time the state of the art does not permit the recommendation of a "best" technique for data base access synchronization, particularly in the distributed case.

Deadlock

The last major section of this report deals with the problem of deadlock. That is, several processes may mutually block each other from further progress, each holding a resource needed by another. This problem has been very heavily studied over the last six years. The basic techniques for handling deadlock fall into three classes:

1) Detection and recovery.

2) Avoidance, or elaborate schemes for assigning resources to processes only when it is "safe" to do so.

3) Prevention, or the imposition of some discipline on the processes (e.g., requiring them to ask for needed resources in a prescribed order) which precludes the circular blocking configuration characteristic of deadlock.

We have assessed the applicability of these techniques in a distributed environment, and particularly with reference to data base access control. We conclude that avoidance is probably too expensive and time-consuming to be practical. The prevention schemes also may impose too much of a burden on the system or, alternately, on the application programmer. Simple detection schemes may be feasible in a network. We propose one such scheme which, unlike one previously discussed in the literature, does not require a central monitor.

There is also the possibility that deadlocks occur infrequently enough that the problem may be ignored; i.e., that a deadlock may be handled like any process failure. Several studies exist in the literature on the expected frequency of deadlock. As one might intuitively expect, as data bases grow larger, contention for the same items (and hence the probability of deadlock) should decrease.

Existing computer systems tend to use a combination of techniques for handling deadlock. Different schemes or disciplines are applied to handling different types of resources. Some possible sources of deadlock are even ignored. The practicality of combining techniques is worth keeping in mind, since there appears to be no single "best" way to handle deadlock.

Clearly, the choice of synchronization mechanism will impact on the deadlock problem. Synchronization techniques need to be studied with one eye on their potential for causing deadlock. If synchronization is handled properly, the need for a separate concern with techniques for handling deadlock can be minimized.

Introduction

The Problems of Synchronization and Deadlock

In any multiprocessing system - i.e., computer system in which several processes may be concurrently in the system - synchronization becomes a concern. Simply stated, synchronization is some sort of provision for the noninterference of these concurrent processes. In some cases the noninterference is automatic. For example, in a single-processor multi-programming system, only one process is actually using the CPU at any given time. In other cases interference can be quite subtle; e.g. an update to a data base may cause a concurrent retrieval to give inconsistent results.

The basic approach to synchronization is to make a process wait until it can proceed without interfering with other processes. This waiting can be effected through various means. One way is for processes to communicate with one another by setting values of special variables ("semaphores") which may be read by the other processes. This approach is discussed in detail below in the section on process synchronization. A variation on this approach is to "lock" resources to reserve them for the exclusive use of the locking process. (This is only a variation, since locking is generally implemented by setting a lock-bit.)

In any case, the progress of a process may be blocked by having to wait for needed resources. Two or more processes may then reach a state where none can proceed since each is waiting for a resource held by one of the others. Such an impasse is called a deadlock.

In studying solutions to the synchronization problem, one must therefore keep one eye on how they affect the deadlock problem. Although the solutions to the two problems can to a large extent be studied independently, any engineering solution must consider both problems. Thus it is appropriate to merge the studies into this one document. Before proceeding to considerations of the synchronization and deadlock problems in some detail, we will give two examples. Particular emphasis is placed on the complexities of the problems in a network environment.

Example 1: Transmission of Data Between Processes

This is the classic producer - consumer problem, discussed at some length by Dijkstra [1968]. There are assumed to be a number of processes "producing" data or information of some kind. This data is then to be sent to other processes which "consume" it. Suppose, for simplicity, that there is only one consumer. For example, the consumer could be an output device, such as a line printer. Transmission is assumed to be by way of a finite-sized buffer. Thus producers put data into the buffer as long as the buffer is not full. If the buffer is full, producers must wait until more space is provided. The consumer removes data from the buffer until it is empty. Then the consumer must wait for a producer to put more data in. Communication among the producers and the consumer is necessary to ensure that they do not interfere with one another when putting data into the buffer or when removing data from the buffer. Thus, considerable synchronization is needed to guarantee the smooth and fault-free operation of this very simple transmission process. Dijkstra presents a solution to the problem using semaphores and his P and V operations. Dijkstra's solution is discussed in a later section of this document.

If the operations which involve waiting are handled properly, there is no way that a deadlock can arise in this simple example. The competition among the processes is only for a single resource - the buffer. Only when processes are competing for several resources can they entangle each other in deadlocks.

If the buffer is used for two-way transmission, it becomes two separate resources - an I/O buffer at each end. Then deadlock can occur. For example, consider the simplest case of two processes, A and B, each putting into the buffer data which are to be extracted by the other. Suppose that the buffer is full, and neither A nor B will read any information from the buffer until it is able to put more into the buffer. Then A and B are deadlocked. Each is waiting for the other to open up the needed buffer space.

In a network setting, the processes sending information to each other may be located at different sites. The paths between them involve not only network transmission lines but buffers at the store-and-forward nodes of the communications subnetwork. Simple semaphores no longer suffice to tell a process when it is "safe" to send data across the network.

Packet switching causes some mitigation of data-transmission synchronization problems in a network. Since each packet independently makes its way through the network, there is no need for synchronization among the actual producers or consumers of the data.

Unfortunately, packet switching, though it solves some problems, has the potential to create others. For example, a reassembly lock-up may occur. This is a deadlock caused by filling the buffer space available for the reassembly of packets into messages. Portions of several messages can fill the buffer space. None of the partial messages can be completed and read out because there is no space to read in the missing packets. Notice that this serious problem arises from altering just one assumption in the easily solved producer-consumer problem; namely, the assumption that the consumer will extract any arbitrary "unit" of information from the buffer. As soon as the consumer can wait for a particular unit or combination of units, deadlock can occur.

Example 2: Accessing a Data Base

If concurrent processes can read from and write to a data base, they may interfere with one another. This interference could cause both the retrieval of incorrect or inconsistent data and the loss of integrity within the data base. The retrieval of incorrect data can occur when a writer is altering the data as it is being read. Loss of internal data base integrity can occur when two writers interfere with one another. To prevent these problems, a writer is usually given exclusive access to the data base (or a portion of it) while he is modifying it. Readers may all be reading concurrently, but not while a writer is writing. In a single-site data management system, synchronization of data usage is readily handled. For example, semaphores can be used or locks can be attached to units of data. (The problem and possible solutions are discussed in some detail in the section on synchronization, below.)

If processes can block each other from access to data units, however, the potential for deadlock may become large. For example, suppose that two Navy supply officers, Jones and Smith, are charged with keeping track of food and fuel, respectively, for the ships of the fleet. Jones, in updating the information on food supplies, locks the file on the Monitor. Meanwhile, Smith, updating data on fuel, locks the file on the Merrimac. Jones, before unlocking the Monitor file, needs to consult the Merrimac file. Similarly, Smith needs information from the Monitor file before releasing the Merrimac file. Smith and Jones are now in a state of deadlock. Neither can proceed, since each is tying up a resource needed by the other.

In a distributed, network environment, where access to the data base can take place from remote sites, the problems of synchronization and deadlock become much more difficult to solve. The source of the difficulty lies primarily in two factors - the increased likelihood of errors and the lack of a central system for monitoring and control. In a network, synchronizing messages are easily lost or delayed. Without some scheme for ensuring the resiliency of the synchronization protocol, there can be no guarantee that a data base user will not inadvertently destroy a colleague's work.

The local system at the site holding the data base can perhaps maintain the data integrity by standard techniques. But if, for purposes of increased availability, two copies of the data are held at different sites, maintaining their consistency is a formidable problem. If one site is assigned the task of maintaining the integrity of both copies, provision must be made for switching control to another site if the first site fails. This switching of control itself is a nontrivial problem, requiring the development of resilient techniques. Thus solving one problem may only cause another to arise.

This document reports on what is perhaps the first comprehensive attack on the problems of deadlock and synchronization in a distributed environment. Many questions, however, remain to be answered.

Process Synchronization

Single-site Synchronization

Introduction. The basic reason for synchronization among concurrent processes is to ensure that the cooperating processes do not interfere with each other. For the most part, of course, processes are noninterfering. Only while executing certain critical sections may processes interfere with one another. For example, the critical section of a process which updates a data base might include the actual updating of the indices to the data base, but not the preliminary decoding and processing of the update request. Synchronization is required to ensure that two or more processes do not update the data base indices at the same time. If more than one process was allowed to update an index, the result could be indeterminate. Thus the essential information to be transmitted among processes is whether or not they are within their critical sections. Clearly the notion of "critical section" can be generalized to include any section whose initiation or completion is "critical" to other processes.

The basic technique for synchronizing processes within a single computer system is by means of semaphores. (We use the term here in a more general sense than the restricted one of Dijkstra "semaphores".) In simple terms, a semaphore is just a variable which can be accessed (read and altered in a strictly defined way) by all the processes to be synchronized. By modifying the values of such variables, a process can communicate its state to the other processes. Conversely, by examining the values of the semaphores, a process can determine whether or not it is safe to proceed.

As generally used, semaphores take on only integer values; in many cases they are simply binary (taking on only the values 0 and 1). The basic arithmetic operations on semaphores can then be restricted to addition or subtraction of one. It is clearly possible, unless other restrictions are imposed, for processes to interfere with one another in operating on the semaphores themselves. Such interference would cause misunderstandings and loss of synchronization. In order to avoid such interference - and to obtain other desirable effects - primitives (uninterruptible sections of code) are usually defined for operation on the semaphores. The restriction is imposed that the semaphores can only be accessed by means of these well defined semaphore primitives. In some systems, the semaphores are handled only by the system. The process "waits on an event" and is notified by the system when the event has occurred.

Much of the literature on process synchronization deals with alternative definitions of semaphore primitives and how they may be implemented and applied. Although details may vary, the basic ideas remain the same. We will therefore only look briefly at the classic synchronization techniques before proceeding to consider the process synchronization problem in a network environment.

Dekker's algorithm. The first valid solution to the mutual exclusion problem was discovered by Dekker. A good discussion of his algorithm is contained in Dijkstra's classic paper [1968]. Three integer variables (binary semaphores) are used for communication between two processes both of which are cycling through a program containing one critical and one noncritical section. Two of the variables (C_1, C_2) are used to indicate whether the corresponding process is in its critical section. Thus process 1 turns C_1 on and off, and looks at C_2 to see whether process 2 is already in its critical section. The third variable is used to resolve an indeterminacy when both C_1 and C_2 are on.

There are two disadvantages to this solution. One is its complexity. Ideally, only one semaphore should suffice to pass the information back and forth that a process is in its critical section. The other disadvantage is that, whenever a process is ready to enter its critical section, it continually tests the other C variable until the test is satisfied (i.e., until the other process has left its critical section). This so-called busy waiting is wasteful of CPU time. It would be much better if the process could wait passively until the test is satisfied. As mentioned above, most techniques use uninterruptible primitives to access the semaphore. Dekker assumes only that two processes cannot read or write a location at the same time. This algorithm is therefore useful, despite its disadvantages, in those applications where it is impossible to allow more complex uninterruptible primitives.

The primitives P and V. One set of uninterruptible primitives consists of the P and V operations defined by Dijkstra [1968]. The operation P(S) decreases the value of a semaphore S by one. If the resulting value of S is non-negative, the process performing the P(S) is permitted to proceed. If the resulting value of S is negative, the process performing the P(S) is blocked and booked on a waiting queue. The operation V(S) increases the value of S by one. If one or more processes are waiting on the semaphore S, then one is removed from the waiting queue and permitted to proceed. Both P and V are assumed to be (in Dijkstra's terminology) indivisible operations. That is, each must be implemented in an inseparable, uninterruptible fashion.

Using these primitives, Dijkstra readily solves the simple mutual exclusion problem which Dekker's algorithm addresses. Only one semaphore (S) is needed. Each process precedes its critical section by P(S) and follows it by V(S). Indeed, this simple scheme is not limited to just two processes. Dijkstra's scheme will guarantee that no more than one of any number of concurrent processes is in its critical section at any time.

Dijkstra's primitives certainly avoid the complexities of Dekker's algorithm. As defined, the busy waiting is also avoided. Whether or not the P operation does involve busy waiting depends upon how the primitive is actually implemented. It would clearly help to have a primitive of this type built into the computer system as a single operation.

Test and set. This operation is the indivisible testing and setting of a binary semaphore. Test and set operations are built into the hardware of modern computer systems. They permit a process to test the value of a memory cell (the semaphore) and write a new value into the same cell in one, uninterruptible memory cycle. Test and set is frequently used with semaphore values zero and one. A zero means that no process is in its critical section. To enter its critical section, a process tests the semaphore for zero and sets it to one. If the test for zero succeeded, the process proceeds to enter its critical section. If the semaphore was a non-zero, the process must wait. When a process leaves its critical section, it sets the semaphore back to zero. Test and set, like Dekker's algorithm, is busy when waiting. In actual use, a programmer will take considerable pains to avoid cycling on the test and set instruction. The techniques used to avoid busy waiting are dependent on the facilities offered by the operating system and on the application.

Waiting on an event. An apparently sophisticated synchronization technique which is implemented (with some variations) in many computer systems (e.g., UNIX, B6700, and Multics) is the wait on event. The idea is that the programmer is provided considerable flexibility in naming events and specifying, at any point in a process, that the process should wait until a designated event occurs. The system will then wake up the process after the event has occurred. The problem with this solution is that if the event has occurred before the wait begins, the system will not remember it and the waiting will go on forever. That is, there is no queueing of events which processes may wait on in the future. The Burroughs system does get around this problem by providing a HAPPENED function, but it is rather cumbersome. This cumbersomeness is inherent in systems using wait on event, since the same events may be periodically reissued (e.g., a buffer full and a buffer available event). In this case it is not enough to know that, for example, a buffer available event has happened. The process needs to know the sequence of the prior event relative to other prior events. In contrast, P and V, as well as test and set, provide a mechanism for remembering events through the explicit setting of semaphores. Thus the extra flexibility supplied to the programmer by the wait on event also causes him difficulty if the event may occur before the wait (i.e., there is a race condition).

