System Deadlocks

E. G. COFFMAN, JR.

Pennsylvania State University, University Park, Pennsylvania

M. J. ELPHICK

University of Newcastle upon Tyne, Newcastle upon Tyne, England

A. SHOSHANI

System Development Corporation, Santa Monica, California

A problem of increasing importance in the design of large multiprogramming systems is the, so-called, deadlock or deadly-embrace problem. In this article we survey the work that has been done on the treatment of deadlocks from both the theoretical and practical points of view.

Key words and phrases: deadlocks, deadly embraces, system deadlocks, multiprogramming, interlock problems

CR categories: 4.10, 4.32

1 INTRODUCTION

One of the objectives of recent developments in operating systems—incorporating multiprogramming, multiprocessing, etc.—has been to improve the utilization of system resources (and hence reduce the cost to users) by distributing them among many concurrently executing *tasks*. In any operating system of this type the problem of deadlock must be considered. Requests by separate tasks for resources may possibly be granted in such a sequence that a group of two or more tasks is unable to proceed—each task monopolizing resources and waiting for the release of resources currently held by others in that group. As a somewhat simplified example, suppose we have a task T_1 that has just produced a request for a file on disk, and that T_1 must suspend operation until the disk becomes available. Suppose another task T_2 has control of the (single) disk channel but that it is still waiting for T_1 to complete the updating of a (shared) file that contains information required by T_2. In order for work to proceed, the disk channel must be given over to T_1. But if we assume that T_2 is in the middle of some file on disk, then, because of hardware or software limitations, a preemption of the disk from T_2 can be effectively tantamount to a sacrifice in the processing already accomplished by T_2. Under these circumstances, if T_1 and T_2 have no (temporary) alternative to waiting, they will be deadlocked and never able to proceed.

As shown by this example, deadlocks, or "deadly embraces" as E. W. Dijkstra has called them, can arise even though no single task requires more than the total resources available in the system. Moreover, deadlocks can arise whether the allocation of resources is the responsibility of the operating system (as is normally the case) or of the programs themselves.

The example also illustrates that the term "resource," in the context of deadlocks, can apply not only to devices (tape and disk drives, channels, card readers, etc.), and processors and storage media (core store, drums, disks, tapes), but to programs, subroutines, and data (tables, files, etc.). Many of these permit only *exclusive* use by one

CONTENTS

task at a time, but some (for example, read-only programs) may be *shared* by more than one task. The mutual exclusion of several tasks using the same resource may be achieved by allocating the use of that resource to only one task, or, if access to the resource is not directly under the control of the operating system, by the use of the primitive P and V operations introduced by Dijkstra [1]. The facilities for queueing requests from separate tasks for a serially reusable resource provided in OS/360 by the ENQ and DEQ macros [2] are examples of this latter approach, as are the LOCK and UNLK macro instructions suggested by Dennis and Van Horn [3].

Another aspect of these various types of resource is the possibility of *preemption:* if a resource has been allocated by the system to a task, it may be possible (at some cost) to seize the resource from that task (which is then suspended) and preserve the current state both of the task and of its use of that resource. Thus, if the resource in question is a CPU, the information that must be preserved consists of the contents of CPU registers and the current instruction counter (or "program status word"). The cost of preemption is low in this case; however, for storage media like magnetic tape, the cost may be much higher. The cost may have to be regarded as effectively infinite in those cases where preemption of the given resource must involve the loss of "progress" of one or more tasks (as in the deadlock example above). We should emphasize that this type of resource may be inherently nonpreemptible or it may simply be that the operating system is not designed to preempt this type of resource.

It should also be noted that users of the system may make requests for resources either implicitly (e.g., by defining the data sets to be used for a given job in OS/360) or explicitly during the execution of a program. In some cases, a request for one resource will imply a further request for other resources: for example, an initial request for access to the information contained in a random-access file may lead to a request for main storage to be allocated for input and output buffers.

