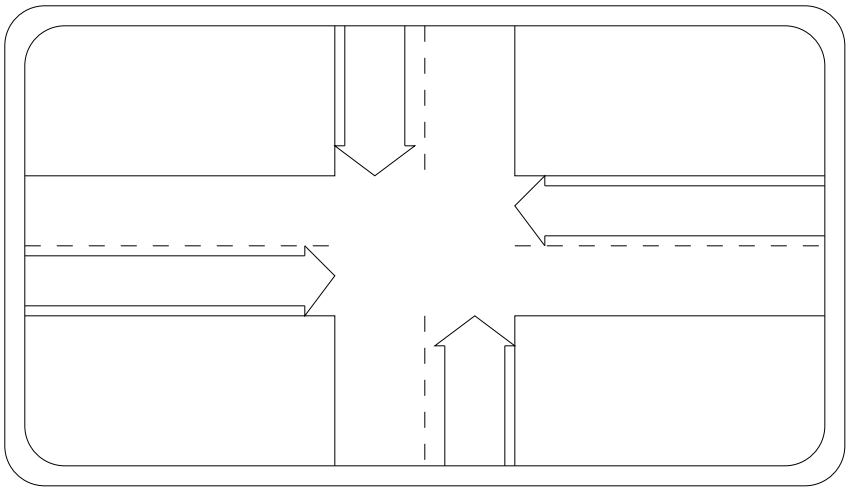

Deadlock is a constant threat in terminal-oriented systems. This comprehensive study of deadlock-handling techniques introduces a method for on-line detection in distributed databases.

The Deadlock Problem: An Overview

Sreekaanth S. Isloor*
T. Anthony Marsland

University of Alberta

Modern multiprogramming systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. The increasing trend among commercial firms for on-line operations, especially those involving integrated databases, and the consequent need by active users for responsive systems have placed heavy demands on operating systems. Compounding these difficulties, distributed processing has arrived as the solution to incremental system growth. Such contemporary systems exhibit a high degree of resource and data sharing, a situation in which deadlock is a constant threat. Deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group. When no member of the group will relinquish control over its resources until after it has completed its current resource acquisition, deadlock is inevitable and can be broken only by the involvement of some external agency.

*Now at Bell Northern Research, Ottawa.

A set of processes becomes deadlocked as a consequence of *exclusive access* and *circular wait.* The simplest illustration of these conditions involves only two processes, each holding, for exclusive access, a different resource and each requesting access to the resource held by the other. The result is a circular wait which cannot be broken until one of the processes releases its resource or cancels its request.

In this article, we survey the resource management problem in computer systems by reviewing the principles, techniques, and tools involved in handling and avoiding deadlocks; we also propose a new method for handling deadlock in distributed databases, closing with some case studies on the resource management problem in large computer systems. Annotated references to works cited and a comprehensive bibliography of supplementary materials are provided.

Examples of deadlock

The deadlock problem occurs in many different contexts, and analogies can be made to real-life situations, provided one interprets the processes and resources involved appropriately. For instance, one often hears about "deadlocked" peace talks between two warring nations. In that context, the peace-negotiating parties of the two countries are the processes, and the occupied territories could represent the resources over which exclusive control is sought. The peace talks could be blocked indefinitely if both parties refuse to give up any occupied land while demanding the return of some land held by their adversary.

Another common example is the traffic deadlock. Consider the situation (Figure 1) where four cars, *A, B, C,* and *D,* arrive at an intersection at approximately the same time. To prevent accidents at four-way stop intersections, right-of-way regulations require that *A* yield to *B, B* to *C,* and so on. As far as *A* is concerned, part of the intersection belongs to *B* for exclusive use, so *A* must wait until *B* passes. A circular wait is an essential component of this traffic deadlock; if all four vehicles move forward, occupying as much of the intersection as possible, all traffic comes to a standstill. In this illustration, the cars are the processes, while the four quarters of the intersection are the resources over which exclusive control is needed.

The deadlock situation is different depending on whether the cars want to turn right, turn left, or go straight. Assuming the exit route for each car is clear, the apparent deadlock does not materialize when all the cars wish to make a right turn, since the cars require access to one quarter of the intersection which no others demand. Things are more difficult when the cars want to proceed ahead or turn left. To go straight, car *A* requires quarters 1 and 2 of the intersection. In these circumstances, a deadlock arises since each car needs exclusive access to a quarter of the intersection to which the car on its right side holds a legal right (controls).

The traffic problem can be handled through the interference of an external agency (e.g., a policeman) allocating space to one of the vehicles. Alternatively, some busy cities "cross-hatch" important intersections, and require that no vehicle enter that area unless its exit route is clear. The cross-hatching technique ensures that no process will acquire a resource (occupy the intersection) which it cannot subsequently relinquish (leave by a clear exit). As a practical matter, traffic deadlock is usually resolved by one driver aggressively entering the intersection, preempting it from the others. As an alternative, one might force vehicles to roll back a random distance and approach the crossroads again. Since each driver thus delays his entry by a different amount, the problem caused by their simultaneous arrival is eliminated. A similar technique is used in Ethernet[1] to resolve access conflicts on its communication medium and is also usual on contention-mode telecommunication circuits.

For deadlocks that may occur in computer operating and database systems, we will use the term "system resource" quite broadly to refer to *storage media* (e.g., primary memory, tapes, disks, and drums), *system components* (e.g., tape drives, disk drives, I/O channels, CPUs, readers, and printers), and *information*

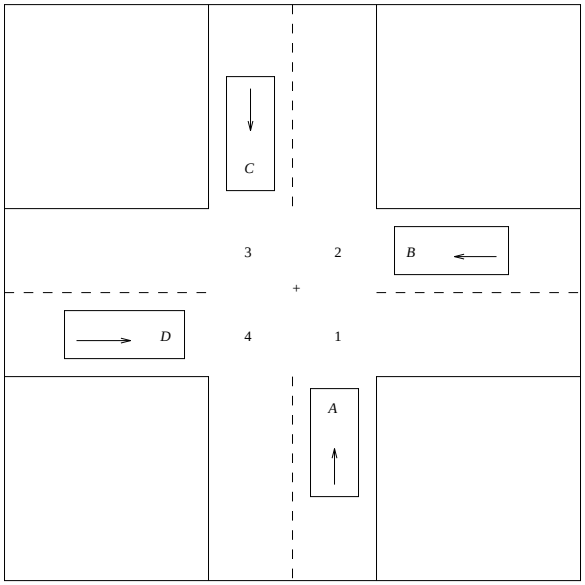

Figure 1. Traffic deadlock at an intersection illustrates a circular wait, in which the cars (A-D) are the processes and the quarters of the intersection (1-4) are the resources over which control is needed.

(e.g., communication messages, data records, files, directories, programs, and system routines). Consider a small multiprogramming system with a single card reader and a printer, in which two user jobs share use of the printer and the card reader by means of request and release operations, as given in standard operating systems texts.[2] Due to independent scheduling of the jobs, request and release operations can be interspersed in several different orders. Some of these sequences lead to a "deadly embrace," a deadlock where two jobs, for example, hold the printer and the card reader, respectively, and—at the same time—request the unit held by the other.

More generally, a set of waiting processes forms a circular chain where each process holds at least one resource and makes a conflicting access request for some resource held by the next process in the chain. Such a circular wait condition can arise when the following necessary conditions hold:

- processes request *exclusive control* of resources,
- processes *hold* resources allocated to them while *waiting* for additional ones, and
- *no preemption* of a resource from a process can be done without aborting the process.

In many ways these conditions are quite desirable. For consistency, data records should be held until update is complete. Similarly, preemption (the reclaiming of a resource by the system) cannot be done arbitrarily and, especially when data resources are involved, must be supported by a *rollback* recovery mechanism. Rollback restores a process and its resources to a suitable previous state from which the process can eventually repeat its transactions.

For the database case (Figure 2), consider two concurrently executing processes P and Q, which modify entities M and N. Let us assume that the database correctness (consistency) assertion on the entities is $M = N$ and that the initial values of M and N are the same. Interleaving the actions of processes P and Q in an arbitrary fashion can lead to different database values. Although it is possible to construct the processes

PP_1: $M = M + 100$ P_2: $N = N + 100$		QQ_1: $N = 2 * N$ Q_2: $M = 2 * M$	
PROCESS P		PROCESS Q	
STEP	ACTION	STEP	ACTION
P_0	REQUEST ENTITY M	Q_0	REQUEST ENTITY N
P_1	LOCK ENTITY M	Q_1	LOCK ENTITY N
P_2	READ ENTITY M	Q_2	READ ENTITY N
P_3	WRITE ENTITY M	Q_3	WRITE ENTITY N
P_4	REQUEST ENTITY N	Q_4	REQUEST ENTITY M
P_5	LOCK ENTITY N	Q_5	LOCK ENTITY M
P_6	READ ENTITY N	Q_6	READ ENTITY M
P_7	WRITE ENTITY N	Q_7	WRITE ENTITY M
P_8	UNLOCK ENTITY M	Q_8	UNLOCK ENTITY N
P_9	UNLOCK ENTITY N	Q_9	UNLOCK ENTITY M

Figure 2. Consistent database deadlock. The sequence of steps leading to deadlock is $p_0q_0p_1q_1p_2p_3q_2q_3p_4q_4$.

so that database correctness is maintained, concurrent operation can still lead to deadlock. Therefore a consistency result has been developed which requires that the processes be "well-formed" and "two-phase."[3]

A well-formed process is one which locks an entity before acting on it further and subsequently unlocks it. A process is thus required to be divided into growing (locking) and shrinking (unlocking) phases. The first unlock action signals the beginning of the shrinking phase, after which a process cannot issue a lock request on any entity in the database until all other entities held have been released. In the context of Figure 2, it is essential to note that the process P (or Q) cannot unlock entity M (or N) before locking entity N (or M), if it is to maintain database correctness. Note also that under concurrent operation the interleaved sequence of actions p_0 q_0 p_1 q_1 p_2 p_3 q_2 q_3 p_4 q_4 ends in deadlock. Several other aspects of concurrent operation such as transaction-, lock-, log-, and recovery-management need not concern us and have been dealt with thoroughly elsewhere.[4]

Basic approaches to deadlock handling

One-basic strategy for handling deadlocks is to ensure violation of at least one of the three conditions necessary for deadlock (exclusive control, hold-wait, and no preemption). This method is usually referred to as deadlock prevention, unless its primary aim is to avoid deadlock by using information about the processes' future intentions regarding resource requirements. A totally different strategy interrogates the process/resource relationships from time to time in order to identify the existence of a deadlock. This latter method presumes that the system can subsequently do something about the problem.