Multi-site Synchronization

Introduction. Problems immediately arise if one attempts to apply the techniques described above to a multi-site environment. First, semaphores reside in shared memory; all the processes to be synchronized must have ready access to the semaphores. Second, even if it is feasible for the semaphores to be stored at one reliable site on the network, the time delays involved in network communications are likely to make synchronization prohibitively slow. Third, the time delays, which may vary considerably, have other adverse affects. They make the problem of races much more severe than it is within a single system. Furthermore, extraordinarily long delays may be virtually indistinguishable from failures. Serious trouble can be caused if a process attempts error recovery on a (remote) process which is still working properly, albeit slowly. Fourth, problems of errors become more severe. Much care needs to be taken to see that communication among processes is resilient to lost and delayed messages and to other common errors.

We discuss two multi-site synchronization schemes here. In their present form, neither of these schemes will provide general solutions to the problems of network partitioning and error recovery. Although there are some obvious specific solutions to these problems, more work needs to be done to determine what options are available and to what degree general solutions are possible. A detailed discussion of these problems and their solutions is beyond the scope of this paper. A separate and more detailed paper is in preparation. A brief discussion follows.

One of the most difficult problems for a network synchronization scheme to handle is partitioning. (The network is said to partition when subnet failure results in the network's being cut into two or more sets of hosts in such a way that communications are maintained within sets but not between sets.) The difficulty in handling this situation is that a host which cannot communicate with a remote host cannot tell whether the host is dead or the network has partitioned. In fact, it is only after communication is restored that it is possible to determine which situation existed. For instance, suppose that several hosts are synchronizing to use some resource and that one of them has access and is in its critical section when the network partitions. To hosts on one side of the partition it will appear that the host has died and that recovery procedures should be instituted. But to the hosts on the other side of the partition the host is still alive in his critical section, and all is well except that several of their cohorts appear to have died.

There are two cases where network partitioning can become critical. The first is the example mentioned above where the network partitions with some process in its critical section. The second is when a network partition causes copies of the semaphore values (which are duplicated for resiliency) to be separated. Various resiliency schemes can be utilized to handle this latter case. In fact it is this case that is addressed by the two solutions which we discuss.

There appears to be no general solution in the first case. If a process does fail while in its critical section, some sort of error recovery must be done. At present, it appears that this recovery will be very application specific. However, more work needs to be done to determine if there are basic strategies that synchronizing hosts can use with a given synchronizing scheme to recover from errors and failures.

In the existing literature, we have found only one paper [Lamport, 1974] that attempts to provide a solution to a multi-processor synchronization problem. We will briefly discuss that paper below. Following that, we will discuss a technique for resilient synchronization that we have developed.

Lamport's algorithm. The problem addressed by Lamport is that of synchronizing a number of processors, each running cyclic programs with two parts, one critical and one noncritical. (This is the multi-processor analog of the classic problem addressed by Dekker and Dijkstra; see above.) It is tempting to try to adapt Lamport's solution to a network environment. We shall see that it is not clear whether this can be done.

Lamport adopts the usual approach of assuming communication by way of shared memory. However, each processor can only write into its own memory, although it can read from the memory of any processor. Whether or not a processor may enter its critical section depends upon whether a priority-setting number it has "chosen" is lowest among those of the waiting processes. (The waiting is busy, and thus wasteful of processor time if other processes may otherwise be run concurrently.)

The principal difficulty with Lamport's algorithm is his assumption that "a processor may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts." Clearly, there are serious resiliency problems with such a solution. The resiliency is highly dependent upon the nature of the operations carried out inside the critical section. For example, if a processor fails inside its critical block while entering or deleting an item in a data structure - and it immediately exits - pointers could be left pointing all over the place. Furthermore, as the algorithm stands, it is impossible for other processors to determine if the last one to be in its critical block exited normally or if it exited due to a failure. Finally, other resiliency problems arise since failures in a network may include not only host (or processor) failures but also failures of communication links. Lamport's algorithm assumes that the semaphores at the remote hosts are always available for inspection by the other processes.

In summary, Lamport's algorithm does not completely solve the synchronization problem in a distributed computing environment. The main obstacle to adopting Lamport's scheme in a network context is the long delays that would be necessary to synchronize. Lamport's scheme requires at least 4(N-1) messages to be exchanged, where N is the number of cooperating hosts. Even assuming some concurrency it can't take less than 2N message times. It remains an open question whether, with appropriate modifications, the scheme could be turned into a viable one.

The Alsberg-Day model. Several of the disadvantages of Lamport's scheme can be overcome by applying the Alsberg-Day scheme for resilient protocol design to the synchronization problem. The key idea of this resiliency scheme, which will be described in detail in a forthcoming report, is to provide that at least two hosts know what is happening before an action is allowed to proceed. The Alsberg-Day scheme can be used to provide resilient maintenance of the semaphore values. In addition to the resiliency scheme, several sets of primitives must be provided to allow creation, attaching/detaching, access control, message security, and error recovery due to user software or hardware failures. It is beyond the scope of this paper to go into the details of these primitives. We only mention them to assure the reader that they have been considered. (The details of this synchronization strategy will be given in another report.)

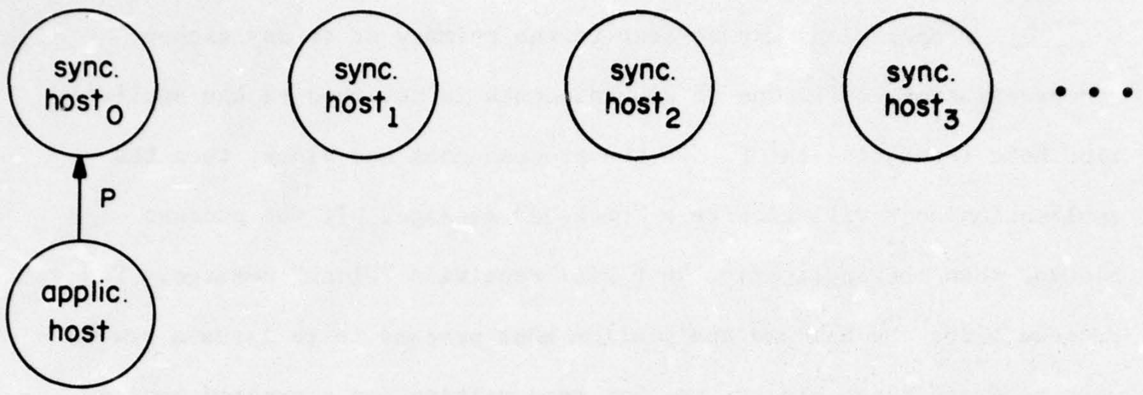
In order to see how the Alsberg-Day scheme is applied to the synchronization problem, let us consider, at least for purposes of description, that there is a set of hosts which do nothing but mediate the synchronization of user processes. (This may appear to be somewhat excessive for the practical case; but if one is really concerned about having reliable synchronization, it is unwise to make it susceptible to the kind of environment found in the typical application host. However, there is nothing about this scheme that requires that the synchronizing function be in a devoted host.) One of the hosts of this set is designated as the primary and the rest are backups. The backups are ordered in a linear fashion. We will assume that recovery schemes are defined to maintain this organization. Let us look at how a P operation is performed in this scheme. (V is done in a similar manner.)

P operations may be sent to the primary or to any backup synchronization host. One of two responses is returned to the application host requesting the P. If the process does not block, then the application host will receive a "proceed" message. If the process blocks, then the application host will receive a "block" message. The procedure for the blocked application host process is to issue a new network read and to go blocked on that read waiting for a proceed message. That proceed message will normally follow the issuance of a V operation on the same semaphore by a different application host process.

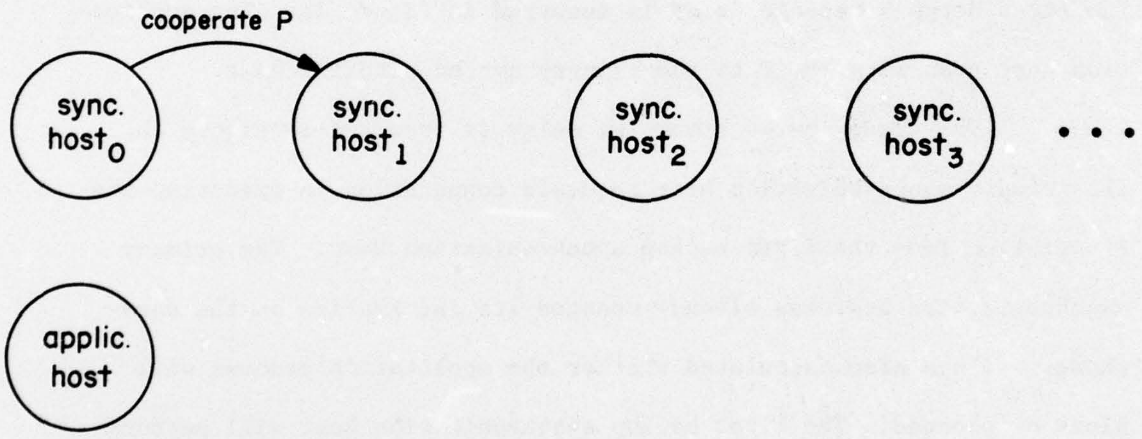
Figure 1 shows the message flow for a P operation which has been transmitted to the primary synchronization host of a semaphore. The first network message delay is incurred in figure 1a. The application host transmits the P to the primary synchronization host.

The second network message delay is incurred in figure 1b. The primary synchronization host requests cooperation in executing the P operation from the first backup synchronization host. The primary synchronization host has already updated its information on the semaphore. It has also calculated whether the application process will block or proceed. The first backup synchronization host will perform the same update and calculation. The backup host will be expected to issue the appropriate block or proceed message to the application host.

In figure 1c the third network message delay is incurred. Three messages are transmitted by the first backup synchronization host. In terms of network delay, these messages are essentially simultaneously transmitted. Small improvements in resiliency can be achieved by issuing them in the designated order. First, the synchronization host passes a backup P message to the next synchronization host. At this time only



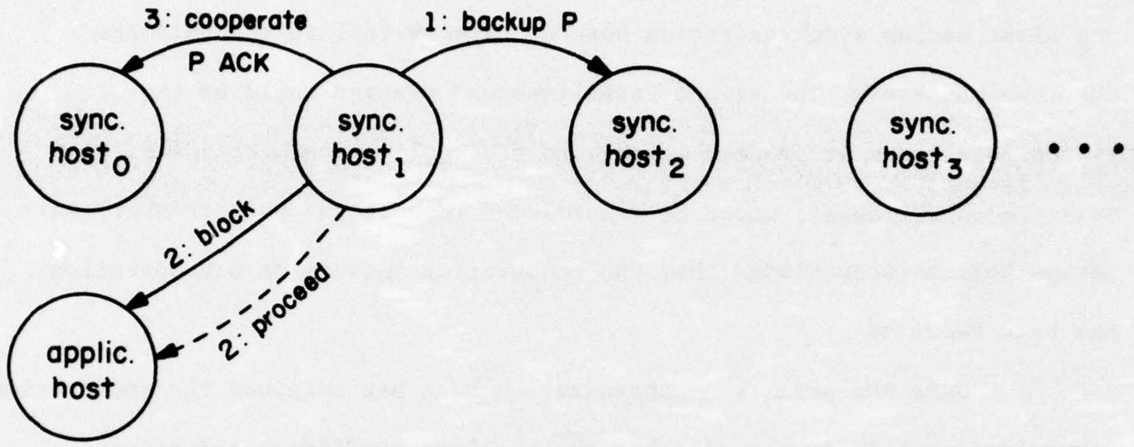
1a: Application host transmits P to primary synchronization host.



1b: Primary synchronization host requests cooperation from the first backup in executing the P operation.

Figure 1

P operation sent to a primary synchronization host



1c: First backup issues three messages in the following order:
1. A backup for a P operation is sent to the next backup.
2. Either a block or a proceed (dotted line) message is sent to the application host.
3. An acknowledgement of the cooperate message is sent to the primary synchronization host.

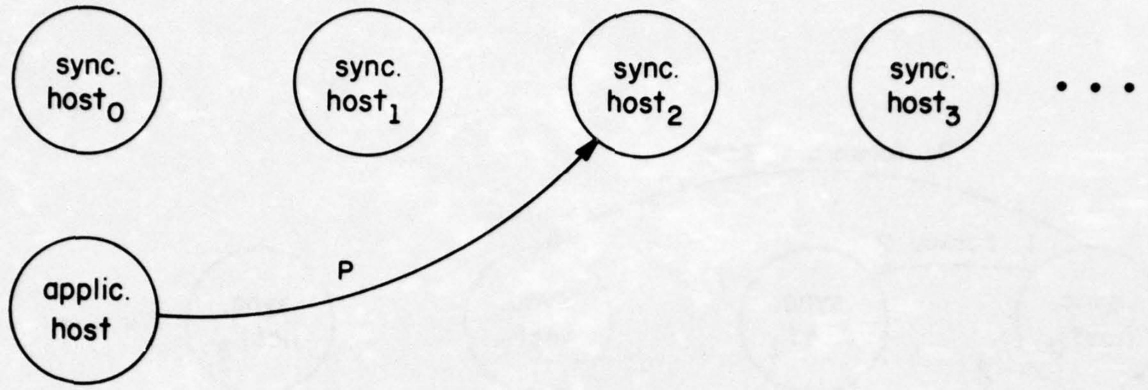
Figure 1 (continued)

P operation sent to a primary synchronization host two synchronization hosts, the primary and the first backup, have positive knowledge of the existence of the P operation. Should the backup message be successfully received at the second synchronization backup host, a third synchronization host would also be aware of the P operation. The third host would be able to assist in recovery should the first backup synchronization host or network fail to transmit the next two messages. The second "simultaneous" message would be the appropriate block or proceed message to the application host. The third "simultaneous" message would be transmitted back to the primary synchronization host to acknowledge that the cooperation request on a P operation has been received.

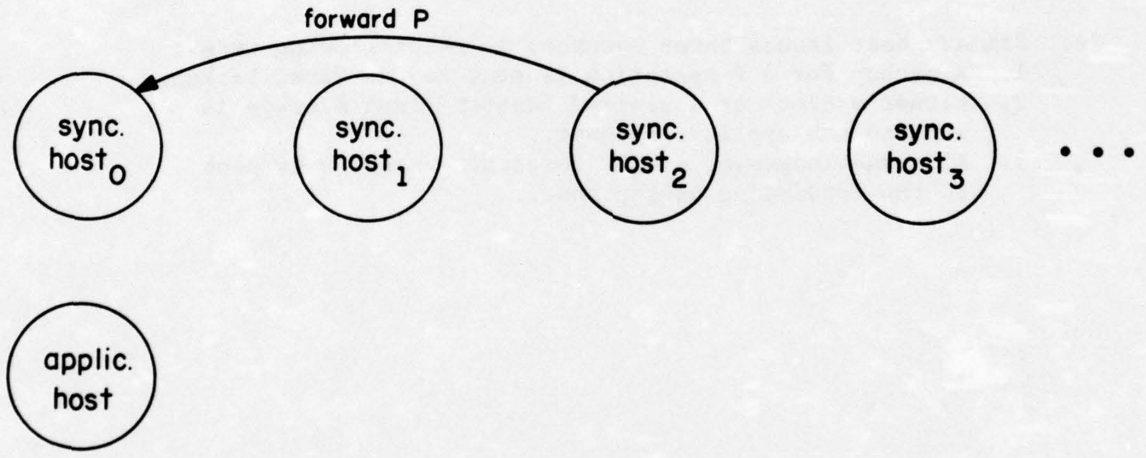
Once the primary synchronization host has received the cooperation acknowledgement it is certain that the two-host resiliency criterion has been met. Similarly, once the application host has received the block or proceed message it is also certain that the two-host resiliency criterion has been met. Should the primary synchronization host fail to receive the cooperation acknowledgement, appropriate retry and recovery techniques will be initiated. These techniques and the process of transmitting backup messages down the backup chain will be discussed in a later document.

Figure 2 shows the message flow for a P operation which has been transmitted to a backup synchronization host. The first network message delay is incurred in figure 2a. The application host transmits the P to a backup synchronization host.

The second network message delay is incurred in figure 2b. The backup synchronization host forwards the P operation to the primary synchronization host. The application hosts have no knowledge of the ordering of synchronization hosts for a given semaphore. However, each



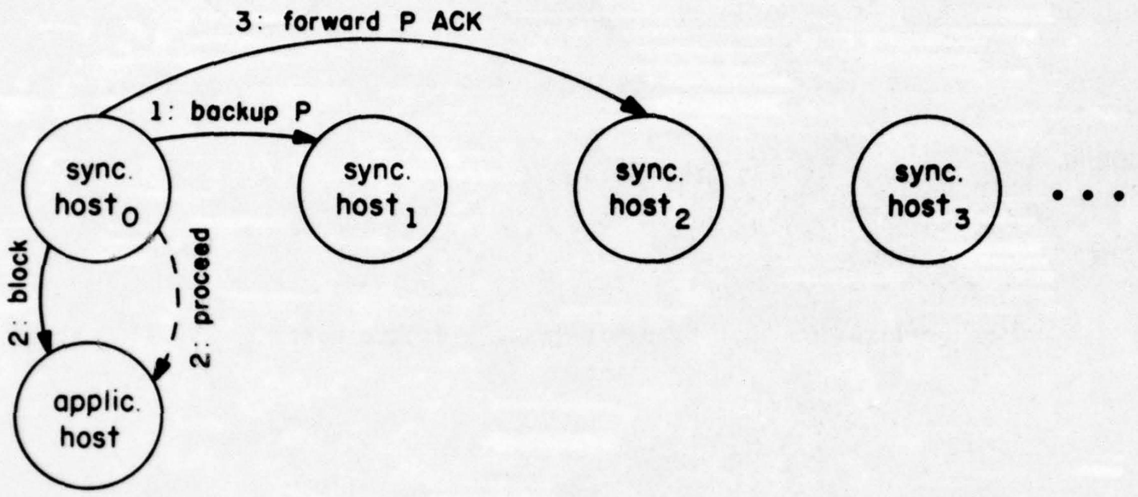
2a: Application host transmits P to a backup host.



2b: Backup host forwards P to the primary host.

Figure 2

P operation sent to a backup synchronization host



2c: Primary host issues three messages in the following order:
1. A backup for a P operation is sent to the first backup.
2. Either a block or a proceed (dotted line) message is sent to the application host.
3. An acknowledgement of the forwarding message is sent to the forwarding backup host.

Figure 2 (continued)

P operation sent to a backup synchronization host of the synchronization hosts is assumed to have explicit knowledge of the ordering. The backup synchronization host performs no updates on the semaphore. All updates must be initiated by the primary synchronization host. However, it now has knowledge of the existence of the P request from the application host. It will not discard this request until a backup message referring to that same P operation ripples down the backup chain and through it.

In figure 2c the third network message delay is incurred. Three messages are transmitted by the primary synchronization host. As was the case previously, these messages are essentially simultaneous but a specific ordering can provide some small improvements in resiliency. First, a backup message is sent to the first backup synchronization host. Second, a block or proceed message is transmitted to the application host. Third, a forward message acknowledgment is transmitted to the forwarding backup host.

Resiliency is achieved in this scheme by a combination of techniques. Basically, acknowledgements with time-outs and an "are you alive" protocol are used to detect host failures. In addition, each synchronizing host is notified that its request has been successfully forwarded to the next backup by its immediate neighbor. This guarantees the two-host failure requirement when propagating backup requests down the backup chain. Also, sequence numbers are assigned to requests and acknowledgements to allow detection of lost or duplicated messages. These techniques and variations on them will be discussed in a separate paper.

There are two properties of this scheme that should be noted. First, regardless of where the user process sends the P request, he will get a response in three message delay times. (If the synchronizing scheme is moved into the application hosts, this delay can be cut to two message times.) Second, two nearly simultaneous host failures are required to disrupt the scheme.

Summary. As we have mentioned, there are still problems with providing multi-site synchronization schemes. Lamport's solution as it stands is not resilient to failures and entails very large delay times. The Alsberg-Day scheme will probably be able to handle most of the resiliency problems. It also keeps the delay at the theoretical minimum. However, even with these minimal delays, the time required to synchronize will be on the order of a few hundred milliseconds. This compares unfavorably with the few hundred microsecond times required for synchronizing in a single-site environment. Given this thousand-fold increase in delay it is clear that frequent multi-site synchronization will not be practical for most applications. For example, synchronizing multi-user access to a heavily used, distributed data base will be much too time consuming. The delays incurred will cause a degradation in response that is intolerable to the user. For this reason, we have begun an investigation of techniques for synchronization avoidance.

Data Base Access Synchronization

Introduction

In the next two sections we consider a specific application of process synchronization, that of synchronizing the accesses to data bases. We begin with single-site data bases in the first section. The second section deals with distributed data bases. Throughout these sections the terminology of the relational model is used. However, the results are equally applicable to other data models. Where appropriate, terminology and examples from these other data models are included. In the interests of brevity, the phrase "data base synchronization" (or just "synchronization") will occasionally be used instead of the more correct "data base access synchronization." In all cases, it should be clear that it is the use of the data base which must be synchronized.

Single-site Data Base Access Synchronization

Introduction. The data base environment is characterized by two main features. First, the use of a large number of objects must be synchronized. Potentially, access to each domain value in a tuple (field in a record) must be synchronized. This is particularly true in non-relational data models in which the fields may contain pointers to other records. Unsynchronized updates of these pointers could render parts of the data base hopelessly garbled or inaccessible.

The second distinguishing characteristic of data base applications is that complex constructs must be synchronized. For example, in some data base systems there is no physical storage location that corresponds identically with a record. When the user retrieves a record, it is built for him. To synchronize access to the record, access to the components from which it is built must be synchronized.

In this section, the single-user illusion and transaction sequencing are discussed first. Synchronization entails guaranteeing that one user's transactions will not interfere with another's. Thus, each transaction must execute as if it were alone on the system. This is the single user illusion. Although a system may permit transactions to execute concurrently, their effect on the data base should be equivalent to executing them with no concurrency.

Next, the level of synchronization is discussed. For each application, data base access can be synchronized at a variety of levels. The choice of level ranges from a single domain value in a single tuple (a field in a record) to the entire data base. The level at which a data base is synchronized is the same as the unit of the data base that can be locked. If too large a unit is locked, then performance can be degraded because other processes may be unnecessarily blocked. If too small a unit is locked, then the integrity of the data base may be threatened if one writer acts upon the partially completed updates of another concurrent writer.

Finally, several techniques for implementing various levels of synchronization are discussed.

The single-user illusion. The goal of synchronizing data base access is to implement the single-user illusion. The single-user illusion implies that the final state of the data base will be identical to the final state reached if all the concurrently executing transactions had been applied sequentially. (Exactly which sequence is achieved is not important at the moment.) By maintaining the single-user illusion the system assures that the data base will always be in a well-defined state.

In some environments the single-user illusion may cause an increase in user contention. In extreme cases only one user may be active until his transaction is complete. Fortunately, this extreme case is likely to be rare.

In many cases there is no possibility of interaction between concurrent transactions. This occurs, for example, when all of a group of transactions are readers. Since they do not modify the data base, they do not interact with each other.

Non-interaction may also occur with writers if the data modified by each is disjoint. For example, suppose transaction 1 is updating the status of Army units in Korea, transaction 2 is retrieving the location of a particular Navy vessel, and transaction 3 is updating the status of Army units in Hawaii. All three of these transactions may be processed concurrently because the three transactions are accessing disjoint data. This principle has been directly employed as a synchronization mechanism by Eswaran et al. [1974], whose work is discussed later.

Transaction sequencing. In the discussion of the single-user illusion we were careful not to specify what the sequence of transactions should be. Different sequences may yield different final states. It is not important exactly which sequence is selected. What is important is that an unambiguous decision be made and adhered to.

Selecting a particular sequence may impact upon the possible concurrency. To illustrate this, suppose there are two transactions A and B. If it is decided that transaction A is to logically precede transaction B, then this decision must be obeyed for the entire duration of both transactions. This may seem fairly trivial, but consider what must happen if A is doing an update and B is doing a retrieval. For purposes of illustration, let an update consist of the following steps:

U1. Use the indices to find the tuples to be modified.

U2. Retrieve each tuple, modify it and re-write it.

U3. Adjust the indices to reflect the changes.

A retrieval consists of the following steps:

R1. Use the indices to find the tuples to be retrieved.

R2. Retrieve the tuples.

If there is any intersection between the tuples to be modified by A and the tuples to be read by B, then B cannot perform step R1 until after A has performed step U3. If this restriction were not followed, then B may attempt to utilize a partially invalidated index. By changing the values of the domains in the tuples, the old indices might point to tuples which no longer have the desired values and might not point to tuples which now have those values.

Suppose that the desired logical sequence is B, A; i.e. just the reverse of the previous example. Now both B and A may concurrently access the indices (steps U1 and R1). Transaction A must delay step U2 until after B has completed step R2. The point of this second example is that by a judicious choice of transaction sequence, it may be possible to obtain more concurrency.

It is difficult to determine whether or not two transactions may operate concurrently. (In fact, the general case is unsolvable.) The basic condition is that the final state of the data base is independent of the relative order of the transactions. In some systems it may also be difficult to force a particular, previously specified sequence to be obeyed. In these systems, concurrent transactions freely contend for the available resources. To force a particular transaction sequence it may be necessary to modify the system so that the outcome of this contention may be "fixed".

The a priori establishment of a transaction sequence is not required in systems where there is only one copy of each file. Any sequence choice will leave the data base in a consistent state. When there are multiple copies of files, each copy must choose the same sequence so that the results are identical at all copies. We will discuss this last point in detail later.

<u>Levels of synchronization</u>. The choice of a particular level of synchronization determines how and when process synchronization primitives are used. In many contemporary systems there really is no choice; the system provides only file level or only record level locking. In other systems, the application programmer, in effect, selects a level of synchronization in the design of his programs.

There are six levels identified here:

1. domain value level (field level),
2. tuple level (record level),
3. multi-tuple level (multi-record level),
4. relation level (file level),
5. cross-relation level (multi-file level), and
6. data base level.

Domain value level: The lowest level of data base synchronization is to synchronize the accesses to values of domains in a tuple (fields in a record). It is difficult to conceive of a data base system which did not provide this level. Consider the chaos which would result if concurrent users could see domains in intermediate states (i.e. if we did not provide domain value synchronization). To illustrate, suppose user A is changing a value from "green" to "brown". User B is a concurrent reader. Without domain value synchronization user B could retrieve a value of "breen" or "broen", both obviously meaningless.

Tuple level: This is a commonly provided level of synchronization. It is often used in inventory systems where updates usually affect individual records only.

Multi-tuple level: When updates must treat groups of tuples as single units, the multi-tuple level is appropriate. For example, suppose there is a doubly linked chain of records (that is, each record has a pointer to its predecessor and to its successor on the chain). To delete a single record, three records must be treated as a unit: the predecessor, the record being deleted and the successor.

Relation level: This is the other commonly provided level of synchronization. Logically, relation level synchronization is equivalent to multi-tuple synchronization where all tuples in the relation are locked. The major advantage of relation level synchronization is that it substantially reduces the overhead of maintaining the synchronization status of a possibly large number of individual tuples. At the same time, concurrency may be reduced because more tuples may be locked than are necessary.

Cross-relation level: In some applications tuples in several relations are logically connected, and must be treated as a unit. For example, consider an accounting system in which each account is a different file and it is required that all accounts must always be in balance. A normal double-entry bookkeeping transaction must be treated as indivisible. These transactions always affect records in two files (one for the debit and one for the credit). The cross-relation synchronization level is a synthesis of previous levels. In the bookkeeping example, record (tuple) level synchronization is used within each account file (relation).

Data base level: At the highest level, access to the entire data base can be locked. While this is simple to implement, it is clearly devastating to the achievement of any concurrency.

<u>Techniques for synchronization of data base access</u>. This section presents a brief summary of five techniques for data base access synchronization. They are:

1. critical sections,
2. tuple locks,
3. predicate locks,
4. passing rules, and
5. readers/writers.

For each, we discuss the basics of its mechanization and the levels of synchronization for which it is appropriate. The techniques are not independent of each other, but do represent significant conceptual differences.

Critical sections: A critical section is a piece of code that can only be executed by one process at a time. Critical sections are used throughout most multi-programmed systems to guarantee the integrity of tables, queues and other shared resources. Critical sections are implemented by programmatically issuing process synchronization primitives like P and V. Access to arbitrarily complex data units can be synchronized in this way. The primary advantage of critical section techniques is that they can be precisely tailored to the application and can lock the absolute minimum of resources. Thus concurrency can be maximized. The primary disadvantage is that they are implemented by convention in each program and can not be institutionalized in the data management system. Thus simple oversights or ignorance of other programs may cause synchronization failure.