Detection techniques. These techniques assume that all resource requests will be granted eventually. A periodically invoked algorithm examines current resource allocations and outstanding requests to determine if any processes or resources are deadlocked. If a deadlock is discovered, the system must recover as gracefully

as possible by preempting resources from affected processes until the deadlock is broken.

Detection-scheme overhead includes not only the run-time cost of the algorithm but also the potential losses inherent in preempting resources. Since no action takes place until a deadlock actually occurs, resources may be held idle by blocked processes for long periods of time. Sometimes, using detection principles effectively is difficult—for example, when preemption of resources such as tape drives might incur unacceptable overhead. Nevertheless, detection techniques have some advantages since the schemes are invoked intermittently and only essential preemptions need be performed.

In the database context, detection methods rely on the management system to abort, roll back to a previous checkpoint, and restart at least one process to break the deadlock. Here, the problem of rollback and recovery assumes great importance from the viewpoint of maintaining database consistency.

Prevention mechanisms. Prevention is the process of constraining system users so that requests leading to deadlock never occur. Most proposals for prevention require each process to specify all needed resources before transactions begin. Deadlocks can be prevented in several ways, including requesting all resources at once, preempting resources held, and ordering resources.

The simplest way of preventing deadlock is to outlaw concurrency, but this leads to very poor resource utilization and is not consistent with current system design philosophies. Another method requires that all resources be acquired before processing starts. Such a scheme is inefficient, since resources held may be idle for prolonged periods, but works well for processes which perform a single burst of activity, such as input/output drivers, since the resource can be released immediately after each use. For processes with fluctuating requirements, the method can be impractical. In a database environment, it may be impossible for a data-driven process to specify and acquire all needed resources before beginning execution. In any case, the scheme discriminates heavily against data-driven processes where relationships in the data indicate what future resources are required for processing.

Certain other prevention methods require a blocked process to release resources requested by an active process. For example, when a process needs more main memory than is currently available, it becomes blocked. Subsequently the process is swapped to secondary storage by preempting its memory for use by an active process. The blocked process is swapped back only when the entire, larger quantity of memory is available. For some peculiar situations in database systems, this use of preemption to prevent deadlocks is subject to *cyclic restart,* in which two or more processes loop by continually blocking, aborting, and restarting each other.

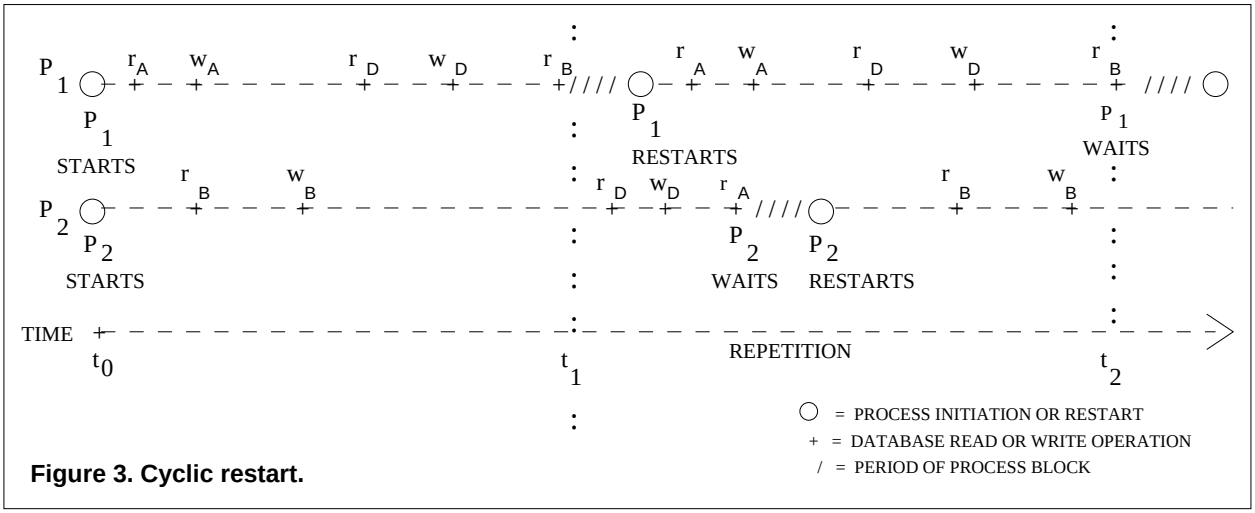

Figure 3. Cyclic restart.

Consider two processes P_1 and P_2, checkpointed and starting at time t_0, as shown in Figure 3. In the first phase, processes P_1 and P_2 issue update requests $\{w_A, w_D\}$ and $\{w_B\}$, respectively, on entities $\{A, D\}$ and $\{B\}$. At time $t_1 > t_0$ process P_1 requests access to entity $B(r_B)$ and gets blocked. Shortly afterwards, P_2 requests access to $D(r_D)$, resulting in a deadlock. The deadlock arbitrator selects the blocked process P_1 for abortion and restart. In this case, the nature of the transactions are such that the situation at time t_1 is repeated at time t_2. Such a cycle tends to be self-synchronizing, continuing indefinitely, notwithstanding minor system environment variations in timing.

Havender[5] developed a set of approaches which excludes a priori the possibility of deadlock by restricting the way in which resources can be requested. In one approach, all the required resources must be requested and granted before the process can proceed. In a second strategy, when a process holding certain resources is denied a further request, the process must be capable of releasing all of its original resources and rerequesting them, together with the additional ones.

A more sophisticated form of prevention employs a unique ordering of the resources.[5] A request for a specific resource is met only if all resources lower in rank, which are needed in the future, have also been allocated. By this means, circular chains of blocked processes cannot occur, since each process requests resources in the same orderly way. The feasibility of enforcing resource ordering by compile-time checks is a major advantage of the scheme. Restrictions on the allowable sequences of process requests force knowledgeable use of the ordering rule. A process may request and hold some units of a resource rather early in the processing stage, in which case a later incremental request for the same resource may be disallowed by the ordering rule. Consequent recovery is possible only by preempting all the resources held by the process.

In a database environment with data-driven processes of fluctuating needs, it is often impossible to order the data resources (records, entities, or fields), so this resource ordering method is seldom applicable.

Avoidance schemes. In avoidance schemes, a resource request is granted only if at least one way remains for all processes to complete execution. One basic scheme, referred to as the "banker's algorithm,"[6] manages multiple units of a single resource by requiring that the processes specify their total resource needs at initiation time. Furthermore, each process acquires or returns resource units one at a time. The algorithm denies a request by any process whose remaining needs are in excess of the available resources. Effectively,

the scheme projects detection into the future to keep the system from committing itself to an allocation which eventually leads to deadlock.

Haberman developed a "maximum claims strategy"[7] to control the future resource requirements for each process. This generalization of the banker's algorithm is a practical example of avoidance but requires quantity information, in the form of upper bounds, on every resource the process needs. Thus, if there is a process which can run to completion using only its allocated resources and those that are immediately available, then the current state of the system is said to be safe or "deadlock-free." Every successor state obtained this way is safe. Deadlock avoidance is achieved by testing each possible allocation and making only those which lead to safe states. If the process originating this allocation can run to completion and release the resources it holds, then all other processes in the system can be completed, since the state prior to the allocation was safe.

Ignoring deadlocks. The deadlock problem could simply be ignored, and this might be referred to as an "indifferent" or "no strategy" approach. It has a superficial advantage of saving processor time and space otherwise required for detection, prevention, or avoidance methods. However, the onus of recognizing deadlocks is borne by either a computer operator discovering blocked processes or a user waiting for an answer, and ignoring deadlocks has disastrous effects on the consistency of data in any database system.

Approaches compared. Empirical observations have suggested that deadlock prevention mechanisms tend to undercommit resources while detection techniques give away resources so freely that prolonged blocking situations arise frequently. Avoidance schemes fall somewhere in between. However, obtaining good upper bounds is a severe technical difficulty, since otherwise resources are used inefficiently.

In a heavily loaded system, Havender's prevention methods provide very few safe resource allocations and processes become blocked for long periods while holding valuable resources. In contrast, avoidance or prevention mechanisms must ensure that a deadlock will never occur for every request, resulting in undue process waits and run-time overhead. A prevention mechanism differs from an avoidance scheme in that the system need not perform run-time testing of potential allocations. In both prevention and avoidance cases, recovery from a system implementation error needs a rollback mechanism.

Table 1 summarizes the basic deadlock detection, prevention, and avoidance techniques for operating systems, along with their primary merits and deficiencies.

Graph-theoretic models for deadlock detection

In considering the deadlock problem, the representation of process interactions during resource allocation phases is important. Processes in computer systems can be dynamic in the sense that one process may create another. Processes are said to interact *explicitly* when they communicate among themselves. Process interactions resulting from competition for access to physical objects are termed *implicit.* Blocking of processes may be caused by either type of interaction. Holt[8] has proposed the distinction of *reusable* and *consumable* resources to model implicit and explicit interations, respectively. Emphasized is the fact that an arbitrary number of processes can wait for a consumable resource, whereas only a fixed number of reusable units are available.

The physical devices of a computer system, such as tape drives, disks, memory, and I/O channels, are reusable resources. There is always a fixed total number of units of these resources in a system. Any unit of a particular resource can be held by one process at a time. Thus, each unit of a resource is either free for allocation or is held by a process. The allocation strategies specify the unit size of the resource. For

Table 1.
Summary of detection, prevention, and avoidance approaches for operating systems.

PRINCIPLE	RESOURCE ALLOCATION POLICY	DIFFERENT SCHEMES	MAJOR ADVANTAGES	MAJOR DISADVANTAGES
DETECTION	Very liberal; requested resources are granted where possible.	Invoke periodically to test for deadlock.	• Never delays process initiation. • Facilitates on-line handling.	• Inherent preemption losses.
PREVENTION	Conservative; undercommits resources.	Requesting all resources at once.	• Works well for processes that perform a single burst of activity. • No preemption necessary.	• Inefficient. • Delays process initiation.
		Preemption.	• Convenient when applied to resources whose state can be saved and restored easily.	• Preempts more often than necessary. • Subject to cyclic restart.
		Resource ordering.	• Feasible to enforce via compile-time checks. • Needs no run-time computation since problem is solved in system design.	• Preempts without much use. • Disallows incremental resource requests.
AVOIDANCE	Selects midway between that of detection and prevention.	Manipulate to find at least one safe path.	• No preemption necessary.	• Future resource requirements must be known. • Processes can be blocked for long periods.

example, memory may be allocated by pages, disks may be held in units of tracks, cylinders, or entire disks, and data may be assigned as records, fields, entities, or files.