Tuple locks: The most obvious way to implement tuple level and multi-tuple level synchronization is through the use of tuple locks. Conceptually, a semaphore is provided for each tuple. When exclusive access to one or more tuples is required, the appropriate locking primitives are executed on each semaphore. When the update is completed, the unlocking primitive is executed on the appropriate semaphores. (It is also easy to implement a multiple-reader/single-writer scheme using two semaphores per tuple.) If multiple tuples are involved, the locking of several resources is required and deadlock becomes a concern. The deadlock problem and its solutions are discussed in detail later.

Predicate locking: Predicate locking is a form of multi-tuple locking. It was proposed by Eswaran et al. [1974] as an alternative to explicit tuple locking. In predicate locking, the user specifies a predicate (i.e. a Boolean expression) that describes the tuples he wishes to lock based on the contents of the tuples. This predicate is processed against the predicates from other concurrent users. If there is no intersection, that is, no other user has locked any of the requested tuples, the request is granted. If there is an intersection, the new request must wait until the intersecting transaction completes. As should be clear, predicate locking is functionally equivalent to explicit tuple locking. Unfortunately, determining the intersection of two arbitrary predicates is, in the general case, an unsolvable problem. (By unsolvable we mean that there is no "program" which can determine whether the intersection of two arbitrary predicates is null.) Fortunately, if the predicates are suitably restricted, a "program" may be written to determine their intersection. Predicate locking is a new concept that has not been implemented except for limited demonstrations.

Passing rules: Passing rules are an even newer technique for multi-tuple synchronization. They work in the following way. First, a sequence of transactions is established. Then, the transactions are started through the relation at the "top." A no passing rule is enforced.

That is, no transaction may access a tuple farther down in the relation than the tuple being accessed by its predecessor in the transaction sequence. (Under tightly controlled circumstances, the no passing rule may be relaxed to allow readers to pass each other. This is analogous to the multiple-reader situations discussed elsewhere.) This appears to allow more concurrency than some other schemes because a large number of concurrent transactions may be supported. Each transaction may proceed as long as the next tuple needed has already been processed by the preceding transaction. Simple passing rules will work as long as the updates on one tuple can be performed independently of the results of the updates on other tuples. For example, a double entry bookkeeping problem applied on a single relation can be solved in this way. However, if the nature of the multi-tuple synchronization problem is that the update of a later tuple is dependent on the modified values of an earlier tuple, then later updates of the first tuple cannot be permitted until the second tuple is updated. In this case, simple passing rules are not adequate to solve the problem. Passing rules may also impose some possibly unacceptable constraints on the system designer. First, it is a practical necessity that only a single process can access the relation at any one time. Otherwise the enforcement of the no passing rule will require excessive inter-process communication. Second, all users must proceed through the relation in the same direction, e.g. top-to-bottom. This may not be the optimum way to process every query. However, mitigating this second point is the fact that the relational model gives the user an associative interface, so he is unaware of the actual tuple sequence. In at least one relational system [Schuster, 1976] all queries are processed top-to-bottom anyway. It is also not clear exactly how to handle indices when using passing rules.

Readers/writers: A readers/writers scheme separates the reader synchronization problems from update synchronization problems. Basically, two copies of the data are supported. All read requests are directed to one copy. Since readers cannot interfere with each other by altering the data base, there is no need to synchronize their activity. All write requests are directed to another copy. The writers' copy is managed using any of the synchronization techniques presented above. If writers are a small fraction of the total load, the demands on the writers' copy are small and contention is reduced. It may even be tolerable to let the entire system handle only one user at a time. Even this restrictive scheme will affect only the writers; it has no impact on the readers. Periodically the copies are switched. A bulk update is applied to the old readers' copy, bringing it up to date. During the bulk update, write requests must be stacked. Read requests can immediately begin to access the new readers' copy. One can view any system which batches its updates as a readers/writers scheme where the writers' copy is not actually maintained. A drawback of a readers/writers scheme is that the most recent updates are unavailable. (It is possible for a reader to become a writer to see recent updates, but this defeats the purpose of the scheme.) Another consideration is that additional storage will be required to hold the second copy. Finally, the switch-over and batch update imposes an added workload that is not present with the other schemes. The strongest argument for readers/writers is its simplicity and its ability to support many concurrent readers.

Summary. This section has discussed the concepts of the single-user illusion and of levels of synchronization. For some applications complex levels of synchronization are required. For many others, in which updates affect only individual and independent tuples, tuple level is sufficient. The major techniques for data base synchronization were discussed. In a practical system it is likely that combinations of these techniques would be used.

Distributed Data Base Access Synchronization

Introduction. In this section we present a discussion of access synchronization in a distributed environment. First, we define what we mean by a distributed data base. Then we outline the most important considerations in the selection of a distributed data base synchronization mechanism. Finally, we consider in some detail the problem of maintaining consistency among multiple copies by sequencing the transactions in a well-defined order.

Since "distributed data base" has become a catch-phrase, we will define precisely what we mean. This work is oriented toward computer networks similar to the ARPANET. The kind of distributed data bases considered here have the following four distinguishing properties:

1. geographic separation of the participating hosts,
2. limited host-to-host communication bandwidth,
3. co-equal roles for the hosts (i.e., not a master-slave relation), and
4. multiple copies of portions of the data base.

Let us briefly discuss the implications of these four properties.

The geographic separation of hosts creates irreducible time delays resulting from the propagation of signals over long distances. For example, in the ARPANET the transatlantic link and the Hawaiian link are implemented using synchronously orbiting satellites. The great distances involved with these satellites induce a propagation delay of 1/2 second on a Europe to Hawaii transmission. Other delays on the order of 10-200 ms are introduced due to the nature of the store-and-forward communication subnet.

The second distinguishing property is that the host-to-host communication bandwidth is a small fraction of the bandwidths to typical secondary storage devices. For example, current disk systems (IBM 3330, HIS DSS 190, etc.) can achieve transfer rates of approximately 800,000 bytes/sec (6.4 Megabaud). In three days' routine ARPANET file transfer traffic between two ARPANET hosts, we observed an average bandwidth of about 4.3 Kilobaud. (*Naturally, the reader should not infer much from* this very restricted example; it is merely intended to show the disparity in bandwidth between local secondary storage accesses and network accesses.)

One important implication of the geographic distribution of hosts and the limited bandwidth between hosts is that new operational structures for distributed data bases must be devised. In particular, naive extensions of conventional techniques for synchronization, such as locks, may lead to intolerably poor performance in a distributed environment.

The third important characteristic of the distributed data base systems discussed here is the equal status of the hosts. This characteristic is the result of a design decision. It is not an unavoidable circumstance like the first two characteristics. By choosing to give the hosts equal status, the resiliency of the overall distributed data management system is substantially improved. There is no longer a critical host on whose failure-free operation the entire system depends. The failure of any participating host may cause aberrations, but not disaster. In current systems when a host crashes, all the users at that host also "crash". In network systems, these users can be connected not to an individual host but to the network itself via front-end machines (mini-hosts). When a particular host crashes, these users can still proceed by shifting their work to the surviving hosts. This type of co-equal structure means that new research areas are opened in designing systems which operate effectively in such an environment. Specifically, new types of synchronization techniques must be devised.

The last characteristic is that we allow the existence of multiple copies of the data. Again, this is a design decision. Multiple copies of critical files greatly increase the availability of those files, as was shown in Belford et al. [1975]. Further, there is no inherent requirement that all the copies have the same structure. Each copy may have a structure which is best-suited to a particular group of applications. The only requirement is that the information contained in each copy be the same. This allows applications to be moved to other copies. For example, suppose that we have an employee file. One copy might be sorted on last names, and another sorted on employee numbers. Another difference might be in the indices for the various copies. Even two identical files may have different indices associated with them. A major problem, then, is to keep the various copies and indices consistent with each other. A query should get the same answers regardless of which copy is actually used.

The implication of the co-equal status of the hosts and the existence of multiple copies is that distributed control strategies must be used. Considerable attention must be devoted to failure recovery. These topics form the substance of a forthcoming technical report, and will be touched on only lightly here.

Evaluating synchronization techniques for distributed data base access. In the previous section we discussed the basic techniques for single-site data access synchronization. In extending synchronization to a distributed environment, complications can arise. A user may need to coordinate access to several sites. Consistency among multiple copies of the data base must be maintained.

A factor which must be kept constantly in mind is the host-to-host message delay. Frequent synchronization among different hosts is infeasible because of this delay. In short, the number of synchronizing messages which must be transmitted across the network should be reduced to a minimum. The data base managers at the various sites should have the burden of seeing to it that the access requests that they receive are handled in an orderly manner. Any single-site synchronization scheme is adequate for this purpose, although, as discussed above, certain tradeoffs should be taken into account.

If activity among several hosts must be coordinated, some minimum number of interhost messages is unavoidable. This minimum number will be affected both by the detailed scheme adopted and by the need for resiliency. We are presently in the process of addressing these issues. In particular, we are investigating how the Alsberg-Day scheme for resilient process synchronization may be applied to data base access. A report on this scheme is forthcoming. A lengthy discussion of these issues would be premature at this time.

Maintaining consistency among multiple copies. In order to maintain consistency among multiple copies of data, it is important to ensure that the updates are applied to each copy in the same order. For example, if one update adds 10 to a field and a second increases the same field by 10 percent, interchanging the order of the operations will change the final result. There are essentially two different ways to maintain the update sequence. One way is by seeing to it that all sites receive the updates in the same order. The second is by explicitly attaching a sequence number to each update.

The first approach can be handled by a scheme very similar in spirit to the Alsberg-Day resiliency scheme. An ordering of the sites holding a copy is established. Each update is first sent to the host designated as the primary. The primary then sends the update to the next host, and so forth. The precise sequence of the updates is therefore determined by the order in which they arrive at the primary. Some safeguards must be taken to see that updates from the same user will arrive in order (if that order is important) and that the primary (or some later host) does not somehow mix up the ordering. Such resiliency problems require further study. It may also be that certain complicated data operations involving multiple sites are not readily handled by this simple scheme. This question is another topic of current investigation.

The second approach is to attach explicit sequence numbers to the transactions. If each update in the network, no matter where it was generated, has a unique sequence number attached, then every data base manager can use these numbers to order the operations. The question that then arises is how the sequence numbers should be assigned. Three different techniques are presented here. These are

1. centralized assignment,
2. partially distributed assignment, and
3. completely distributed assignment.

Centralized assignment: The most obvious way to assign sequence numbers is to have a distinguished host (the <u>primary</u>) from whom sequence numbers are requested. In this scheme, the host generating the update sends it to the primary, where a sequence number is attached. At this point, two variations are possible. (1) The update, with the assigned number, may be returned to the originating host for transmission to the various data base copies. This scheme envisions the number-assigner as a very simple piece of software which does nothing but hand out numbers on request. (2) The primary may have the responsibility of broadcasting the update (with its sequence number) to all the copies. This scheme is functionally equivalent to the approach (discussed above) which orders the sites. Adding sequence numbers can be thought of as one way of providing some of the needed resiliency; i.e., of ensuring that none of the sites destroys the order of the transactions. A centralized assignment scheme with updates broadcast by the primary has been discussed in some detail by Bunch [1975].

Centralized sequence number assignment has several defects. First, the primary can be a bottleneck, because it must assign a sequence number to each transaction. Second, each transaction can incur delay because it must wait for the sequence number to be assigned. However, the bottleneck problem need not be severe. For example, different sites could be designated as the primary for different parts of the data base. And if the primary holds a copy of the relevant data, it would have to receive and process the update eventually anyway. The overhead associated with the actual number assignment should be minimal. The transaction delay should also be negligible (a few hundred milliseconds). However, the host generating the update might find it inconvenient to wait for a number to be returned from a remote site before applying the update.

Partially distributed assignment: A more distributed scheme that we are currently studying is the "reservation center" scheme of Grapa [1975, 1976]. We will here use the more descriptive term "partially distributed sequence number assignment". The essence of the scheme is that sequence number assignment is a two phase process. First, the reservation center, a distinguished host, gives each host a block of sequence numbers. These numbers are valid only during a limited time interval. All numbers issued are ordered. When a host wishes to initiate a transaction, it takes the next unused sequence number from its block of numbers for the current time interval. Some strategy must be devised to handle the case when more numbers are requested than are available. However, this is not essential to our purposes here. Some of the problems of the central number assigner are ameliorated by using the reservation center. For example, when the number assigner fails and this task is transferred to a new host, there may be some difficulty in determining the "next" number to be assigned. There is no such problem if the reservation center fails. Instead, sequence numbers from the next time interval are assigned. They will be greater than any previously sent numbers. The reservation center is also less likely to be a bottleneck because its response is not critical for the continuation of an update, as was the case with a primary copy. The workload of the reservation center is more periodic and predictable. Every time interval it must make up and transmit new blocks of sequence numbers. There is a minor problem in determining exactly which sequence numbers have been used. This is necessary to detect a failure in which updates have been lost.

Completely distributed assignment: The reservation center's only critical purpose is to ensure that there are no duplicate sequence numbers. This can also be achieved by other means. If we let each host generate its own sequence numbers, we have a completely distributed scheme. One such scheme has been described by Johnson and Beeler [1973] and by Johnson and Thomas [1975]. Their scheme was more concerned about maintaining the temporal sequence of updates; i.e., maintaining the precise order in which the updates are generated by the users. They tried to achieve this by generating the sequence numbers partly from the local clock time. The required uniqueness of the sequence numbers may be achieved by appending host id, user id, etc. There is an interval of uncertainty because the clocks on the various hosts may not agree. One would expect this interval to be small in the WIN environment because the military already requires good time coordination. For other environments, there are techniques for detecting and correcting an inaccurate clock [Grapa, 1976]. However, there is a similar interval of uncertainty in the other schemes considered. In a centralized scheme, network delays may cause updates from different sites to be ordered quite differently than their precise time of generation would indicate. The reservation center approach was in large part developed to formalize the viewpoint that the precise sequence of updates within a certain time interval doesn't matter. It appears that overconcern with applying updates in their "real" order is something of a red herring. Even for a single-site time-sharing system, the order of transactions from different users may sometimes be determined as a random outcome of the terminal polling process.

In determining which of the sequencing schemes described above is best used in any particular distributed environment, careful consideration must be given to the operations that can be performed on the data. For example, Johnson and his co-workers assume that only assignments are allowed. This solves some problems. Superceded assignments may be simply thrown away. On the other hand, if only increments and decrements are allowed (as for an inventory system) the order in which these operations are performed doesn't matter, and no sequencing scheme at all is needed.

In a realistic, multi-operation system, assignments, increments, decrements, etc., will all be allowed. If an update arrives at a site out of order (as determined by explicit sequence number), it may cause earlier arriving updates to have to be redone. That is, the system must have some provision for undoing and redoing operations. As an alternative, the system could wait for "missing" sequence numbers - an approach that is feasible only if all numbers are assigned centrally and all sites receive all updates. Once one begins to consider problems of this kind, the implicit sequencing (by site ordering) discussed at the beginning of this section looks very attractive.

If operations can be data dependent and modify multiple records (e.g. increase the price of all blue uniforms by 5 percent), then the analysis of the problems and of the costs of the various schemes becomes very complex. We are presently in the midst of an intensive study of these and related concerns. A detailed report will be produced in the future.

Summary

In this chapter we have looked at the problem of data base access synchronization. It should be clear that the state of the art will not permit a clear-cut favorite technique to be selected. Rather, each technique proposed has strengths and defects. Nowhere is this more true than when distributed data base access synchronization is considered. The basic characteristics of that environment are only now beginning to emerge. While broad requirements may be formulated, advocacy of any technique is premature.

The Deadlock Problem

Introduction

Deadlock can occur when processes which concurrently compete for resources have the power to lock those resources against access by the other processes. Consider the following simple example. Suppose process P_1 has locked resource R_1 and requires the use of R_2 before unlocking R_1. But suppose that process P_2 has previously locked R_2 and will not release it until after it has used R_1. Then P_1 and P_2 are in a state of deadlock. Each is unable to proceed until it can obtain a resource locked by the other. This situation is easily pictured by a bar graph (see figure 3). The bars indicate the periods of time during which each process would have the resources locked if it were the only process in the system. In figure 3, process P_1 becomes blocked at time T_1 and the two processes become deadlocked at time T_2.

A more complicated deadlock situation is pictured in figure 4. At time T_4, P_1 requests R_2 and is blocked from proceeding since P_2 already has R_2 locked. (This point past which P_1 cannot proceed is indicated by an arrow in the figure.) At T_5, P_2 becomes blocked by its need for R_4, which has been locked by P_3. Finally, at T_6, P_3 also becomes blocked, so that it never releases R_4 to P_2, which in turn never releases R_2 so that P_1 can proceed. The three processes are now deadlocked. Notice that deadlock will occur in this example even if, after the first resources are locked, the second (and later) resources needed by each process are not requested to be locked but only accessed (e.g., a read operation to a file). Thus, even if many operations do not require locks, the potential for deadlock may be substantial.

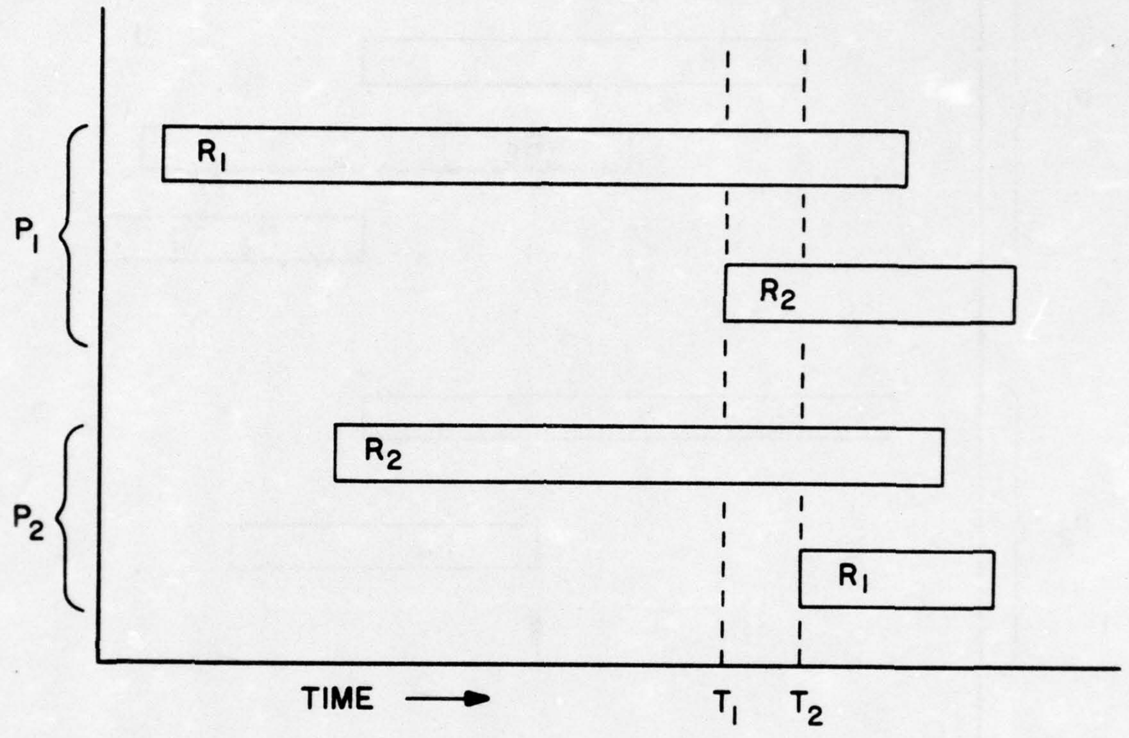


FIGURE 3: SIMPLE DEADLOCK

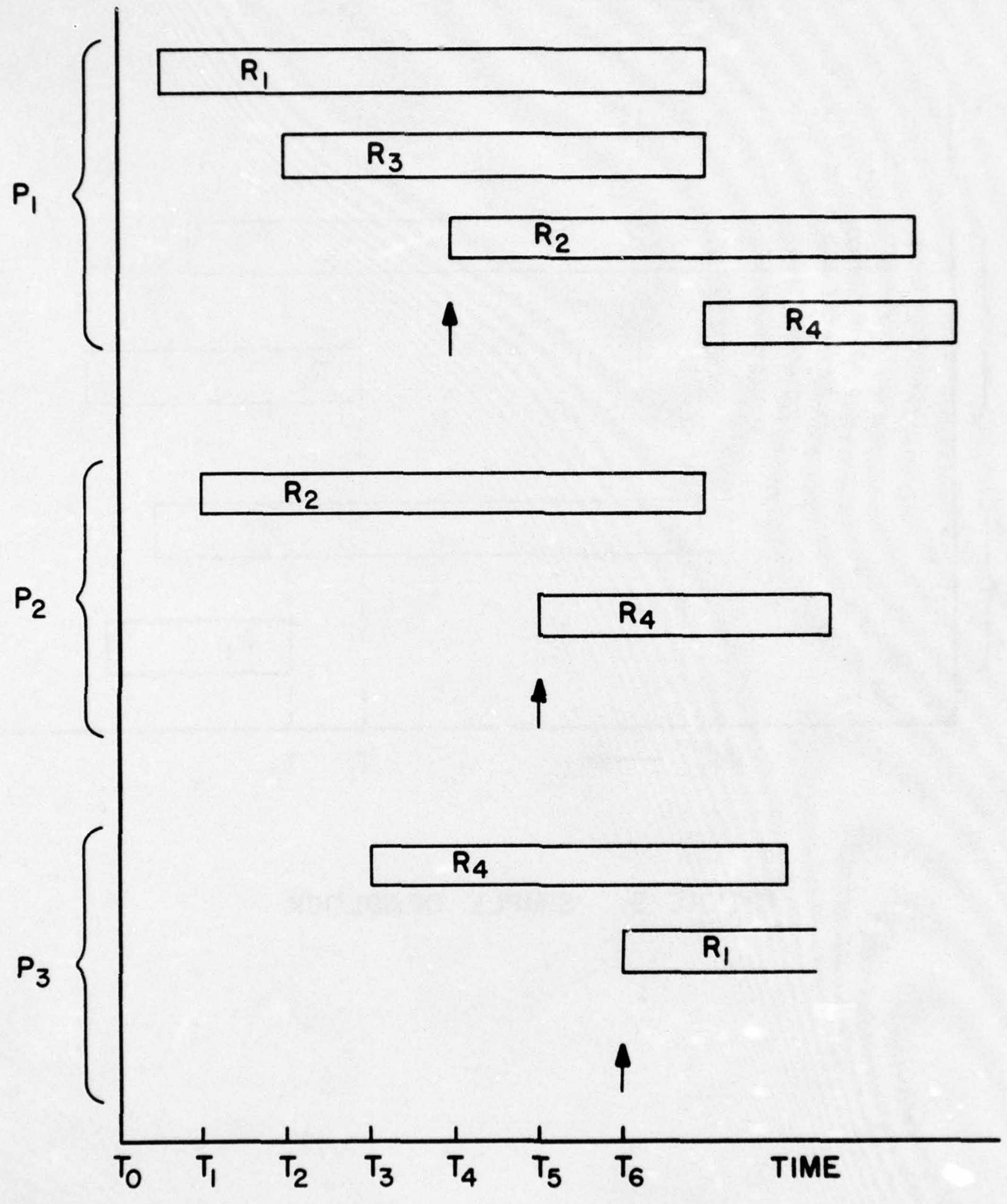


FIGURE 4: COMPLEX DEADLOCK

The situation pictured in figure 4 may be generalized to ever more complicated interactions in which large numbers of processes impinge on each others' needs in a way which ultimately causes deadlock. A helpful alternative method of picturing processes and their resource needs in complicated situations has been introduced by King and Collmeyer [1973]. This alternative method uses what are called access state graphs. These are snapshots, at successive instants of time, of processes and the resources allocated to (and locked by) them. Figure 5 shows a set of such access state graphs which give essentially the same information as figure 4. At T_0, the graph consists only of nodes for the three processes in the system. As each process requests and locks resources, arcs are drawn from the process node to the first resource node, and then to successive resources. Thus, at T_1, P_1 has locked R_1 and P_2 has locked R_2. (Ordinarily these locks would be represented by two separate graphical steps.) At T_4, the dashed line from R_3 to R_2 indicates that P_1 is blocked from access to R_2. At T_5, P_2 becomes blocked attempting access to R_4. And finally P_3 is blocked in accessing R_1. Notice that the T_6 graph contains a cycle, consisting of the arcs from R_1 to R_3 to R_2 to R_4 to R_1. Indeed, King and Collmeyer show that, if the system at time T_{j-1} is not deadlocked, then the necessary and sufficient conditions for the access state graph at T_j to represent a deadlock are:

1) the existence of a cycle, and
2) the fact that the last arc added is dashed (i.e., represents a blocked access).

When resources are released or processes are completed, the corresponding nodes and arcs are deleted from the graph. In this way, access state graphs can be used by a system to keep a dynamic record of processes and associated resources. Such a record is clearly of potential usefulness

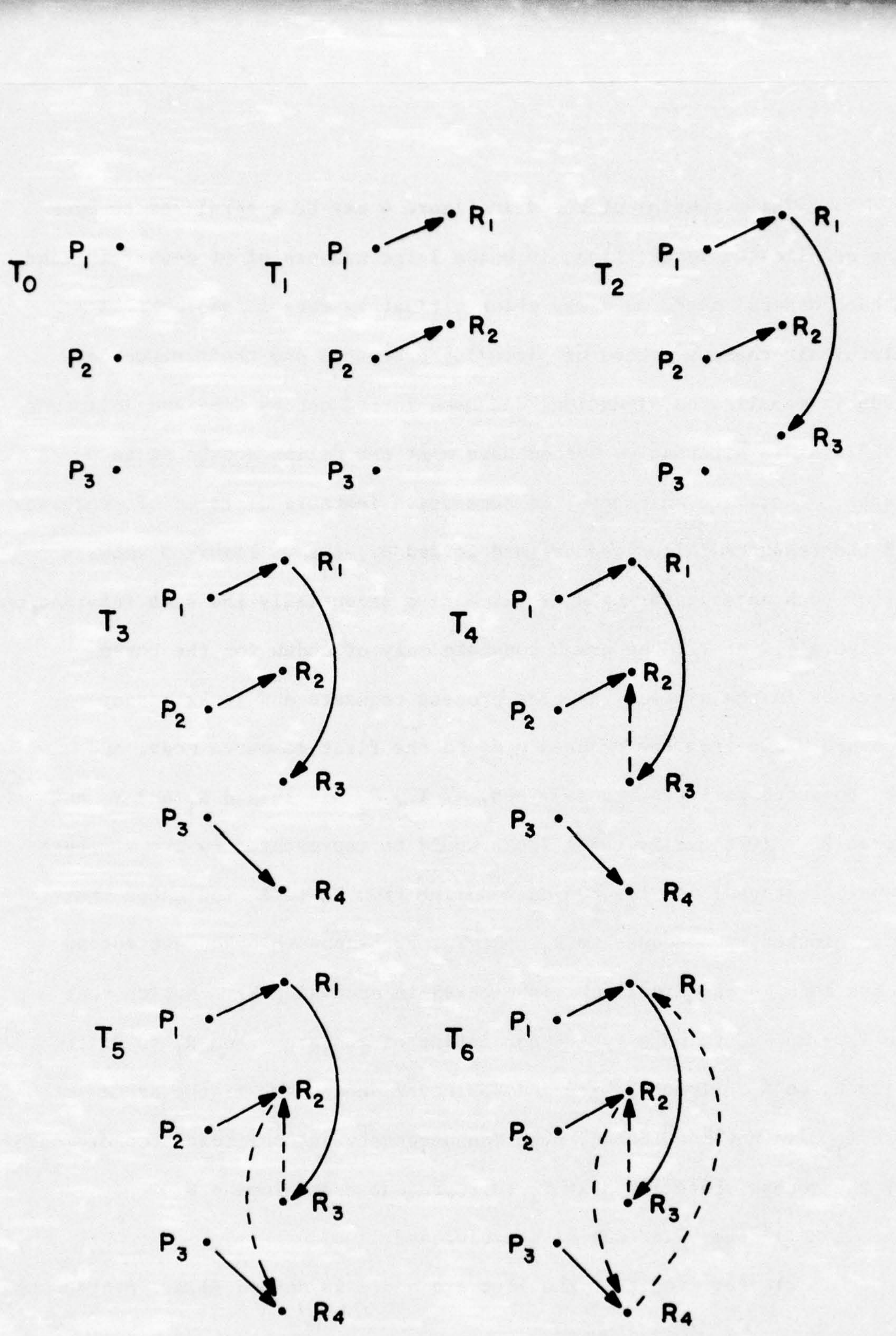


FIGURE 5: ACCESS STATE GRAPHS CORRESPONDING TO FIGURE 4

in identifying a deadlocked state so that appropriate action may be taken. But we leave the question of treating the deadlock problem to the next section.