Message text from operators, external interrupts, interprocess communications, add card images produced by a card reader are examples of consumable resources. Typically, the total number of resource units is not fixed. When a consumable resource is acquired by a process, it ceases to exist. A process which creates (produces) a consumable resource may release any number of units at a time and must be treated as if it were holding the resource. Consumable resources are created and released by a producing process and are requested, acquired, and used by other processes. On the other hand, reusable resources are assigned by the resource manager to requesting processes, which eventually return them to the manager.

Many physical resources permit only exclusive use by one process at a time, but others, like data resources and read-only programs, may allow shared use by several processes. Another property of a resource is its ability to allow preemption. Some resources may be taken back by the system without any action by the

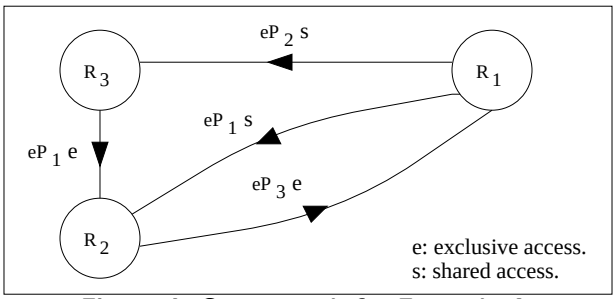

Figure 4. State graph for Example A.

process. Such a process is then either aborted, rolled back (if necessary) and restarted, or forced to rerequest and thus wait for the preempted resource. The cost of aborting or restarting the process accounts for the preemption losses.

In certain cases, it is possible to suspend a process and preempt one of its resources, yet preserve the current states of the process and its use of that resource for later resumption. Typically, such resumption does not lose processing time already spent. For example:

- CPU interrupts, in which the resource preempted is the CPU and the information that must be preserved for later restoration is the status of the process (for example, registers and the "program status word"), and
- swapping, in which the preempted resource is the primary memory and backup is provided in secondary storage.

When the system has different resource types and more than one resource of the same type, the complexity of the deadlock problem increases. Attempts to model and formalize the problem have resulted in two major proposals.[8,9] Graph-theoretic models of the process interactions have been developed, and deadlocks have been expressed precisely in terms of graph topologies.

State graph model. The relationships between a set of processes and a set of distinct resources in use by these processes can be described by a *state graph*.[9] This is a directed graph whose nodes correspond to the resources and whose edges are defined so that if some process P has access to resource R_i and is waiting for access to resource R_j, then there exists an edge directed from node R_i to node R_j.

Example A. Let $\{P_1, P_2, P_3\}$ and $\{R_1, R_2, R_3\}$ be the processes and resources in a system, respectively. Let R_1 be held for shared access by both P_1 and P_2, and let P_3 be waiting for exclusive access to R_1. Assume that R_2 and R_3 are held for exclusive access by P_3 and P_1, respectively, while P_1 and P_2 wait for exclusive access. The process interactions here can be represented by the state graph shown in Figure 4. For clarity and better understanding, we have labeled the edges in the state graph. For instance, the edge directed to R_3 from R_1 is labeled eP_2s indicating that P_2 holds R_1 for shared (s) access and awaits exclusive (e) access to R_3. The existence of a *circuit* in the state graph means that a deadlock exists. A circuit is a closed directed path linking a subset of the nodes in the graph. Thus the path $R_3R_2R_1R_3$ linking three nodes, shows that three processes are involved in deadlock.

In the general case, involving multiple-unit resources, the state graph just defined is inappropriate. Coffman[9] has proposed that the resources be partitioned into different types, in such a way that resources of

a given type are identical. The nodes in the state graph then represent resource types. A directed edge in the graph exists between a node representing one resource type to another, whenever any process has acquired access to at least one unit of one resource type and has requested access to at least one unit of another type. Such a method needs a more elaborate state description mechanism, supplemented with "allocation" and "request" matrices and an "available resources vector." A detection algorithm must be designed which uses these data structures to discover a deadlock by simply investigating every possible allocation sequence for the processes that remain.

General resource graph model. Holt's model[8] of a system of interacting processes provides a versatile representation for resource management. The approach is characterized by the use of a "general resource system" which models reusable as well as consumable resources. A *general resource graph* is defined as a "bipartite graph"[10] whose nodes correspond to the set of processes and the set of resources (reusable and consumable). An *available units vector,* whose elements are positive measures of the quantity available, is associated with the set of resources. Edges directed from a process node to a resource node are termed *request edges.* Edges directed from reusable and consumable resource nodes to processes are called *assignment* and *producer edges,* respectively. A process is blocked if and only if the number of request edges from the process to a particular resource exceeds the number of available units of the resource. A process is deadlocked when it is impossible to get the process out of the blocked state. Holt introduces a *graph reduction* method to check if a process is deadlocked. A graph reduction corresponds to the best sequence of operations a particular process can execute to help unblock other processes. This is achieved by determining if the successive elimination of all sink nodes produces predecessors which are also sinks. A sink node is one with no edges emanating from it (i.e., no wait requests). All nodes will become sinks if and only if no circuits are present in the graph. Holt also develops an algorithm to determine if a particular blocked process is deadlocked. The algorithm systematically traces out all paths emanating from the corresponding process node. A path which leads to a sink exists if and only if the process was not deadlocked.

It can be shown that for a general resource graph in which all processes having requests are blocked, the existence of a circuit is a necessary and sufficient condition for deadlock. A simple illustration of a general resource graph corresponding to Example A is shown in Figure 5. It is evident that the processes P_1, P_2, and P_3 are deadlocked, since each process controls some resource and is requesting exclusive access to at least one resource held by the next waiting process. Thus P_3 waits for P_2 to release R_1, P_2 waits for P_1 to release R_3, and finally P_1 waits for P_3 to release R_2. For the multiple units of resource case, a weight is associated with each allocation edge to represent the number of units held by the resource.

Deadlocks in databases

Many different deadlock-handling algorithms and approaches developed for operating systems have been presented. Database and distributed database systems, however, pose unique problems of their own. The concern is not only with avoiding deadlock situations but also with protecting against database inconsistencies.

The schemes for handling deadlock in operating systems, discussed above, form the basis for database control, but the nature of database access is such that prevention and avoidance methods are less feasible. Basically, too many database applications are data-driven—that is, the next action is based on the value of the current item retrieved. Thus detection methods and associated recovery schemes which select resources for preemption on a basis of their recovery cost become important.[9] These recovery techniques are designed to avoid preemption of resources with high inherent losses.

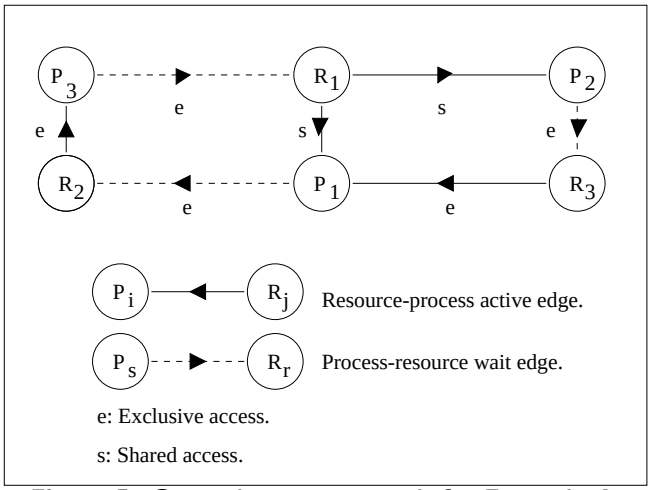

Figure 5. General resource graph for Example A.

Database systems. In database systems, the resources which processes may wish to lock for exclusive use include pieces of shared data. Very often the processes may issue lock requests, such as "Lock the personnel records of all employees in the Systems Programming Dept.," for which the locking criterion depends on data values. The difficulties introduced by this method are normally absent in an operating system environment. In addition, a particular data resource may be described in more than one way, or the nature of a data resource may be altered by a process operating on it. Locks may be interdependent in the sense that further locks may have to be requested depending on the first lock. Because of these difficulties in anticipating future requirements, many conventional approaches to resource management in operating systems are difficult or impossible in the database environment. In the case of preemption of a data resource, the necessity of maintaining uniformly correct data in the system dictates the abortion, rollback, and restarting of one or more processes. This is generally expensive and may complicate the deadlock problem.

Recently, various aspects of concurrent operation of database processes have been topics of active research. Several reports[11−15] have dealt with the deadlock problem in database systems, and some are reviewed here.

It is assumed that a database can be modeled by a graph, with nodes representing collections of information.[11] A directed edge exists between two nodes whenever one node contains the address of the other. Such a database is assumed to be accessed by a set of routines called primitives. Under concurrent operation, two or more primitives may conflict with each other while accessing a node. To overcome these conflicts a procedure called the "walking algorithm" was formulated. It requires a primitive to (1) set the lock of the next node it wishes to access before unlocking the node it is currently accessing and (2) keep a node locked only for the duration of its access. The walking algorithm avoids the possibility of two primitives writing the same node simultaneously. Also, the algorithm requires that a node be unlocked and all its database pointers erased before it can be deleted and returned to the free list. This overcomes a problem that could arise if a primitive erased a pointer from a node immediately after it was read by some other primitive. If such erasure were allowed, subsequent actions by the reading primitive which depend on that pointer might be invalid.

The Codasyl[16] approach to data management uses a LOCK-UNLOCK mechanism to enable incremental

allocation of data resources to processes. The status of all accesses to the database is maintained in an *access state graph*,[12] a directed graph whose nodes correspond to the union of the set of active processes and the set of allocatable data elements. The set of edges is referred to as the *lock list,* and each element represents an edge emanating from the active process node and terminating at the data element allocated to that process. The basic mechanism of the scheme is to model each operation, LOCK, UNLOCK, ALLOCATE, and DEALLOCATE. For instance, a lock action can be represented by mapping the access state graph before a process was allowed to lock a data element onto the access state graph after such allocation was allowed. In essence, the LOCK function adds only a process node to the access state graph since all allocatable data elements are already in the graph. Other functions are modeled. It can be shown that *only* an ALLOCATE function can lead to a deadlock. A major shortcoming in the approach is that a process cannot wait for more than one resource at a time.