The reader should by now have gained some intuitive insight into how deadlock may occur. More than intuition is needed, however. Coffman et al. [1971] have helped to formalize the study of deadlock by setting down four necessary conditions which must hold before deadlock can occur in a system running processes concurrently. These conditions are:

1) Mutual exclusion condition. (Processes can lock resources for their exclusive use.)
2) Wait-for condition. (Processes may hold some resources while waiting for additional ones.)
3) No-preemption condition. (No process can be forcibly required to release a resource it holds.)
4) Circular-wait condition. (There exists a circular chain of processes, each one holding resources needed by the next.)

In an interactive data management system many independent concurrent users may be accessing and updating the data. Locks are a virtual necessity to avoid severe loss of integrity and deadlock is a problem which must be addressed. In a network environment the problems can be accentuated by, for example, the absence of a central control.

In the remainder of this paper, we consider the various approaches to handling the deadlock problem. Although terminology in the literature tends to vary, these approaches can be divided into three basic types:

1) Detection and Recovery. This involves monitoring the processes and their resources to identify a state of deadlock - e.g., by the use of access state graphs - and then backing out or aborting one or more processes.

2) Avoidance. In this approach, even more complex graphs are generated and studied to determine a safe strategy for resource allocation before deadlock becomes inevitable.

3) Prevention. This involves setting up the system in such a way that one of the four necessary conditions of Coffman et al. does not hold.

In the next section, we will describe these approaches in more detail - particularly those which seem to be most promising for use in distributed data management. Following this general description of techniques for handling deadlock, we provide an assessment of their usefulness in the context of large, distributed data bases. A key problem seems to be that most of the techniques described in the literature require some sort of central monitoring and/or control. We therefore present and discuss in some detail a new scheme for implementing deadlock detection in a network without centralized control.

Treatment of Deadlock

Detection and recovery. The best comprehensive discussion of detection and recovery techniques - at least in the context of data management - is contained in the paper by King and Collmeyer [1973]. We have already described their access state graphs and their main theorem, which may be applied to detecting deadlocks. Detection then becomes, at least in theory, a rather straightforward procedure.

Other work on the deadlock detection problem has been based on much the same principles - i.e., the construction (at least conceptually) of a graph linking processes and resources and the examination of this graph for configurations (equivalent to King and Collmeyer's cycles) which indicate deadlock. Perhaps the earliest paper describing such techniques was that of Murphy [1968]. Although Murphy primarily discusses matrix techniques, they are equivalent to graphical ones. Indeed, the obvious way to represent graphs in a computer is as a matrix where rows and columns correspond to nodes and non-null entries represent linkages between node pairs.

A rather elaborate graphical approach was published by Holt [1972]. Holt's analysis is complicated by distinctions between _consumable resources_ (e.g., external interrupts) and _reusable resources_ (e.g., I/O devices), and by the possibility of multiple units of each. In this setting, a simple cycle in a graph is still necessary, but no longer sufficient, for deadlock. Holt's algorithms for deadlock detection therefore look far more complicated than those of Murphy or of King and Collmeyer, but Holt claims that they are practical to implement (with execution times proportional to the number of resources times the number of processes).

Recovery is a far more difficult problem than detection. Holt dismisses the problem with the statement that "if deadlock has occurred, it will be necessary to terminate jobs or to pre-empt resources from jobs." Both of these solutions constitute "recovery" from deadlock only in the sense that the resources involved in the deadlock are released so that the operating system may proceed normally. In many cases one also wants "recovery" to include the preservation of the states of the aborted processes so that they may be conveniently restarted and/or the return of all resources (e.g. data which may have been left inconsistent) to a usable state.

Murphy suggests the possibility of requiring a process to release resources and re-request them later. This is basically a pre-emption solution. Although there is a hint here that the backing out of the process may be orderly (hence making process recovery possible), no details are considered.

King and Collmeyer, however, discuss recovery in some detail. It may be worthwhile to quote their definition here before discussing their approach. "Recovery is the procedure by which the effects of a (necessarily) aborted process on the object data base are reversed so that the process can be restarted." Their procedure has two main features:

1) A checkpoint (the recording of the state of the process) is performed whenever a process begins to lock resources. Thus, if the process must subsequently release its resources, it may be returned to its previous state.

2) A process map (a device specific to deadlock in data base management) essentially allows changes in the data to be made in a copy of the locked portion of the data base. The changes become visible to other processes only after an unlock is performed. Thus if the process is aborted, the data changes vanish.

By using both checkpointing and a process map, King and Collmeyer allow for full recovery of both the processes and the resources (data). A price must be paid for this full recovery, however, since maintaining checkpoints and process maps requires considerable CPU time and storage space.

Another problem related to recovery is deciding which process to abort. That is, if two processes are seen to be deadlocked, the system must decide which should be allowed to continue. There is a small amount of discussion in the literature on strategies for making this decision. Clearly the basic principles should be maximization of ease of recovery (in some sense) and/or minimization of wasted effort.

Avoidance. The term *avoidance* does not always have the same meaning in the deadlock literature. We are using it here in a technical sense. Avoidance techniques handle deadlock by an elaborate scheme for monitoring processes and the resources they request. Resources are allocated only after a careful analysis of that allocation's potential for causing deadlock. The techniques are very similar to those used for deadlock detection. The only difference is that, instead of using access state graphs (or an equivalent scheme) to determine when a deadlock has taken place, the system subjects the graphs to an elaborate analysis whenever a lock is requested. If there is any possibility that satisfying the request may lead to deadlock, the request is (temporarily) denied.

The earliest scheme of this type to be proposed was the "banker's algorithm" of Dijkstra [1968], which first appeared in 1965. The idea behind this algorithm is that an allocation can be safely made as long as there are enough resources left that, by allocating needed resources to each process in turn, the system can run all processes to completion. Thus the system must have a list of the maximum possible needs of all processes. Before each allocation the system checks to see that all processes can be guaranteed to be able to finish.

A formal study of this type of technique is contained in the classic paper of Habermann [1969]. Essentially, Habermann defines the *allocation state of a process* as a specification of

a) resources currently allocated to the process, and

b) resources *claimed* (i.e., which may be ultimately used) by the process.

The set of the allocation states of all processes, plus information on available resources, make up the overall allocation state which must be examined by the system. Certain allocation states are safe in that "starting from a safe state there is at least one way to allocate the claimed resources to each process P_k even ... when each P_k asks for all the resources it has claimed and does not release any resources until it has been allocated all its claimed resources." Thus the system may avoid deadlock by allocating resources in such a way that successive allocation states are safe. Based on several theorems relating to safety, algorithms for identifying safe states are given.

Holt's graphical approach to keeping track of processes and resources also provides a mechanism for deadlock avoidance. In fact, Holt claims to improve somewhat on Habermann's algorithms.

Two features of these deadlock avoidance schemes should be emphasized.

1) Processes must claim resources in advance.

2) The system must carry out a complicated analysis, which involves examining all processes and their claims, before every lock.

Prevention. We are using the term prevention to denote techniques which guarantee that one of the four necessary conditions for deadlock do not hold. To repeat, these conditions are:

1) Mutual exclusion condition
2) Wait-for condition.
3) No-preemption condition.
4) Circular-wait condition.

Papers by Coffman et al. [1971] and by Havender [1968] discuss the system constraints which can be used to eliminate each of these conditions. Elimination of condition 1 simply means allowing processes to interfere with one another - an unacceptable solution. Elimination of condition 3 involves allowing the system to preempt resources from processes using them. As we indicated above in the discussion of detection and recovery, the forcible removal of resources from a process opens up the question of whether the process (and/or the resource) is left in an undesirable state.

The way to eliminate condition 2 is simply never to allow a process to lock some resources and then request more. Practical approaches to imposing this constraint are of two types.

(a) A process must preclaim any set of resources which are needed concurrently, and the system does not lock any of them until all are available for allocation to the process.

(b) A process is required to release all currently locked resources before making new requests. Notice that this scheme differs only in viewpoint from (a). Resources actually needed concurrently must still be requested and allocated as a group.

Finally, condition 4 may be avoided by imposing a linear ordering on the resources and requiring that resources only be requested in that order. Consider, for example, figure 3. Suppose the resources are ordered R_1, R_2. Notice that deadlock would not occur if P_2 were required to request resources in the order R_1, R_2, for then P_2 would simply be blocked until after P_1 released R_1. This solution has many attractions and will be discussed further below.

Ignoring the problem. One approach to handling the deadlock problem is simply to ignore it. Specifically, one can assume that deadlock occurs so infrequently that it may be handled like any of many other unforeseen hardware or software failures - i.e., the offending processes are aborted with no attempt at recovery. This approach does not require any elaborate detection mechanism. Processes may simply be aborted for having been in the system "too long" - for whatever reason. Or resources held by a single process "too long" may be preempted and the process aborted.

One should be reluctant to adopt this solution unless there is some rationale for assuming that deadlock will not be important. In an interesting simulation study, Shemer and Collmeyer [1972] attempted to assess deadlock frequency in the context of a shared data base. Because of the difficulty of identifying true deadlocks, they define a simulated deadlock as occurring when a process attempts to lock a resource already held by a blocked process. In the situations pictured in figures 3 and 4, the simulated deadlock is a real deadlock. This is not always the case. Suppose, for example, that, in figure 4, P_3 releases R_4 at T_5. Then P_2 is allowed to continue and will ultimately release R_2 and unblock P_1. No deadlock occurs. But a simulated deadlock still occurs at T_6. The frequencies computed by Shemer and Collmeyer are therefore overestimates of true deadlock frequency.

The parameters in their model are (with small changes in terminology)

a = number of lockable blocks in the data base,

b = percent of accesses that are updates,

c = mean number of blocks hit per access, and

N = number of users.

To quote a typical result, they find that for a = 300, b = 70%, c = 6, and N = 20, the average number of deadlocks in 10,000 accesses is about 35. They display their results graphically; selected curves from this graph are reproduced here in figure 6. Some interesting features are

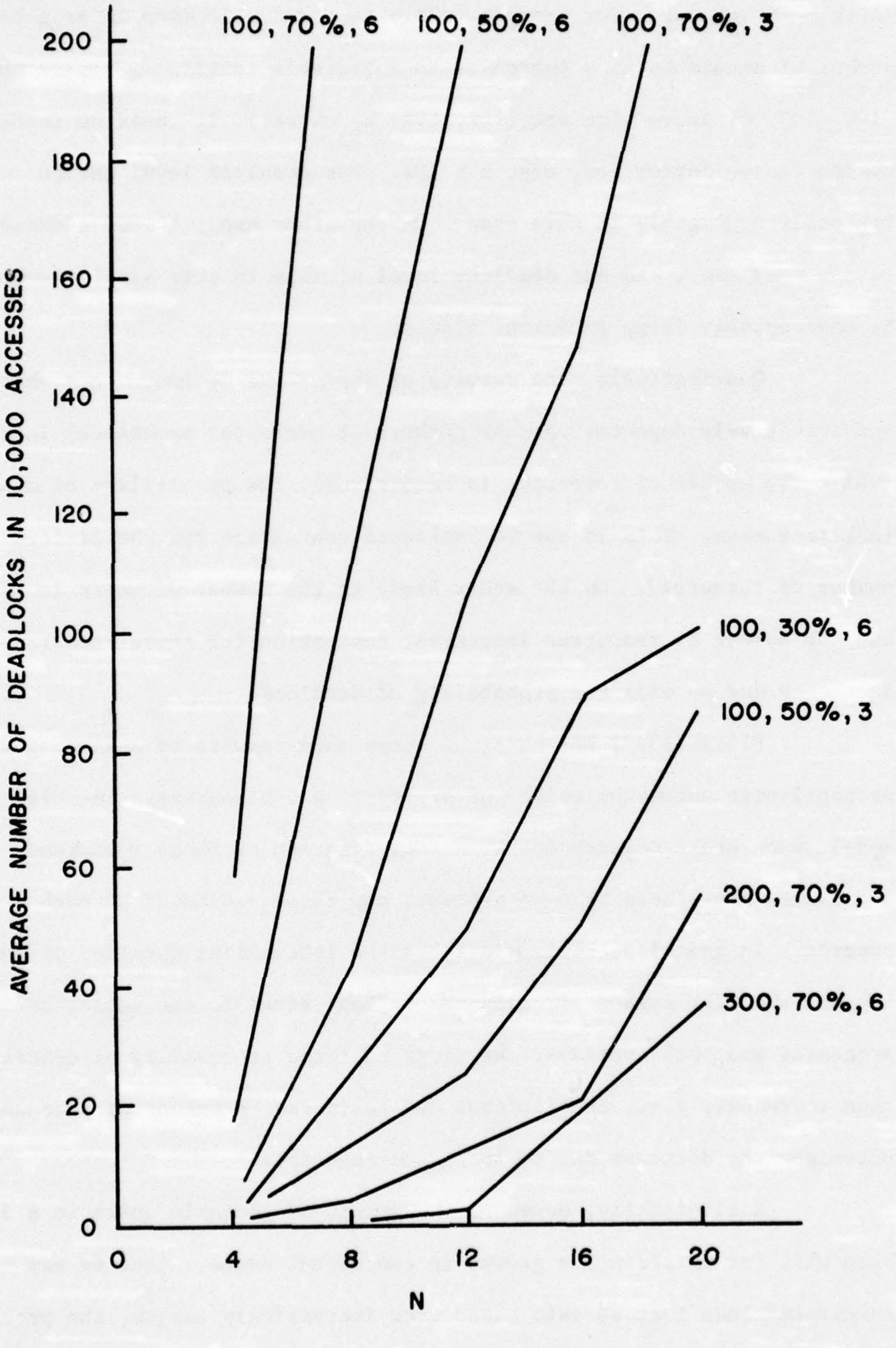


Figure 6

Selected results from the deadlock simulation study of Shemer and Collmeyer [1972]. Curves are labeled with values of a, b, c.

worth pointing out. For example, there is a dramatic drop in expected number of deadlocks as b decreases to reasonable levels. (Compare the (100, 50%, 6) curve with the (100, 30%, 6) curve.) It would be interesting to see curves for, say, b = 10%. The deadlock level may be virtually negligible in this case. On the other hand, 10,000 accesses is not very many, and any deadlock level visible in this simulation may be unacceptably large in a real system.

Qualitatively, the results of Shemer and Collmeyer are what one intuitively expects. As the numbers of users (or processes) increases (while the number of resources is held fixed), the probability of deadlock increases. This is due to increased contention for the limited number of resources. On the other hand, if the number of users is fixed and the number of resources increases, contention for those resources decreases and so will the probability of deadlock.

Ellis [1973] has obtained these same results by analyzing a probabilistic automaton model for processes and resources. In this model, each state corresponds to a specification of those resources available, those held by each process, and those requested by each process. In addition, Ellis addresses the interesting question of what happens when the number of resources is kept equal to the number of processes and both increase. He finds that the probability of deadlock then increases; i.e., the increase due to increased number of processes outweighs the decrease due to increased resources.

Realistically, however, the growth of lockable units in a data base will far outstrip the growth in concurrent usage. Thus we may safely conclude that as data bases grow increasingly larger, the probability of deadlock in data access will decrease.

It is worth noting that Shemer and Collmeyer also keep track of the number of occurrences of interference (retrieval blocked by an update) and roadblock (update blocked by a user who is not blocked). These events may also seriously degrade performance, but they do not cause the potential loss of integrity that the resolution of a deadlock does. For example, they find that for a=100, b=30%, c=6, and N=20, the average number of users on the wake-up list (i.e., that have encountered roadblock and are waiting for access) is seven. With the same parameters, the probability that a retrieval will encounter interference is about 0.45. Again, these are upper bounds and, provided that the blockages are not long-lived, they do not indicate serious performance degradation. The reader should consult Shemer and Collmeyer's paper if he is interested in seeing more results along these lines. We restrict ourselves here to quoting from their conclusions.

"What is important to note is that ... the performance of a shared database system is still quite satisfactory unless the operating situation is atypical [high percent of updates]. When the size of the database becomes reasonably large, the number of deadlocks and interferences decrease more than proportionately. In the simulation when there were more than 300 groups [lockable units] in the database, there were virtually no occurrences of deadlocks and the number of interferences was also very small... ."

Combinations of techniques. In many real systems, a combination of techniques is used. One technique may seem most suitable for certain resources, while other resources are best handled in some different way.

Havender [1968] has described the approach taken in the design of the SYSTEM/360 Operating System job initiator. In dealing with data sets, the approach is to require a process to preclaim all needed data sets. The process is then not initiated until all preclaimed sets are available to be allocated to that process. This is a prevention technique - the elimination of the wait-for condition. In allocating devices, the same basic approach is used - elimination of wait-for. But instead of requiring that all needed resources must be pre-claimed and pre-allocated, the process is simply required to release the presently held group of resources before requesting more. Main storage is also allocated on essentially this basis. There are therefore three types of resources - main storage, other devices, and data sets - and within each type a deadlock prevention mechanism is defined. But something more is needed - a way to avoid conflict between allocation of these types. The approach taken was to define an order in which the three types are allocated, with the goal of eliminating the circular-wait condition. The system, however, does not have the complete information (i.e., the future plans of all processes) needed to guarantee a correct allocation, and a certain amount of deadlock will occur and is "ignored."

Frailey [1973] describes the resource management system developed for Purdue University's MACE operating system (for a CDC 6600). In this system, many devices (core memory, I/O channels, etc.) are allowed to be preempted. The state of the process using the preempted resource is saved so that the process may be restarted. The resources determined to be preemptive are just those for which restart is not too troublesome. Certain resources, such as data files, are non-preemptive. These are handled as follows (in Frailey's words).

"Each claim is treated as a request and, if the resource is not available, the task waits for it. Otherwise, access is granted. In this method, no more than one resource is requested at any time by a single task . . . Multiple claims are not allowed."

Notice that nothing is said about releasing previously held resources before making such a claim. The system does nothing to prevent deadlock that may arise from locking of non-preemptive resources. However, Frailey reports that very few deadlocks actually occur (two in a year), probably because "the most frequently used resources are treated as preemptive." Frailey concludes: "In the final analysis, the cost of full deadlock prevention was felt to be too high for complete implementation."

Sekino [1975] describes a deadlock prevention mechanism implemented within a Bell Laboratories data management system. The basic scheme is to eliminate circular waits by ordering all elements in the data base. However, because of the inefficiencies which may arise if the user is always required to access elements in, say, ascending order, the user is allowed to access data elements in arbitrary order. If he is blocked in accessing an element with a lower number than is attached to one he already holds locked, then he must release all higher numbered elements. Consider the example pictured in figure 4. When P_3 fails to obtain R_1, it must release R_4. This prevents the deadlock which would otherwise occur at time T_6. On the other hand, when P_1 failed to obtain R_2 it would release R_3 - an unnecessary action. This scheme is therefore a variation on the standard approach for elimination of the wait-for condition. The difference is that instead of requiring that all currently held resources be released before more are allocated, only the set which might become involved in circular wait (i.e., the lower ordered ones) need be released.

Techniques proposed for distributed data bases. There is very little in the literature on the deadlock problem in a network environment. In fact, the only two relevant papers that we know of have appeared within the past two years. These are a symposium paper by Chu and Ohlmacher [1974] and Mahmoud's Ph.D. thesis [1975].

The simplest scheme discussed by Chu and Ohlmacher is of the preclaim type. The process must specify in advance all of the files it might need and these are all allocated before the process is initiated. If all files needed are at a single site, the system there can decide whether they are all available and then allocate them. If the files are at several sites, an additional mechanism is needed. Otherwise a circular wait may occur if (as is desirable) each site has the power to allocate its own files independently of the other sites. This possibility of circular wait is eliminated by simply numbering the sites and always distributing requests to sites in numerical order.

As a refinement of this technique, Chu and Ohlmacher introduce the notion of _process sets_, or sets of processes having requests for the same file(s). Thus processes in different process sets can never interfere with one another, and this may simplify the allocation problem. For example, suppose there are three processes in the system - two in one set and one in a second. The one in the second set can not interfere with the others; it can be initiated without preallocation of resources.

Chu and Ohlmacher also discuss a detection mechanism of the standard graphical kind. They suggest that implementation be "accomplished by appointing one node of the network to monitor requests for files and detect deadlocks." They do not address the difficult problems of restart/recovery in a network, nor do they indicate what should happen if the monitoring node fails.

Mahmoud [1975] discusses Habermann's avoidance technique in some detail and briefly indicates how it may be extended to a distributed situation. Again, the difficulties posed by the need for some central site to monitor the allocations are ignored. A so-called demand-graph model is also discussed, but the basic idea is the same - i.e., a restriction of the system to "safe" allocation states. Finally, Mahmoud considers a deadlock detection mechanism which (like that of Chu and Ohlmacher) does not appear particularly novel. He suggests an optimal preemption scheme (but does not give details) for "recovery" when deadlock is detected.

Assessment of the State of the Art

Our primary interest is the prevention of deadlocks due to data access by concurrent processes. It is this problem that we will be considering here. We hope, however, that the reader has not forgotten the discussion above of the SYSTEM/360 Operating System. There was an important lesson to be learned there - that even if one has an impeccable scheme for preventing data-related deadlocks, one must still consider interactions between data allocation and allocation of other devices.

Complexity of avoidance. In reviewing the techniques discussed above, we are immediately struck by the complexity of the deadlock avoidance schemes. Even if it is possible - as Mahmoud seems to claim - to implement such a scheme in a distributed environment, the overhead, both in cost and in delay, seems to be prohibitive.

Feasibility of detection and recovery. Deadlock detection schemes, though they also require complicated monitoring, have a much smaller overhead and might be feasible. Indeed, Chu and Ohlmacher claim that a standard detection mechanism is "easily implemented" in a network. However, they ignore the problem of what should be done if the monitoring node fails. If deadlock is an infrequent occurrence, perhaps the concurrent failure of the monitor is of sufficiently low probability that this possibility can be tolerated. It is also possible to implement deadlock detection with distributed control. A scheme for doing this is discussed below.

Recovery after detection of a deadlock is a more serious problem. If a process that is updating the data base is aborted, the data may be left in an inconsistent state. King and Collmeyer's scheme or some equivalent approach - in which the updates are not really applied to the data base until after the process is complete - seems to be the only feasible solution in sight. It should be possible to implement this solution for a distributed data base, although the overhead will be high.

Difficulties of prevention. We are therefore led to considering the deadlock prevention mechanisms in more detail. We have already noted that the elimination of mutual exclusion is not acceptable. And allowing the preemption of resources brings us right back to the recovery problem. We are left with the possibility of eliminating either the wait-for condition or the circular-wait condition.

Consider first the circular-wait condition. The possibility of circular wait may be eliminated by ordering all of the lockable elements in the data base. If all processes access the elements in the same order, then clearly a circular wait can not occur. Two problems arise in implementing this approach. First, it may be expensive to maintain an ordering on a very large, dynamic, distributed data base, particularly if the lockable units are small. If the lockable units are whole relations (or files) - and these are added or deleted infrequently - then ordering is quite feasible. All one needs to do is to assign a unique name to every resource at each host. The ordering can then be any natural sort order, e.g. alphabetic. The only problem is that the system overhead to keep track of the ordering can become large in a dynamic environment where new names are constantly being created.

Second, there is the problem of implementing the requirement that data units be accessed in the prescribed order. If the requirement is implemented at the level of the user or applications program, several difficulties occur.

1. The order of the data elements must be specifically known to the user.
2. The programmer must take into account the data ordering in designing an efficient program. That is, an unnatural order may be imposed on the job, and the programmer should try to work around this.
3. Some monitoring device will have to exist in the system to see that the prescribed order is not violated. If the order is violated, some action must be taken.

Everest [1974] summarizes the difficulties nicely: "It may be impossible to establish a preordering ... such that all processes could operate properly by requesting the files according to the preordering. In any case, it would impose a severe, and perhaps unnecessary, discipline on the programmer."

To avoid these difficulties, the burden of allocating resources in the proper order could be placed on the system. The ordering of the lockable data units is then transparent to the user. But the system then requires complete information on all the resources that may be required by a process in order to properly order the allocation. For example, if the process works with resources numbered 100 through 150 and then, much later, (might) access number 1, the system must know this and begin by attempting to allocate resource number 1. Thus the process must essentially preclaim all the resources that it may possibly need.

The basic technique for elimination of the wait-for condition is also preclaiming. (The alternative is to require processes to release previously held resources before requesting more - or possibly for the system to preempt some or all of the previously held resources. This alternative again requires recovery procedures or discipline on the part of the user - that is, the process must be prepared to release resources without drastic results.) We therefore see that preclaiming - in the sense that the process must know in advance the complete set of resources it might need and it must convey this knowledge to the system before any allocations are made - is a necessity in any foolproof scheme for treating deadlock without worries about recovery.

What the system then does with this perfect information is another question. It may wait until the entire set of resources is available and allocate them as a group before initiating the process. It may go through an elaborate deadlock avoidance scheme, allocating resources to processes one by one and checking that each step is "safe". The former ties up resources for unnecessarily long periods of time; the latter requires enormous system overhead. As a compromise, the system could take some intermediate approach. Allocating resources in some fixed order is an example of this. Another example is Chu and Ohlmacher's use of process sets.

In summary, we quote again from Everest: "The conclusion to be drawn is that _if_ processes are to run concurrently, _if_ data is a dominant resource, and _if_ deadlock is to be avoided, _then_ all processes must state their resource needs a priori."

Elimination of concurrency. We see clearly from this quotation that there is another possible way to prevent deadlock in data management. This way - which is foolproof - is to eliminate concurrency. That is, we may simply not allow independent processes to access data at will. Such a scheme may be implemented, for example, by requiring all data access to take place through a central data manager. The data manager would essentially act as a batch processor, being itself a _single_ process which handles sets of one (or more) queries which have been submitted to it. Processes may find themselves waiting for some length of time for responses to their queries, but they are guaranteed that they will not be blocked forever.

In spirit, this solution is much like that of eliminating circular wait by putting the burden on the "system" to see that records are accessed in some prescribed order. In that solution also, processes are not allowed to act independently to get in each others' way.

Indeed, imposition of a centralized control for each data base may be the best solution to the deadlock problem in the database context. Notice that it is only updating, in general, which requires mutual exclusion. It may be that only updates should be batched and run by a central process, while (whenever updating in not taking place) retrievals can be carried out concurrently and in interactive mode.

Even if this solution is adopted, however, there may be other concurrent processes which can deadlock in a network environment. If these deadlocks are not very common, they might best be handled by simple detection. We have, therefore, looked closely at the problem of implementing a distributed detection scheme in a network. Our proposed algorithm for distributed detection is presented and discussed in the following section.

Distributed Deadlock Detection in a Network

As we discussed above, there are serious disadvantages to setting up a central monitor to detect deadlock in a network environment. There appears to be no reason why detection can not be carried out in a decentralized manner. We suggest the following scheme, which is based on King and Collmeyer's access-state-graph analysis of deadlock. (See the introduction above.)

Suppose that a process P_1 is blocked in attempting to claim resource R_1. P_1 is then placed on the list of blocked processes waiting for R_1. After some reasonable time interval has passed, the site where P_1 is located suspects that P_1 is in a deadlock and initiates the following algorithm to detect a cycle in the access state graph.

1. A resource list is initialized with R_1.

2. The name of the process holding R_1 is obtained. (This step - and similar steps - will in general involve querying other sites in the network.) Call this process P_2. Initialize a process list with P_2.

3. For every process in the process list which is not already tagged as having been checked, check to see if it is blocked. If it is, add the resource (or resources) it is waiting for to the resource list. In any case, tag it as having been checked. If no new resources are added in this step, stop. P_1 is not in a deadlock cycle.

4. For every new resource added to the resource list in step 3, add the process holding it to the process list.

5. Check the process list. If P_1 appears in this list, then a deadlock exists.

6. Go to step 3.

Notice that this algorithm will determine whether or not P_1 is itself involved in a deadlock cycle, but will not detect the fact that P_1 is waiting for a resource which is tied up in a deadlock not directly involving P_1. This is probably a reasonable approach. A site is likely to want to take drastic action against (e.g., abort) a process which is in a deadlock. But presumably a process that is merely waiting for a deadlocked resource will shortly be able to proceed normally after the deadlock is detected and broken up. This presumption depends upon all sites' being equally alert to detecting and following up on possible deadlock situations. A site may wish to consider the duplication of any process P_i in the process list as evidence of a possible deadlock involving P_i and to either

a) initiate a deadlock check beginning with P_i, or

b) notify the site where P_i is located that it should initiate such a check.