One proposed technique[17] modifies and combines the following steps:

(1) try to preclaim needed resources;
(2) preempt data resources when preclaiming leads to a deadlock; and
(3) impose a presequencing scheme for processes by time stamping, to avoid deadlock due to indefinite delay.

The method requires each process to lock all of its data resources during a *seize phase* before starting the *execution phase.* Competition for resources therefore occurs only during seize phases. Backing up a process to the start of this phase is easy, since there are no repercussions from preempting a resource, even though it has been tentatively allocated elsewhere. Once in its execution phase, a process is not allowed to claim additional resources until after it has released *all* the resources it holds. A new seize phase may then follow. To avoid deadlock due to indefinite delay, an age indicator is attached to blocked processes, for use as a priority by the scheduler. Thus the method is deadlock-free.

One-level and two-level lockout mechanisms[14] have also been considered for synchronizing database access. In the one-level lockout scheme, shared access (by readers) to the database is allowed at any time, regardless of the allocation state. Conversely, writers are required to lock the data resources before altering them. As in other schemes, the presence of a circuit in the state graph is a necessary and sufficient condition for deadlock. Schlageter provides an algorithm for detecting deadlock by traversing the graph from a blocked process node, in an attempt to return to that blocked node. In the two-level lockout scheme, readers are split into two classes: those which are insensitive to concurrent updates and those which prevent writers from concurrently accessing the data. This is implemented by using primitives LOCKR (read) and LOCKW (write). Data locked by LOCKR can be accessed by any reader. Data locked by LOCKW can be accessed only by readers which do not need to be protected against changes of data. Under the two-level lockout mechanism, starting at a blocked process node and traversing paths to detect deadlocks is no longer simple, because a resource may be held by several processes simultaneously and each of those may be regarded as blocking any waiting process.

The scheme proposed for deadlock avoidance in database systems[15] requires the processes to pre-declare their anticipated resource requirements, with the system granting only safe requests. The algorithm is tailored to the needs of database systems, unlike other approaches.[5,8] A series of time-varying graph representations are defined for database interactions. A "holds-claims graph" represents only those processes that are currently making a claim on some common system resources. A "claims-claims graph" represents the processes which are potentially capable of denying resources to one another, their claims being in contention. A "holds-holds graph" represents the allocation status of the system. A deadlock exists if and only if there is a circuit in the holds-holds graph, where as a deadlock can be avoided if and only if the holds-claims

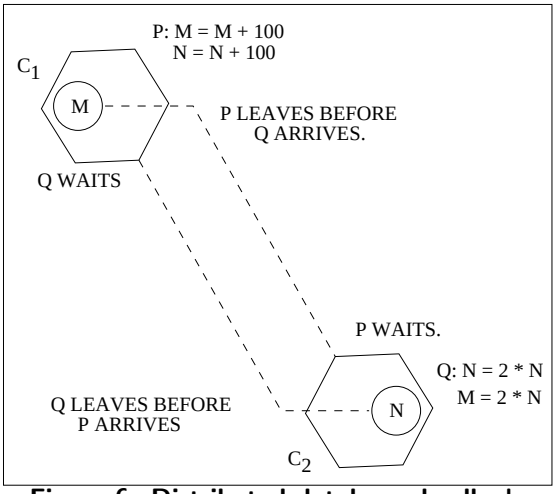

Figure 6. Distributed database deadlock.

graph is circuit-free. A deadlock avoidance scheme has been devised which performs appropriate actions on claims-claims, holds-claims, and holds-holds graphs in the event of process entry/deletion and resource request/release. Even though the possibility of indefinite delay is not entirely eliminated, it is reduced by the strategy of granting requests in an incremental way.

Distributed databases. There is a growing body of work on distributed databases, but early attempts to handle deadlocks had practical drawbacks. The salient features of some methods will be reviewed here, while an on-line approach[17] will be dealt with in the next section.

The main characteristic of distributed computing is that operating systems must contend with more autonomous behavior, thus aggravating the control problem. Presence of appreciable time-lags renders coordination of the various controllers in the system much more difficult. Moreover, in geographically distributed databases all information needed to detect deadlocks is not necessarily available at any single installation, so the deadlock problem is somewhat different. Consider a variation of the example in Figure 2, but with two separate computers C_1 and C_2, as shown in Figure 6. Assume that process P and resource entity M reside at C_1 and that process Q and resource entity N reside at C_2. Processes P and Q, after locking and updating local data resources M and N, arrive at remote locations and get involved in a deadlock which neither computer C_1 nor C_2 can detect, based on the information available at their respective installations.

Communication delays may also make it difficult to get an accurate view of the status of the computer network. As a consequence a new deadlock may not be detected, or a deadlock may be indicated where one no longer exists. Synchronizing the updates of files which are replicated at different sites is also nontrivial, and abortion, rollback, and recovery become very involved and require complex interprocess communication.

Some work on the prevention of deadlocks in distributed databases has been reported. A typical approach requires that all data resources be allocated to the process before initiation. Consequently, process initiation is needlessly delayed. An alternative technique[18] is based on the concept of a *process set,* which is a collection of processes with access to common data resources. A process is allowed to proceed only if all data resources required by the process and the members of its process set are available. In another proposal,[19] each process has to transmit its *shared data resources list* (conceptually similar to a process set) to all other processes

before it can proceed. This shared data resources list is determined by using what is called a process profile, which contains a list of data resources that can be updated by the process. The communication and computation of process sets or shared data resource lists, which are performed continually as processes enter/leave the system, requires substantial system overhead.

Several techniques for deadlock detection in a computer network have been proposed.[20–22] One approach[20] requires each installation to maintain a resource table, which contains information pertaining to

- processes allocated local resources,
- processes waiting for access to local resources,
- local processes allocated remote resources, and
- local processes waiting for access to remote resources.

The type of access requested by the process is also stored. By using such tables, it is hypothesized that well-known algorithms for detecting deadlocks in a single computer system could be extended to detect deadlocks in a network of computers, by communication between installations and by appropriate expansion of resource tables. Schemes to expand resource tables in a network environment are also included.

The *centralized control* approach[21] to deadlock detection in distributed databases creates a picture of the global network status by using file and pretest queues (queues of requests which can only be granted at a future time) received from all other installations in the network. A potential problem with this approach is the selection of suitably sized groups of data to transmit, because if message congestion occurs at the control node the performance of the whole detection scheme degrades. In the distributed control approach,[21] support for N computers requires the transmission of (N - 1) identical messages containing status and queues of files. Also each installation receives (N - 1) different messages from other installations.

Goldman's deadlock detection schemes[22] are based on the creation and expansion of an ordered blocked process list. Each process in an OBPL is waiting for a resource held by the next process in the list. Whenever an OBPL is transmitted between installations, a data resource name is inserted into the identification part of the OBPL. The last process in the list has access to, or is waiting for, that resource. In the former case the state (blocked or active) of the last process in the OBPL must be determined, while in the latter case one needs to know the state of the process which *holds* the data resource. To determine these states, and to eventually detect deadlock, techniques are proposed to expand OBPLs by periodically transmitting them between installations.

While this approach solves many problems, it has several shortcomings. Each process is restricted to having only one outstanding request, which in reality is usually not the case. In addition, when several readers share access to a data resource, the scheme requires the creation and expansion of one different copy of the OBPL for each reader since, if one of the readers is deadlocked, any process which requests access to that resource is blocked forever. It is possible that OBPLs, while undergoing expansion, could be transferred (sequentially) among several installations or several times between the same two installations before a deadlock is detected. Also, OBPLs could become large, leading to substantial overhead, especially when records or entities, instead of files, are considered as data resources.

Other contributions from Goldman's work are examples of deadlocks not detected by the earlier schemes.[20,21] The basic flaw in the other schemes is the assumption that allowing a local process at one site to wait for a local resource will not cause a network-wide deadlock.

In any case, *all* these proposals require the communication of large tables between installations. This in turn has repercussions, since the information may be out of date by the time it arrives. A summary of these techniques for handling deadlock in database systems is given in Table 2, along with an indication of their advantages and costs.

Table 2.
Summary of deadlock-handling techniques in distributed databases.

PRINCIPLE	ADVANTAGES	MAJOR COSTS	RELEVANT REFERENCES
DETECTION	• Facilitates on-line approach. • Suits distributed database environment fairly well.	• Run-time cost. • High rollback and restart costs. • Communication costs in maintaining information for deadlock detection.	• Isloor and Marsland[23] • Marsland and Isloor[17] • Goldman[22] • Mahmoud and Riordon[21] • Chandra, Howe, and Karp[20]
PREVENTION	• Rollback and restart is not necessary. • In the absence of software errors, consistency of database is guaranteed.	• Costs of waiting for highly data-driven processes. • Communication costs in transmitting the advance information on future requests.	• Maryanski[19] • Chu and Ohlmacher[18]
AVOIDANCE	• Minimal use of rollback and restart. • Data resource allocation policy is midway between that of highly conservative prevention and very liberal detection principles.	• Costs of waiting processes and run-time costs. • Moderate rollback cost. • Communication costs of transmitting future data resource allocation requests.	• Chu and Ohlmacher[18]

On-line detection in distributed databases

The concept of on-line detection of deadlocks in a distributed database has been introduced.[17,23] It is defined to be the recognition of deadlock as requests are made or granted, by both local and remote resource allocators, without the necessity of further communication among installations. Existing algorithms[20–22] for deadlock detection in distributed databases, because they were not designed for on-line detection, have several-problems for on-line use:

(1) For every request, granted or not, existing algorithms need to obtain the global network status by simultaneous transmission of each installation's status, leading to heavy communication traffic.
(2) Since this traffic results in communication delays, a new deadlock may go undetected or an apparent deadlock may no longer exist.
(3) After obtaining the complete network status, the algorithms still have to perform deadlock detection computations.