If each process can be waiting for no more than one resource at a time, then each non-terminating loop through steps 3-6 will add precisely one resource to the resource list and one process to the process list. In this case a duplication in the process list does demonstrate the existence of a deadlock cycle. The site then <u>knows</u> that P_1 can not proceed until the needed resource is released from the deadlock. Appropriate action can then be initiated.

Notice that the algorithm may not see a deadlock which does not actually occur (i.e., the cycle is not completed) until after the detection algorithm is initiated. Again, we assume that such a deadlock will eventually be detected. When the process which completes the deadlock cycle becomes blocked, the algorithm will necessarily detect the cycle.

It is important to consider the overhead of such a detection scheme. Data structures must be maintained to provide the necessary information for the detection scheme. For example, each site might keep a list of its processes and the resources each is waiting for and/or a list of its resources and the processes waiting for each. (It might also be possible to store all the necessary data at a central location, but we will assume here that in a truly distributed detection mechanism we would want the information on processes and resources to be distributed. Otherwise we have problems of availability, of updating the centralized data, etc.) In the preliminaries to the algorithm, we stated that a list of waiting processes should be kept for each resource; this type of list simplifies the search required for step 4. On the other hand, if step 3 is not to require a search of all queues for all resources at all sites, inverted lists of the resources waited for by each process are also needed. How much overhead is required to keep these lists will depend upon how many - and what kind of - resources are involved in the detection scheme. For example, the system will almost surely be keeping track of buffer allocations anyway. But suppose that in a large data base each record can be locked by any process. Even if keeping track of all records locked by all processes can be made possible, the system overhead would be enormous.

Another aspect of the algorithm which requires analysis is the time needed to search for the deadlock cycle. The major contributor to this time will be the time delays of network communications. In general, a delay of on the order of one second will be incurred every time a site is queried across the network. The total time for the search will then be roughly s seconds, where s is the number of times a remote site must be queried in following through the chain of resources. The parameter s is not an easy one to estimate. In many cases, most of the resources in the chain will be held locally. In other cases, the resources may be mostly remote. Even if the network has few sites, repeated querying of a remote site may be necessary to follow the resource chain. (It should be possible, by introducing some further complexity into the detection process, to provide remote sites with a mechanism for following through portions of the resource chain held locally without communicating with the site that initiated the search on every step.) In general, we expect s to be on the order of 2fr, where r is the average length of the resource chain which must be searched to determine whether or not there is a deadlock, and f is the fraction of those resources expected to be remote. The factor of two arises because for each resource checked, the process holding it must also be checked, and this may require going to a second site.

In turn, the parameters f and r will depend in complicated ways on how the resources are distributed among the sites in the network, which resources are likely to become involved in real (or suspected) deadlock, how many processes are running concurrently, etc. It is probably impossible to predict the values of these parameters for a given system. Simulation studies may be of help if pertinent system characteristics may be defined closely enough so that the results are valid. On the other hand, once a simple deadlock detection mechanism is implemented for a system, it would be a simple matter to gather statistics on these parameters.

By assuming that s is proportional to r, the length of the resource chain to be searched, we have tacitly assumed that the local site knows where the resources (and thus information on the processes holding them) are located. This may not be true. It may be that, unless a resource is held locally, the detection process will have to query each site in turn until the resource is found. In this case we would have (if there are n+1 sites in the network), $s \approx frn$, since on the average one would expect to have to query half of the remote sites before finding the resource (or process). As another possibility, the detection mechanism might have to query a directory, located remotely, to find out where each resource or process is. This would mean two remote accesses for each remote resource or process, or $s \approx 4fr$. What we are getting into here is the <u>directory problem</u> - the development of efficient techniques for locating resources (or processes) in a network. This is a problem for further study. In this report, we are limited to pointing out that how the directory problem is solved can have an important impact on deadlock detection.

Another research area that impacts on deadlock detection is that of <u>name space management</u>. Process and resource names throughout the network must be chosen from the same name-space. That is, each process and resource must be assigned a unique name by which it is recognized by all systems in the network. Without such unique names, it would clearly be impossible to carry out the detection scheme. Furthermore, the naming mechanism can have some effect on the efficiency of detection. For example, if one component of a resource name is the name of the site holding the resource, then knowing the name is tantamount to knowing its location and there is no need to consult directories or to query sites to find it.

In summary, it seems to be quite possible to implement a distributed deadlock detection mechanism in a network. Such a scheme would probably be practical for handling potential deadlocks among only a rather limited set of resources. For large numbers of resources (e.g., records in a data base) deadlock is best prevented. But if, in a relatively few cases, prevention is not feasible, distributed detection is definitely a viable safeguard. We leave open the question of what action to take if a deadlock is detected. The local system may have no choice but to abort the process with which the check began. If this process has very high priority, however, this solution would be undesirable. It should be possible to develop a scheme by which sites involved in a deadlock may cooperate in deciding on what action to take. A point to keep in mind in developing such a cooperating scheme is that two sites may discover the same deadlock more or less simultaneously, and some safeguard may be needed to prevent them from working at cross purposes during recovery.

Conclusions; Areas Needing Further Study

The literature on the deadlock problem is relatively extensive. Basic techniques for handling the problem have undergone considerable study and have been developed to a high level of sophistication. Nothing very novel in the way of deadlock treatment is likely to appear in the future; the tools seem to be complete and ready for use.

There are three questions which do require further study.

1. In a given situation, which technique - or combination of techniques - should be implemented?

2. How may the chosen technique (or techniques) be implemented in a network (distributed) environment?

3. What action should be taken when a deadlock is detected?

One can attempt to answer the first question by making qualitative arguments. This is the approach we took in our assessment of deadlock techniques. Avoidance techniques, for example, seemed "obviously" to entail too high an overhead (besides requiring a central monitor) to be practical for a large, distributed data base. After a certain point, however, qualitative discussions no longer yield definitive answers. For example, we have discussed in some detail a distributed detection scheme. But is the implementation of such a scheme really worthwhile? The answer to this question will depend upon a careful analysis of "cost" (storage, CPU time, etc.) tradeoffs. Which is less expensive - letting an occasional deadlocked process tie up resources temporarily before it is aborted for being in the system "too long", or maintaining a deadlock detection mechanism to abort processes on a more rational basis? It may be impossible to provide a clearcut answer to this question, since so many factors are involved. Indeed, a need for the ability to detect promptly a deadlock involving a high priority, critical process may be of such incalculable importance that it outweighs all cost considerations and obviates careful analysis of tradeoffs in "average" situations. Nevertheless, such an analysis would be useful.

We have provided at least a partial answer to the second question posed above in the case of deadlock detection. Further analysis of our algorithm, in conjunction with some experimentation, would, however, help us to identify any weaknesses and test its practicality. Again, only experimentation can demonstrate the feasibility of carrying out synchronization in such a way that deadlock is prevented. The completion of the experimental distributed data management system now being designed at the Center for Advanced Computation will give us a much needed tool. Using this tool, we expect to arrive at a greatly improved understanding of how best to treat the deadlock problem in a distributed environment.

The third question brings up the problem of recovery. Even if it is determined that a sophisticated scheme for preserving states of processes or resources is neither feasible nor necessary, there is still the problem of deciding which process to abort. We have hinted at the possibility of constructing a distributed scheme for making such a decision. It would be worthwhile to examine this possibility in detail. Superficially, there seems to be no barrier to making this decision on a distributed basis, taking into account priority and/or policy information provided by the various sites holding the processes. Again, this would be an interesting problem for study in the context of the experimental system.

References

Belford, G.G.; J.D. Day; S. Sluizer; and D.A. Willcox
1975 "Initial Mathematical Model Report" CAC Document Number 169, JTSA Document Number 5511; Aug. 1975.

Belford, G.G.; P.M. Schwartz; and S. Sluizer
1976 "The Effect of Backup Strategy on Data Base Availability," CAC Document Number 181, CCTC-WAD Document Number 5515; Feb. 1976.

Bunch, S.R.
1975 "Automated Backup" in Preliminary Research Study Report, CAC Document Number 162, JTSA Document Number 5509; May 1975.

Chu, W.W. and G. Ohlmacher
1974 "Avoiding Deadlock in Distributed Data Bases" ACM Nat'l Symp. 1, Nov. 1974, pp. 156-160.

Coffman, E.G., Jr; M.J. Elphick; and A. Shoshani.
1971 "System Deadlocks", Computing Surveys 3 (No. 2) June 1971, pp. 67-78.

Dijkstra, E.W.
1968 "Cooperating Sequential Processes," Programming Languages, F. Genuys, ed. Academic Press, pp. 43-112. [First published by T.H. Eindhoven, The Netherlands, 1965].

Ellis, C.A.
1973 "On the Probability of Deadlock in Computer Systems," Operating Systems Review 7 (No. 4), pp. 88-95.

Eswaran, K.P.; J.N. Gray; R.A. Lorie; and I.L. Traiger
1974 "On the Notions of Consistency and Predicate Locks in a Data Base System" to appear in CACM. Available from IBM Research as report RJ 1487; Dec. 1974.

Everest, G.C.
1974 "Concurrent Update Control and Database Integrity" Data Base Management, J.W. Klimbie and K.L. Koffeman, eds., North-Holland, pp. 241-270.

Frailey, D.J.
1973 "A Practical Approach to Managing Resources and Avoiding Deadlocks," CACM 16 (No.5) May 1973, pp. 323-329.

Grapa, E.
1975 Internal memorandum, Center for Advanced Computation, University of Illinois at Urbana-Champaign.

Grapa, E.
1976 Internal memorandum, Center for Advanced Computation, University of Illinois at Urbana-Champaign.

Habermann, A.N.
1969 "Prevention of System Deadlock," CACM 12 (No.7), July 1969, pp.373-377,385.

Havender, J.W.
1968 "Avoiding Deadlock in Multitasking Systems," IBM Systems Journal 7 (No.2), pp. 74-84.

Holt, R.C.
1972 "Some Deadlock Properties of Computer Systems," Computing Surveys 4 (No.3), Sept. 1972, pp. 179-196.

Johnson, P.R. and M. Beeler
1974 "Notes on Distributed Data Bases:, Draft Report, available from the authors (Bolt Beranek, and Newman, Inc., Cambridge, Mass.)

Johnson, P.R. and R.H. Thomas
1975 "The Maintenance of Duplicate Databases," RFC #677, NIC #31507, Jan. 1975. (Available from ARPA Network Information Center, Stanford Research Institute - Augmentation Research Center, Menlo Park, CA.)

King, P.F. and A.J. Collmeyer
1973 "Database Sharing - An efficient mechanism for supporting concurrent processes," AFIPS NCC 1973, pp. 271-276.

Lamport, L.
1974 "A New Solution to Dijkstra's Concurrent Programming Problem," CACM 17, pp. 453-455.

McCauley, E.J. and P.A. Alsberg
1975 "Scenario Report," CAC Document Number 159, JTSA Document Number 5506, May 1975.

Mahmoud, S.A.
1975 "Resource Allocation and File Access Control in Distributed Information Networks." Ph.D. Thesis, Carleton University, Ottawa.

Murphy, J.E.
1968 "Resource Allocation with Interlock Detection in a Multi-task System" AFIPS FJCC 1968, pp. 1169-1176.

Schuster, S.
1976 Personal communication.

Sekino, L.C.
1975 "Multiple Concurrent Updates," Preprint, Bell Telephone Laboratories, Holmdel, N.J.

Shemer, J.E. and A.J. Collmeyer
1972 "Database Sharing: A Study of Interference, Roadblock, and Deadlock," Proc. 1972 ACM-SIGFIDET Workshop, pp. 147-163.

Shoshani, A. and A.J. Bernstein
1969 "Synchronization in a Parallel-Accessed Data Base" CACM 12 (No.11), Nov. 1969, pp. 604-607.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CAC Document Number 185 CCTC-WAD Document Number 6503	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Research in Network Data Management and Resource Sharing - Synchronization and Deadlock		5. TYPE OF REPORT & PERIOD COVERED Research
		6. PERFORMING ORG. REPORT NUMBER CAC #185
7. AUTHOR(s) Peter A. Alsberg et al.		8. CONTRACT OR GRANT NUMBER(s) DCA 100-75-C-0021
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Advanced Computation University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center WWMCCS ADP Directorate 11440 Isaac Newton Sq., N., Reston, VA. 22090		12. REPORT DATE March 1, 1976
		13. NUMBER OF PAGES 85
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Copies may be obtained from the
National Technical Information Service
Springfield, Virginia 22151

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No restriction on distribution

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed data management
database access synchronization
computer system deadlock
network protocol resiliency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document presents the results to date of a research study of the problems of synchronization and deadlock. Particular emphasis is on applicability of solutions to a network environment and to data base access. In addition to providing a review and assessment of currently available techniques, this paper presents some new ideas in the areas of protocol resiliency, decentralized deadlock detection, and maintenance of update order.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUC-CAC-DN-76-185 ✓	2.	3. Recipient's Accession No.

4. Title and Subtitle Research in Network Data Management and Resource Sharing - Synchronization and Deadlock	5. Report Date March 1,1976
	6.
7. Author(s) Peter A. Alsberg et al.	8. Performing Organization Rept. No. CAC #185 ✓
9. Performing Organization Name and Address Center for Advanced Computation ✓ University of Illinois at Urbana-Champaign Urbana, Illinois 61801	10. Project/Task/Work Unit No.
	11. Contract/Grant No. DCA100-75-C-0021 ✓
12. Sponsoring Organization Name and Address Command and Control Technical Center WWMCCS ADP Directorate 11440 Isaac Newton Sq., N., Reston, Va. 22090	13. Type of Report & Period Covered Research
	14.

15. Supplementary Notes

16. Abstracts

This document presents the results to date of a research study of the problems of synchronization and deadlock. Particular emphasis is on applicability of solutions to a network environment and to data base access. In addition to providing a review and assessment of currently available techniques, this paper presents some new ideas in the areas of protocol resiliency, decentralized deadlock detection, and maintenance of update order.

17. Key Words and Document Analysis. 17a. Descriptors

distributed data management
database access synchronization
computer system deadlock
network protocol resiliency

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement No restriction on distribution Available from the National Technical Information Service, Springfield, VA 22151	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 85
	20. Security Class (This Page) UNCLASSIFIED	22. Price