The use of a *reachable set*[8] may overcome these problems to some extent. Thus, in a system graph representing process interactions, the set of all nodes traversed by a directed path from a given node constitutes its reachable set, and a process is deadlocked if and only if the corresponding process node belongs to its own reachable set.[23] Maintaining reachable sets by incremental updates, as resources are allocated and freed

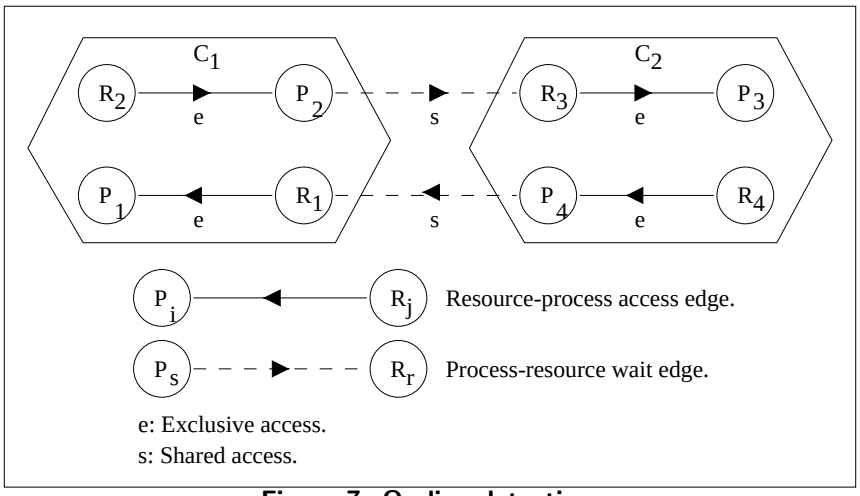

Figure 7. On-line detection.

at each installation, enables local recognition of deadlocks, and allows the allocation decision to be made without further messages.

The on-line approach to deadlock handling eliminates the need for periodic transfer of lengthy messages among the computers in the network to ascertain the deadlock status of the system. Because communication delays are negligible, the method reduces the time interval during which deadlocks go undetected.

Consider, for instance, Figure 7 representing computer systems C_1 and C_2. Let processes $\{P_1, P_2\}$ and resources $\{R_1, R_2\}$ reside on C_1 and similarly processes $\{P_3, P_4\}$ and resources $\{R_3, R_4\}$ reside on C_2. For simplicity assume that each resource R_i is held by process P_i for exclusive access (for all i), that P_4 is waiting for access to R_1, and that P_2 is waiting for R_3. At any instant (say, t_0) both C_1 and C_2 possess updated reachable sets for R_2 and R_4, namely $\{P_2, R_3, P_3\}$ and $\{P_4, R_1, P_1\}$, respectively. Subsequently, let P_1 and P_3 request access to R_2 and R_4, respectively. Deadlock is now inevitable and will be recognized upon the introduction of the process-resource wait edges from P_3 to R_4 and P_1 to R_2. At the instant t_1, neither computer detects trouble, but when C_1 and C_2 receive the information on the addition of the edges at each other's installation, the deadlock is detected at both sites as the reachable sets are updated. The essence of this example is that requests by P_1 for R_2 and P_3 for R_4 occur within the same time frame. Thus, no matter what detection is used, the allocation will occur because neither computer can be aware of the activities of the other. Under these circumstances, a preemption mechanism is *necessary* in order to recover control of the system. It was just such a situation as this which allowed Goldman to deduce that other methods[20,21] for deadlock detection in distributed systems were flawed.

The motivation for on-line detection is a consequence of several factors. As large integrated databases respond to more users, the possibility of deadlocks increases rapidly. Furthermore, the arrival of distributed processing on the scene enhances the need for immediate deadlock detection. In a distributed environment, any delay in the detection of deadlocks has adverse effects on database consistency.

Several aspects of on-line detection may be considered for the following complete set of process-resource interactions:

(1) a new process enters the system;
(2) a new data resource is accessed;

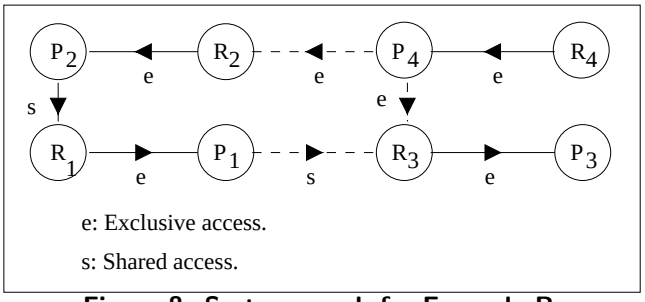

Figure 8. System graph for Example B.

(3) a process runs to completion and releases all the data resources held;
(4) a process in the system requests access to a data resource held by another process; and
(5) a data resource held by a process is preempted from it.

In cases (1) and (2), a deadlock-free system remains deadlock-free. For case (3), allocation decisions for released data resources without any waiting-access requests, or with a single waiting-access request, do not lead to deadlock. However, an allocation decision for a released data resource with multiple waiting-access requests can lead to potential deadlocks as shown in Example B. Even so, at least one safe allocation exists.[17]

Example B. Consider processes P_1, P_2, P_3, and P_4 which hold data resources R_1, R_2, R_3, and R_4, respectively, for exclusive use. Let us assume that P_1 is waiting for shared access to R_3, P_2 for R_1, and P_4 for exclusive use of R_2 and R_3. Suppose that process P_3 runs to completion and releases resource R_3. R_3 has two waiting-access requests from P_4 and P_1. Assume that P_4 issued its request before P_1 did. If the allocation of R_3 to P_4 is done in a FIFO manner, then processes P_1, P_2, and P_4 will be deadlocked. It is obviously more advantageous to make the allocation of R_3 to P_1, and let P_1 run to completion, than to make the allocation of R_3 to P_4 and be deadlocked. A necessary and sufficient condition, based on reachable sets, has been derived[23] to recognize situations in which processes have more than one outstanding request, and to avoid deadlock accordingly. In Figure 8 it is process P_4 which has waiting-access requests for both R_2 and R_3.

Marsland[17] shows that, in the case of a deadlock-free system with multiple processes waiting for access to a released data resource, there exists at least one process such that an allocation made to this process maintains the system deadlock-free. Further improvement may be possible by allocating the resource to a safe process which has minimum waiting-access requests on other data resources. In batch operating systems, a process is typically allowed to have more than one outstanding request at a time; however, if the system is unable to satisfy *all* the outstanding requests *at once,* the process is required to drop them.[8] This in turn rules out the occurrence of a situation analogous to that described in Example B. Consequently, early researchers of the deadlock problem in databases also disallowed multiple outstanding requests. Since database requests are data-driven and content-based, the possibility of multiple outstanding requests is high, and so the on-line approach is useful.

In case (4), honoring the request may lead to a deadlock, while in case (5) an attempt is being made to break deadlock through preemption of a resource. For all these cases algorithms are available for updating the reachable sets for every edge added or deleted in the system graph.[17,23]

Besides its low level of communication activity, the on-line approach has the following major advantages:

- A process is allowed to have any number of outstanding requests.
- When a number of readers share a data resource, no special treatment is needed, unlike schemes which require that a different copy of the OBPL be formed for each such reader.[22]
- Every request is dealt with in a uniform way, since the detection technique does not classify requests according to the relationship between the origin of the process and the addressed data resource.

In all other approaches[20–22] the algorithms deal with each access request according to some classification.

Combined approach to deadlock handling

Howard[24] hypothesizes that none of the three basic approaches alone—detection, avoidance, or prevention—is appropriate for the entire spectrum of resource allocation problems encountered in operating systems. Instead, individual techniques can be tailored to optimally handle subproblems of resource allocation and still operate globally to prevent deadlocks. The basis for a mixed technique lies in the structure of operating systems. Resource ordering in a hierarchical structure provides the framework. In some operating systems this hierarchy is in the form of *enveloping layers* of capability. Each layer modifies and extends the facilities provided by an inner layer. A primitive function or operation at one level is implemented by the creation of an inner-level process, which performs necessary actions and returns a result. Thus, an outer process must wait for a message, signaling the completion of the inner process. Such messages are treated as consumable resources by deadlock-handling mechanisms. In the case of resource ordering, this implies that an outer process cannot hold any resources required by the inner process it creates. Such a restriction automatically enforces a "natural" ordering of resources, since resources required by inner-level processes would automatically appear later in the ordering.

By way of illustration, consider the following example involving a typical multibatch computing system. The resources are classified as follows[24]:

(1) space in the swapping store;
(2) assignable devices such as tape drives, and job resources such as access to files;
(3) central memory for user jobs; and
(4) internal resources such as memory for transient system overlays, and access to channels and controllers.

In Howard's ideal mixed-method solution to resource management, the resources are organized into the four ordered classes above, and a different scheme is used for each class. Thus, preallocation may be used for the swapping space, since the maximum storage requirements are usually known. Since a job can always be swapped out, central memory can be preempted, so deadlock prevention is feasible. For job resources, much information about the intended requirements can be obtained from the job control cards, so deadlock avoidance can be used. In principle, deadlock detection is also possible, since resource ordering can be used, but problems may arise if file preemption is attempted during an update. Finally, for internal system resources, prevention through resource ordering is possible, since run-time choices between pending requests are unnecessary. As Howard concludes, "The practical advantages of the combined approach in operating systems will outweigh the theoretical desirability of using a single method throughout."[24]

A comprehensive combined approach to deadlock handling in database systems or distributed databases has not been devised so far. However, the idea has been applied for on-line detection in distributed databases, where there may be multiple outstanding requests on a data resource released by a completing process. In

	D	U	R	O
DESTROY	1	1	1	1
UPDATE	1	1	1	0
READ	1	1	0	0
OPEN	1	0	0	0

Figure 9. Blocking matrix for file operations.

the case of a deadlock-free system, an unwise resource allocation decision can potentially lead to deadlock as depicted in Example B. But, as shown, at least one process exists (in the set of waiting processes) to which the resource can be allocated that maintains the system deadlock-free. In other words, a potential deadlock is detected and avoided accordingly. The approach uses both detection and avoidance principles. Thus, in the case of multiple processes waiting for access to a released data resource, the resource is allocated to the first process which maintains the system deadlock-free after the allocation. Further improvement may be possible by allocating the resource to the first process which not only maintains the system deadlock-free but also has minimum waiting-access requests on other data resources.

Some current deadlock-handling implementations

In modern timesharing computer systems, the potential for deadlock during access to the file system is high. Consider the Michigan Terminal System,[25] an operating system under continuous development by several major universities in various countries. Before any specific file operation is performed, the files are locked in one of three inclusive levels (read, update, or destroy).[26] In order to ensure that the rules of concurrent usage are not violated before locking, MTS maintains a table indicating, at any instant,

- all the files currently open and/or locked,
- how they are locked and by what *task* (process),
- what tasks are currently waiting to lock a file and why they are waiting.

From this table, one can determine whether or not a particular type of opening and/or locking of a file can be allowed, according to the following rules of concurrent usage.[26]

(1) If a file is not locked for updating or destroying, any number of tasks can have shared (read) access to this file.
(2) If a file is not locked for reading or destroying, then only one task can have this file locked for updating (writing, emptying, or truncating).
(3) If a file is not being used, then only one task can have this file locked for destroying (or renaming or permitting).

If a file cannot be locked as requested, the task is queued to wait for the file, but waiting can lead to deadlock due to mutual blocking of tasks. As defined by the matrix in Figure 9, a process wishing to update a file is blocked by processes wanting to destroy, update, or read it but not by a process which has simply opened the file. Similarly, opening a file is only blocked by a process in the act of destroying the file.

MTS maintains the information about file usage in an $M \times M$ matrix, B. Each element $B_{ij} = 1$ if and only if TASK_i has a file open or locked in such a way that it blocks TASK_j. Note that M is the number of

	P_1	P_2	P_3	P_4														
P_1	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0	0		
P_2	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0		
P_3	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1	0		
P_4	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1		
B^* =	B				+	BB			+	BBB			+	$BBBB$				

Figure 10. Computation of transitive closure for example shown in Figure 7.

tasks blocking or being blocked by another task. Usually this is far fewer than the number of active tasks or the number of open files. The deadlock situation is determined from the status of matrix B^*, the transitive closure[10] of B. In essence B^* is obtained by ORing the rows of B with each other. Thus for a 3×3 matrix, $B^* = B + BB + BBB$. Consequently, if $B^*_{ii} = 1$ then TASK$_i$ is involved in a deadlock. For the example shown in Figure 7, after P_1 and P_3 have been made to wait, the existence of deadlock is provided by the computation of B^* from B, as shown in Figure 10.

Data management systems, on the other hand, vary in the way they detect and resolve a deadlock. As others have pointed out, many of the early systems implemented techniques which were quite rudimentary.[27] In products such as IDMS—Integrated Database Management System—Total, and Model 204, deadlock is not possible because of restrictions on processes. In these systems, since a process can lock only one record at a time, heavy restrictions are placed on users to ensure that deadlocks never occur.[27] Consequently, they are neither general nor realistic and exhibit low levels of concurrent data resource usage. In Adabas—Adaptable Data Base System[27]—on the other hand, if a process requests a locked record five times, the record is unlocked and control transferred to the requester. Thus it is possible for a process to lose control of a record after it has locked it, necessitating a reacquisition before the update is repeated. This method leads to unnecessary preemptions and is also subject to cyclic restart, as illustrated in Figure 3.

Advanced models for deadlock analysis

Although the basic models for analyzing the deadlock situation in computer systems have been reviewed, many other aspects have not been covered. In particular, we have not mentioned probabilistic or game-playing models which attempt to avoid deadlock by ensuring, with high likelihood, that every state in the system is secure. A brief treatment of these new and interesting methods follows.

A game-playing model. The game-playing model of the deadlock situation has not received much attention, perhaps because it appears to be practical only for small systems. Nevertheless it does provide a valuable alternative viewpoint. In general, the approach recognizes that the resource allocation problem is effectively a zero sum game in which the resource manager, RM, is competing against independent processes demanding service.[28] In this case, the manager may be thought of as winning if all the processes complete successfully; the opposition wins if it creates a deadlock. Devillers[28] informally defines deadlock avoidance: "From some *a priori* information about the processes, the resources, the operating system, etc., determine what situations may be realized without endangering the smooth running of the system."

He also presents a flowchart model which in some sense generalizes the maximum-claims[7] and task-step[9] methods. Unfortunately, with this model it is not easy to define the safe states, not only because a process

may have more than one possible series of steps that it may traverse (future history), but also because there may not exist a simple worst possible future history for each process.

To overcome this problem Devillers proposes a global approach, in which a state is defined *safe* if and only if a strategy exists for the RM which ensures its success whatever operation the processes in that state choose. A state will be *losing* if an operation exists for the processes such that the RM will lose the game whatever strategy it chooses. This approach throws new light on the deadlock problem by providing a way to construct the set of unsafe states, and hence providing a basis for a systematic study of the properties of the safe states.

Probabilistic model. A probabilistic approach to the deadlock problem has also been tried,[29] and an investigation made of the likelihood of deadlock in certain classes of systems. Any state diagram used to represent process-resource interactions can also be viewed as a finite state automaton. A probability measure can be attached to an occurrence of each possible transition. A random resource allocation model is assumed, and the sum of the probabilities associated with transitions out of a given state is required to be unity. By adding auxiliary storage to the automaton, first-come-first-serve (FCFS) and last-come-first-serve (LCFS) schedulers can be modeled to form, respectively, a probabilistic queue automaton and a probabilistic push-down automaton.[29] The likelihood of deadlock is measured in terms of the expected value of (1) the number of system actions to deadlock or (2) the number of resource allocations to deadlock.

Calculations are carried out for systems containing small numbers of processes and resources. For a system with two resources and two processes the mean time to deadlock under FCFS or LCFS scheduling is shown to be slightly less than that under random allocation. The fact that one would expect the probability of deadlock to decrease if the number of units of the resources increases (while the number of processes remains fixed) is substantiated. Conversely, in a fixed-resource system, increasing the number of processes increases the probability of deadlock since more processes compete for the same number of resources. However, it is not intuitively clear what happens to the deadlock probability if the number of processes and resources are uniformly increased. For small systems it has been shown that the probability of deadlock actually increases.[29] However, since the model considered no more than five resources and processes, which is by no means very many in the commercial world, further research is warranted.

Directions for research

A comprehensive probabilistic model for computer deadlocks of large systems has not yet appeared in the literature, and an extension to systems with consumable resources is also needed. Further, in concurrent database accesses, far too little is understood about the relative probability of interference and deadlock. For transaction processing systems, it is thought that interference is rare and that elaborate avoidance algorithms would not be economical.[30]

Because of the increasing complexity of distributed databases, the deadlock situation must be handled efficiently. However, it is difficult to estimate the performance effects of these techniques, or the probability of occurrence of deadlocks, since communication time is critical. As distributed databases become more widely available, experimental data can be gathered to measure performances and probabilities, permitting exploration of the communication aspects.

A combined approach to handling deadlocks in database systems is probably the most practical solution, but there may be further problems in integrated database systems which allow processes access to classified data. Additional levels of locking may be necessary, which at the very least will increase the locking complexity.

Research is necessary to determine an efficient and effective method of rolling back a process. Such a mechanism can make existing deadlock detection techniques[12,14,17,22] much more attractive. With the present-day trend toward increased concurrent access, a deadlock detection method outweighs prevention schemes in distributed systems by enhancing concurrency.

As more and more data is integrated over a network of computers, resulting in the databases becoming more accessible to larger numbers of diverse application jobs, the database administrator's function becomes increasingly complex. The actions of the operating system (which manages application jobs) and those of the DBA (who maintains process integrity and the consistency of the database) have to be coordinated. Both the DBA and the operating system designer must thoroughly understand relationships among concurrency controls, processors, processes, deadlock-handling and recovery techniques, communication aspects, and protocols. This area of responsibility between the system designer and the DBA calls for deeper study, especially in a network environment.

Acknowledgments

This research was supported in part by a grant from the National Science and Engineering Council of Canada.

The efforts of all CMPUT 516 students who carefully read and commented on earlier versions of this article are much appreciated, but special thanks are due Terry Crocker, Murray Reid, and Arnold Rivera for their constructive criticisms and insights.

References

1. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM,* Vol 19, No. 7, July 1976, pp. 395–404.

2. S. E. Madnick and J. J. Donovan, *Operating Systems,* McGraw-Hill, New York, 1974, pp. 255–261.

3. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Data Base System," *Comm. ACM,* Vol. 19, No. 11, Nov. 1976, pp. 624–633.

The significant differences between locking protocols in a shared database system and those in an operating system are brought out. The concepts of transaction and consistency are defined. It is shown that maintaining consistency implies that a transaction cannot request new locks until after it has released all existing locks.

4. J. N. Gray, "Notes on Data Base Operating Systems," in "Operating Systems (An Advanced Course)" in *Lecture Notes in Computer Science 60,* R. Bayer et al., eds., 1978,PP. 393–481.

This paper is a conglomeration of research results from IBM authors engaged in the development of the relational database management system, System R. There is a thorough discussion on data management, data communications, transaction scheduling, consistency and concurrency aspects, lock management, and recovery aspects. The approach is intuitive and tutorial. This is an excellent source for obtaining a bird's-eye view of a database-oriented operating system.

5. J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Systems J.,* Vol. 7, No. 2, 1968, pp. 74–84.

The approaches suggested eliminate the possibility of deadlocks. The strategies used include (a) requesting all required resources at once; (b) preempting and rerequesting resources, in the case of a denial of incremental request for resources; and (c) resource ordering.

6. P. Brinch Hansen, *Operating System Principles,* Prentice-Hall, Englewood Cliffs, N. J., 1973, pp. 42–49, 124–125.

7. A. N. Habermann, "Prevention of System Deadlocks," *Comm. ACM,* Vol. 12, No. 7, July 1969, pp. 373–377, 385.

The approach is a practical model and is a good example of avoidance. It uses the "maximum claims strategy" with regard to information on future resource requirements of each process. Deadlock avoidance is achieved by testing each possible allocation and making only those which lead to safe states.

8. R. C. Holt, "Some Deadlock Properties of Computer Systems," *Computing Surveys,* Vol. 4, No. 3, Sept. 1972, pp. 179–196.

A thorough and comprehensive model for systems of interacting processes is presented. Various aspects of deadlocks in reusable-, consumable-, and general-resource systems are developed. Good examples describe every new concept and several existing problems. The technique of graph reductions is developed and used in designing efficient algorithms for deadlock detection and prevention. The paper is clear and is an extremely good source for teaching deadlock aspects in operating systems.

9. E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys,* Vol. 3, No. 2, June 1971, pp. 67–78.

Representation of process interactions by a state graph is proposed. A more elaborate state description mechanism is suggested for representing interactions in the case of more than one resource of a type. Using such a description, an efficient deadlock detection and recovery technique is designed. The recovery technique facilitates the inclusion of preemptible resource types of varying preemption costs in the notion of deadlock. Several theoretical aspects of deadlock avoidance are presented informally.

10. N. Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall, Englewood Cliffs, N. J., 1974.

11. A. Shoshani and A. J. Bernstein, "Synchronization in a Parallel Accessed Data Base," *Comm. ACM,* Vol. 12, No.11, Nov. 1969, pp. 604–607.

The concurrent operation of database processes is considered. Synchronization rules to maximize the amount of parallel activity are formulated. A database is represented by a graph, and algorithms are developed to overcome conflicts and avoid deadlock while concurrent access at the same node takes place.

12. P. F. King and A. J. Collmeyer, "Database Sharing: An Efficient Mechanism for Supporting Concurrent Processes," *Proc. 1973 NCC,* Vol. 42, AFIPS Press, Montvale, N. J., June 1973, pp. 271–275.

The LOCK-UNLOCK mechanism of the Codasyl approach to data management, which enables incremental allocations, is described. A necessary and sufficient condition for the existence of deadlock in terms of the effect of the ALLOCATE function is derived. Using this a deadlock detection scheme is devised. A recovery technique in the event of a deadlock is suggested.

13. D. D. Chamberlin, R. F. Boyce, and I. L. Traiger, "A Deadlock-Free Scheme for Resource Locking in a Data-Base Environment," Information Processing 74, *Proc. IFIP Congress,* North-Holland Publishing Co., Amsterdam, Aug. 1974, pp. 340–343.

The technique is a very shrewd modification and combination of (a) try to preclaim needed resources; (b) if preclaiming resources leads to a deadlock, preempt resources; and (c) impose a presequencing mechanism for processes by time stamping, to avoid deadlock due to indefinite delay.

14. G. Schlageter, "Access Synchronization and Deadlock Analysis in Database Systems: An Implementation-Oriented Approach," *Information Systems,* Vol. 1, No.2, 1975, pp. 97–102.

One-level lockout and two-level lockout mechanisms for access synchronization are discussed. Given a process graph, deadlock detection schemes for both lockout mechanisms are to start at the blocked process node and test if a path returns to the process node.

15. D. B. Lomet, "A Practical Deadlock Avoidance Algorithm for Data Base Systems," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data,* Toronto, Canada, Aug. 1977, pp. 122–127.

The deadlock avoidance scheme proposed here requires the processes to pre-declare their anticipated resource requirements. A series of graph representations for database interactions is developed. Necessary and sufficient conditions for the existence of a deadlock and the avoidance of deadlock are derived. A deadlock avoidance scheme in the event of a process entry/deletion and resource request/release is devised.

16. CODASYL Data Base Task Group Report, Conf. on Data System Languages, ACM, New York, Apr. 1971.

17. T. A. Marsland and S. S. Isloor, "Detection of Deadlocks in Distributed Database Systems," *INFOR,* Vol. 18, No. 1, Feb. 1980, pp. 1–20.

A discussion of deadlock-handling techniques in distributed systems is provided. A tutorial approach to the on-line detection method is taken. Examples from real world applications are depicted to show the reality of a process having multiple outstanding resource requests. For this case, the combined approach of deadlock detection and avoidance is suggested.

18. W. W. Chu and G. Ohlmacher, "Avoiding Deadlock in Distributed Data Bases," *Proc. ACM Nat'l Conf.,* Nov. 1974, pp. 156–160.

Deadlock prevention techniques for distributed data-bases are provided. The first approach requires the allocation of all needed resources before process initiation. The concept of a process set is introduced and used in the second approach proposed for deadlock prevention.

19. F. J. Maryanski, "A Deadlock Prevention Algorithm for Distributed Data Base Management Systems," Technical Report CS 77-02, Computer Science Dept., Kansas State University, Manhattan, Kan., Feb. 1977, 24 pp.

Prevention algorithm requires each process to communicate its shared data resources list to all other processes before it can proceed. The shared data resources list is determined by using process profile. Proof of correctness of the algorithm is sketched.

20. A. N. Chandra, W. G. Howe, and D. P. Karp, "Communication Protocol for Deadlock Detection in Computer Networks," *IBM Technical Disclosure Bulletin,* Vol. 16, No. 10, Mar. 1974, pp. 3471–3481.

The technique of deadlock detection requires the maintenance of resource tables at each installation, containing information on processes (local/remote) having access to resources (local/remote). Schemes for the expansion of resource tables are given in the network environment.

21. S. A. Mahmoud and J. S. Riordon, "Software Controlled Access to Distributed Data Bases," *INFOR,* Vol. 15, No. 1, Feb. 1977, pp. 22–36.

Two approaches–centralized control and distributed control–to deadlock detection are suggested. The centralized approach detects deadlocks by creating an overall global picture of the network status. In the distributed-control approach each computer sends identical messages to every other one, and receives different messages from each one, so that a deadlock may be detected at any particular installation.

22. B. Goldman, "Deadlock Detection in Computer Networks," Technical Report MIT/LCS/TR-185, Laboratory for Computer Science, MIT, Cambridge, Mass., Sept. 1977, 180 pp.

The scheme is based on the creation and expansion of what is called an ordered blocked process list. Techniques to expand OBPLs between installations are provided to detect deadlocks in the network environment. Formal proof of correctness of the scheme is provided. Failure of two published algorithms is demonstrated.

23. S. S. Isloor and T. A. Marsland, "An Effective 'On-Line' Deadlock Detection Technique for Distributed Data Base Management Systems," *Proc. COMPSAC 78,* Chicago, Ill.,Nov. 1978, pp. 283 – 288.*

The applicability of deadlock detection principles to distributed data bases is investigated. The concept of on-line detection of deadlocks in distributed systems is defined and introduced, and an effective algorithm for on-line detection is suggested. Several highlights of the proposal in a distributed environment are emphasized. Further research in the area is identified.

24. J. H. Howard, Jr., "Mixed Solutions for the Deadlock Problem," *Comm. ACM,* Vol. 16, No. 7, July 1973, pp.427–430.

A thorough discussion on the inappropriateness of the applicability of detection, avoidance, or prevention principles alone for the entire spectrum of resource allocation problems in operating systems is provided. A mixed solution combining the basic principles and allowing the selection of the optimal one for each class of resources in a system is suggested. Within the framework of resource ordering, the method closely corresponds to the hierarchical structure of operating systems.

25. D. W. Boettner and M. T. Alexander, "The Michigan Terminal System," *Proc. IEEE,* Vol. 63, No. 6, June 1975, pp. 912–918.

26. G. C. Pirkola, "A File System for a General Purpose Timesharing Environment," *Proc. IEEE,* Vol. 63, No. 6, June 1975, pp. 918–924.

27. D. C. Tsichritzis and F. H. Lochovsky, *Data Base Management Systems,* Academic Press, New York, 1977.

28. R. Devillers, "Game Interpretation of the Deadlock Avoidance Problem," *Comm. ACM,* Vol.20, No. 10, Oct.1977, pp. 741–745.

The deadlock avoidance problem is defined as determining safe situations which may be realized without endangering the smooth running of the system, from some causal information about the processes, resources, and operating system. A global approach to the deadlock phenomenon is taken, and the evolution of the system is interpreted as a game between the operating system and "nature" represented by the processes. The notion of risk and safety are precisely defined, and can be used to study the properties of the safe states.

29. C. A. Ellis, "Probabilistic Models of Computer Deadlock," Report CU-CS-041-74, Dept. of Computer Science University of Colorado, Boulder, Colo., Apr. 1974, 25 pp.

The probability of increase or decrease in deadlocks, as the number of processes and resources within a computer system increases, is investigated. A model applicable to the investigation of this problem is

presented. Basically the process-resource interactions are treated as potential members of the set of strings accepted by a probabilistic automaton. Results of calculations on actual computer system models are also described, indicating that the probability of deadlock increases for the types of systems considered.

30. R. Peebles and E. Manning, "System Architecture for Distributed Data Management," *Computer,* Vol. 11, No.1, Jan. 1978, pp. 40–47.

Bibliography

Bayer, R., "On the Integrity of Data Bases and Resource Locking," in *Lecture Notes in Computer Science 39, Proc. 5th Informatik Symp.,* H. Hasselmeir and W. G. Spruth, eds., Germany, Sept. 1975, pp. 339–361.

Bayer, R., "Integrity, Concurrency, and Recovery in Databases," in *Lecture Notes in Computer Science 44, Proc. ECI Conf.,* K. Samelson, ed., F. R. Germany, 1976, pp. 79–106.

Bernstein, P. A., J. B. Rothnie, Jr., N. Goodman, and C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Trans. Software Engineering,* Vol. SE-4, No.3, May 1978, pp. 154–168.

Bjork, L. A., "Recovery Semantics for a DB/DC System," *Proc. ACM Nat'l Conf.,* 1973, pp. 142–146.

Braude, E. J., "An Algorithm for the Detection of System Deadlocks," IBM Technical Report TR00.791, IBM Data Systems Division, Poughkeepsie, N. Y., 1968.

Coffman, E. G., Jr., and P. J. Denning, *Operating Systems Theory,* Prentice-Hall, Englewood Cliffs, N. J., 1973 pp. 31–82.

Collier, W. W., "System Deadlocks," IBM Technical Report TR-00.1756, IBM Systems Development Division, Poughkeepsie, N. Y., 1968.

Collmeyer, A. J., "Database Management in a Multi-Access Environment," *Computer,* Vol. 4, No. 6, Nov.-Dec. 1971, pp. 36–46.

Curtice, R. M., "Integrity in Data Base Systems," *Datamation,* May 1977, pp. 64–68.

Date, C. J., "Integrity" (Chapter 20), *An Introduction to Database Systems,* Addison-Wesley, Reading, Mass., 1975, pp. 301–316.

Davies, C. T., "Recovery Semantics for a DB/DC System," *Proc. ACM Nat'l Conf.,* 1973, pp. 136–141.

Deppe, M. E., and J. P. Fry, "Distributed Data Bases: A Summary of Research," *Computer Networks,* Vol. 1, No. 2, Sept. 1976, pp. 130–138.

Ellis, C. A., "On the Probability of Deadlock in Computer Systems," *4th Symp. Operating System Principles,* Oct. 1973.

Eswaran, K. P., and D. D. Chamberlin, "Functional Specifications of a Subsystem for Database Integrity," *Proc. Int'l Conf. Very Large Data Bases,* Framingham, Mass., Sept. 1975, pp.44–68.

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

Florentin, J. J., "Consistency Auditing of Databases," *Computer J.,* Vol. 17, No. 1, 1974, pp. 52–58.

Fontao, R. O., "A Concurrent Algorithm for Avoiding Deadlocks in Multiprocess Multiple Resource Systems," *Proc. 3rd Symp. Operating System Principles,* Oct. 1971.

Fossum, B. M., "Data Base Integrity as Provided for by a Particular Data Base Management System," in *Database Management,* J. W. Klimbie and K. L. Koffeman, eds., North-Holland, Amsterdam, 1974, pp. 271–288.

Frailey, D. J., "A Practical Approach to Managing Resources and Avoiding Deadlocks," *Comm. ACM,* Vol. 16, No. 5, May 1973, pp. 323–329.

Fry, J. P., and E. H. Sibley, "Evolution of Data-Base Management Systems," *Computing Surveys,* Vol. 8, No. 1, Mar. 1976, pp. 7–42.

Gardarin, G., and S. Spaccapietra, "Integrity of Databases: A General Lockout Algorithm with Deadlock Avoidance," in *Modeling in Data Base Management Systems,*G. M. Nijssen, ed., North-Holland, Amsterdam, 1976, pp. 395–411.

Gold, E. M., "Deadlock Prediction: Easy and Difficult Cases," SIAM J. *Computing,* Vol 7, No. 3, Aug. 1978, pp. 320–336.

Goldstein, B. C., "On the Resolution of Deadlocks," IBM Technical Report TR00.2176-1, Poughkeepsie, N.Y., 1973.

Gouda, M. G., "A Hierarchical Controller for Concurrent Accessing of Distributed Databases," *Proc. Fourth Workshop Computer Architecture for Non-Numeric Processing,* Syracuse University, N. Y., Aug. 1978, pp. 65–70.

Gray, J. N., "Locking," *Concurrent Systems and Parallel Computation,* Conf. Record, J. B. Dennis, ed., 1969.

Gray, J. N., "Locking in a Decentralized Computer System," IBM Research Report RJ 1346, IBM Research Laboratory, San Jose, Calif., Feb. 1974.

Gray, J. N., R. A. Lorie, and G. R. Putzolu, "Granularity of Locks in a Shared Data Base," *Proc. Int'l Conf. Very Large Data Bases,* Framingham, Mass., Sept. 1975, pp. 428–451.

Gray, J. N., R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modeling in Data Base Management Systems,* G. M. Nijssen, ed., North-Holland, Amsterdam, 1976, pp. 365–394.

Habermann, A. N., *Introduction to Operating System Design,* Science Research Associates, Chicago, Ill., 1976, pp. 80-85, 325–336.

Hawley, D. A., J. S. Knowles, and E. E. Tozer, "Database Consistency and the CODASYL DBTG Proposals," *The Computer Journal,* Vol. 18, No. 3, 1975, pp. 206–212.

Hebalkar, P. G., "Deadlock-Free Resource Sharing in Asynchronous Systems," PhD dissertation, Electrical Engineering Dept., MIT, Cambridge, Mass., Sept. 1970.

Herschberg, I. S., and J. C. A. Boekhorst, "Concurrent File Access Under Unpredictability," *Information Processing Letters,* Vol 6, No. 6, Dec. 1977, pp. 203–208.

Holt, R. C., "On Deadlock in Computer Systems," PhD dissertation, Dept. of Computer Science, Cornell University, Ithaca, N. Y., Jan. 1971.

Holt, R. C., "Comments on Prevention of System Deadlocks," *Comm. ACM,* Vol. 14, No. 1, Jan. 1971, pp. 36–38.

Hsiao, D. K., *Systems Programming, Concepts of Operating and Data Base Systems,* Addison-Wesley, Reading, Mass., 1975, pp. 166–171.

Iazeolla, G. G., "Deadlock-Free Sequencing for Multiple Resources Allocation in Multiprocess Systems: Optimization Perspectives," Report No. 1-02, Universita di Roma, Istituto di Automatica, Rome, Italy, 1971.

Kameda, T., "A Polynomial-Time Test for the Deadlock-Freedom of Computer Systems," in *Lecture Notes in Computer Science 44, Theoretical Computer Science 3rd, GI Conf.,* G. Goos and J. Hartmanis, eds., F. R. Germany, Mar. 1977, pp. 282–291.

Lomet, D. B., "Subsystems of Processes for the Prevention of Indefinite Delay with Deadlock Avoidance," IBM Research Report RC 6897, IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y., Dec. 1977.

Lomet, D. B., "Multi-Level Locking with Deadlock Avoidance," IBM Research Report RC 7019, IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y., Mar. 1978.

Macri, P. P., "Deadlock Detection and Resolution in a CODASYL Based Data Management System," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* Washington, D.C., June 1976, pp. 45–49.

Mahmoud, S. A., "Resource Allocation and File Access Control in Distributed Information Networks," PhD thesis, Electrical Engineering Dept., Carleton University, Ottawa, Canada, Jan. 1975.

Mahmoud, S. A., and J. S. Riordon, "Protocol Considerations for Software Controlled Access Methods in Distributed Data Bases," *Proc. Int'l Symp. Computer Performance Modeling, Measurement, and Evaluation,* Cambridge, Mass., Mar. 29-31, 1976, pp. 241–264.

Maryanski, F. J., "A Survey of Developments in Distributed Data Base Management Systems," *Computer,* Vol. 11, No. 2, Feb. 1978, pp. 28–38.

Miller, T. J., "Deadlock in Distributed Computer Networks," UIUCDCS-R-74-619, Dept. of Computer Science, University of Illinois, Urbana, Ill., 1974.

Munz, R., and G. Krenz, "Concurrency in Database Systems–A Simulation Study," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* Toronto, Canada, Aug. 1977.

Murphy, J. E., "Resource Allocation with Interlock Detection in a Multi-Task System," *Proc. 1968 FJCC,* Vol. 33, AFIPS Press, Montvale, N. J., pp. 1169–1176.

Nutt, G. J., "Some Applications of Finite State Automata Theory to the Deadlock Problem," Report CU-CS-017-73, University of Colorado, Boulder, Colo., Apr. 1973.

Papadimitriou, C. A., P. A. Bernstein, and J. B. Rothnie, "Some Computational Problems Related to Database Concurrency Control," *Proc. Conf. Theoretical Computer Science,* University of Waterloo, Waterloo, Ontario, Canada, Aug. 1977.

Parnas, D. L., and A. N. Habermann, "Comment Deadlock Prevention Method," *Comm. ACM,* Vol 15, No. 9, Sept. 1972, pp. 840.

Ramsperger, N., "Concurrent Access to Data," *Acta Informatica,* Vol. 8, 1977, pp. 325–334.

Reiter, A., "A Resource Allocation Scheme for Multi-User On-Line Operation of a Small Computer," *Proc. 1967 SJCC,* AFIPS Press, Montvale, N. J., pp. 1–7.

Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Systems,* Vol. 3, No. 2, June 1978, pp. 178–198.

Russell, R. D., "A Model of Deadlock-Free Resource Allocation," PhD thesis, Dept. of Computer Science, Stanford University, Stanford, Calif., July 1971.

Schlageter, G., "The Problem of Lock by Value in Large Data Bases," *The Computer Journal,* Vol. 19, No. 1, 1976, pp. 17–20.

Schlageter, G., "Locking Protocols in Distributed Databases," *Proc. Int'l Conf. Data Base Management Systems, ICMOD 78,* Milano, Italy, June 1978, pp. 261–276.

Schlageter, G., "Process Synchronization in Database Systems," *ACM Trans. Database Systems,* Vol. 3, No. 3, Sept. 1978, pp. 248–271.

Sekino, L. C., "Multiple Concurrent Updates," *Proc. Int'l Conf. Very Large Data Bases,* Framingham, Mass., Sept. 1975, pp. 505–507.

Shaw, A. C., *The Logical Design of Operating Systems,* Prentice-Hall, Englewood Cliffs, N. J., 1974, pp. 203–243.

Shemer, J. E., and A. J. Collmeyer, "Database Sharing: A Study of Interference, Roadblock and Deadlock," *Proc. 1972 ACM SIGFIDET Workshop Data Definition, Access, and Control,* Nov. 1972, pp. 147–163.

Shoshani, A., and E. G. Coffman, Jr., "Prevention, Detection, and Recovery from System Deadlocks," *Proc. 4th Ann. Princeton Conf. Information Sciences and Systems,* Mar. 1970.

Shoshani, A., and E. G. Coffman, Jr., "Sequencing Tasks in Multiprocess, Multiple Resource Systems to Avoid Deadlocks," *Proc. 11th Ann. Symp. Switching and Automata Theory,* Oct. 1970, pp. 225–233.

Stearns, R. E., P. M. Lewis II, and D. J. Rosenkrantz, "Concurrency Control for Database Systems," *Proc. IEEE 17th Ann. Symp. Foundations of Computer Sci.,* Oct. 1976, pp. $19 - 32$.*

Stucki, M. J., J. R. Cox, Jr., G. C. Roman, and P. N. Turcu, "Coordinating Concurrent Access in a Distributed Database Architecture," *Proc. Fourth Workshop Computer Architecture for Non-Numeric Processing,* Syracuse University, Syracuse, N. Y., Aug. 1978, pp. $60 - 64$.*

Trinchieri, M., "On Managing Interference Caused by Database Sharing," *Alta Frequenza,* Vol. 44, No. 11, 1975, pp. 641–650.

Tsichritzis, D. C., and P. A. Bernstein, *Operating Systems,* Academic Press, New York, 1974, pp. 45-49 and 259–261.

Wiederhold, G., "Integrity of Databases" (Chapter 13), *Database Design,* McGraw-Hill, New York, 1977, pp. 542–564.

Yourdon, E., "The Problem of Simultaneous Access to the Data Base" (Chapter L), *Design of On-Line Computer Systems,* Prentice-Hall, Englewood Cliffs, N. J., 1972.

Sreekaanth S. Isloor joined the CAD Database Development Department of Bell Northern Research, Ottawa, in September 1979 and is presently with the Language Development Department. During 1977–78, he served as sessional lecturer at the University of Alberta. His research interests include distributed database management systems, relational databases, and compiler construction.

He received the B. Tech. degree with distinction in electrical engineering from the Indian Institute of Technology, Madras, India, and the M. Tech. degree in computer sciences from the Indian Institute of Technology, Kanpur, India, in 1975. He graduated with the PhD degree in computing science from the University of Alberta, Edmonton, Canada, in August 1979. Isloor is a member of ACM, IEEE, and the Computer Society.

T. A. Marsland is a professor of computing science at the University of Alberta, with primary research interests in computer systems. Prior to his present appointment he was a member of the technical staff at Bell Laboratories in New Jersey. His postgraduate degrees in electrical engineering are from the University of Washington, where he was also a visiting assistant professor in 1968. After completing an honors degree in mathematics at the University of Nottingham, he worked in the U. K. aerospace industry and subsequently as a scientific programmer for the Boeing Company, Seattle.

An active member of ACM, IEEE, and CIPS, Marsland is an ACM National Lecturer on distributed processing and computer chess.