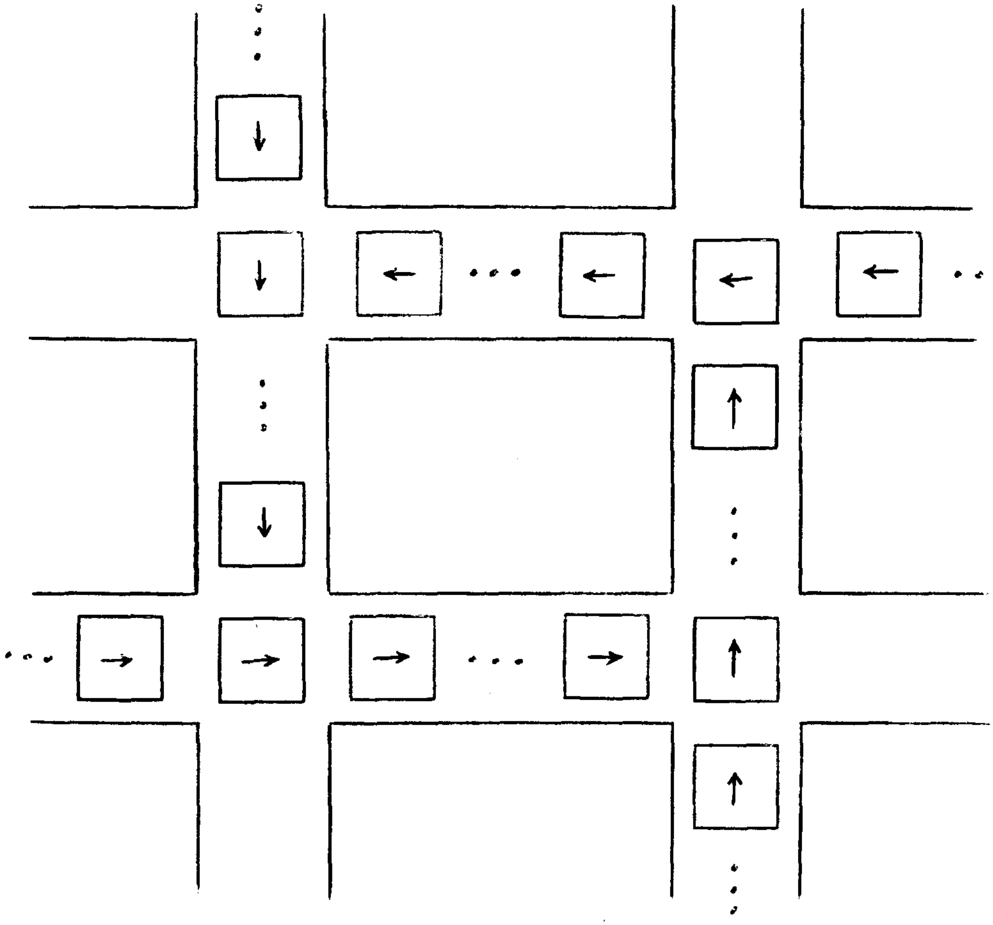

FIG. 1. Traffic deadlock.

Since the deadlock problem is a logical one it can arise in different contexts, provided that tasks and resources are interpreted properly. For example, consider the situation of traffic flow along the edges of a square as shown in Figure 1. The small boxes represent cars, and the arrows the direction in which they want to move. Since there are four cars blocking the traffic at each corner we have a traffic deadlock. In this case we can think of the space occupied by a car as a resource for which other cars are competing. Cars represent tasks that request the next resource (space) before they can release the current resource (space they currently occupy). This is a simple example of deadlock problems that can appear in traffic control. Many other examples arise in production management and control, where machines, tools, people, input information, etc., can be considered as resources needed for the execution of some task.

The deadlock problem becomes more complex when a system has different resource *types* and, in general, more than one resource of the same type. In this case resources of the same type are not labeled differently. When a request is made for a number of resources of a particular type, any resources of that type will do. Thus, the case of only one resource of each type is a special case of the above. Examples of many resources of the same type are memory pages (i.e., units of storage allocation), disk drives (if there is more than one in a system), processors in a multiprocessor system, etc.

The remainder of this paper is organized as follows. In the next section we summarize the conditions necessary for the existence of deadlocks and present two graphical methods for representing deadlocks. In sections 3–5 the treatment of deadlocks is classified according to the assumption of whether advance information about future resource requirements of tasks is available. For the case when no such information is available, we consider in section 3 the design of systems in which the possibility of deadlocks occurring is *prevented* from the start by removing one or more of the necessary conditions. We then discuss in section 4 how a state of deadlock may be *detected,* and how *recovery* can be carried out, preferably at the least cost to the users of the system. If advance estimates of the resources required by each job can be obtained, techniques are available for *avoiding* deadlock, and we discuss their cost and effectiveness in section 5. Finally, we attempt some conclusions about the relative merits of these various approaches to the problem of deadlock.

2 CHARACTERIZATIONS OF DEADLOCKS

In order to illustrate the conditions under which deadlocks can occur, we abstract the previous example and consider two tasks T_1 and T_2, each requiring the exclusive use of two different resources R_1 and R_2 (not necessarily together). We can represent the combined progress of these tasks in the following way (due to Dijkstra). For each task the number of instructions executed subsequent to some selected initial time is used as a measure of its progress, and a pair of such values defines a point in a two-dimensional "progress space," as shown in Figure 2. The joint progress of T_1 and T_2 is then represented by a sequence of discrete points in this space; sub-sequences in which only one coordinate increases correspond to time intervals in which one task is in sole control of the CPU, while simultaneous increases in both coordinates can only occur in a multiple processor system. It is clear that such a "trajectory" can never result in a decrease in either coordinate—progress (i.e., the execution of instructions) is irreversible.

The intervals during which tasks T_1 and T_2 require resources R_1 and R_2 are shown in Figure 2, as are those areas representing simultaneous use of a resource by both tasks. Since we have assumed that each task requires exclusive control of any resources, these areas cannot be entered by the trajectory representing their progress.

Given the sequence of resource usage by the two tasks shown in Figure 2, it is clear that if the trajectory is allowed to enter the region D, then a state of deadlock is inevitable—T_1 holds resource R_1, T_2 holds resource R_2, and the subsequent requests from each task for its second resource must be denied. Other tasks in the system may be able to continue if they do not require these resources, but the overall performance of the system will be degraded by the unavailability of R_1 and R_2.

This deadlock situation has arisen only because all of the following general conditions were operative:

1) Tasks claim exclusive control of the resources they require ("mutual exclusion" condition).
2) Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
3) Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).
4) A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

The existence of these conditions effectively defines a state of deadlock.

In the previous example these conditions were indeed in effect, and a circular wait existed, since T_1 was waiting for T_2 to release R_2, while T_2 was waiting for T_1 to release R_1.

In terms of these necessary conditions it is

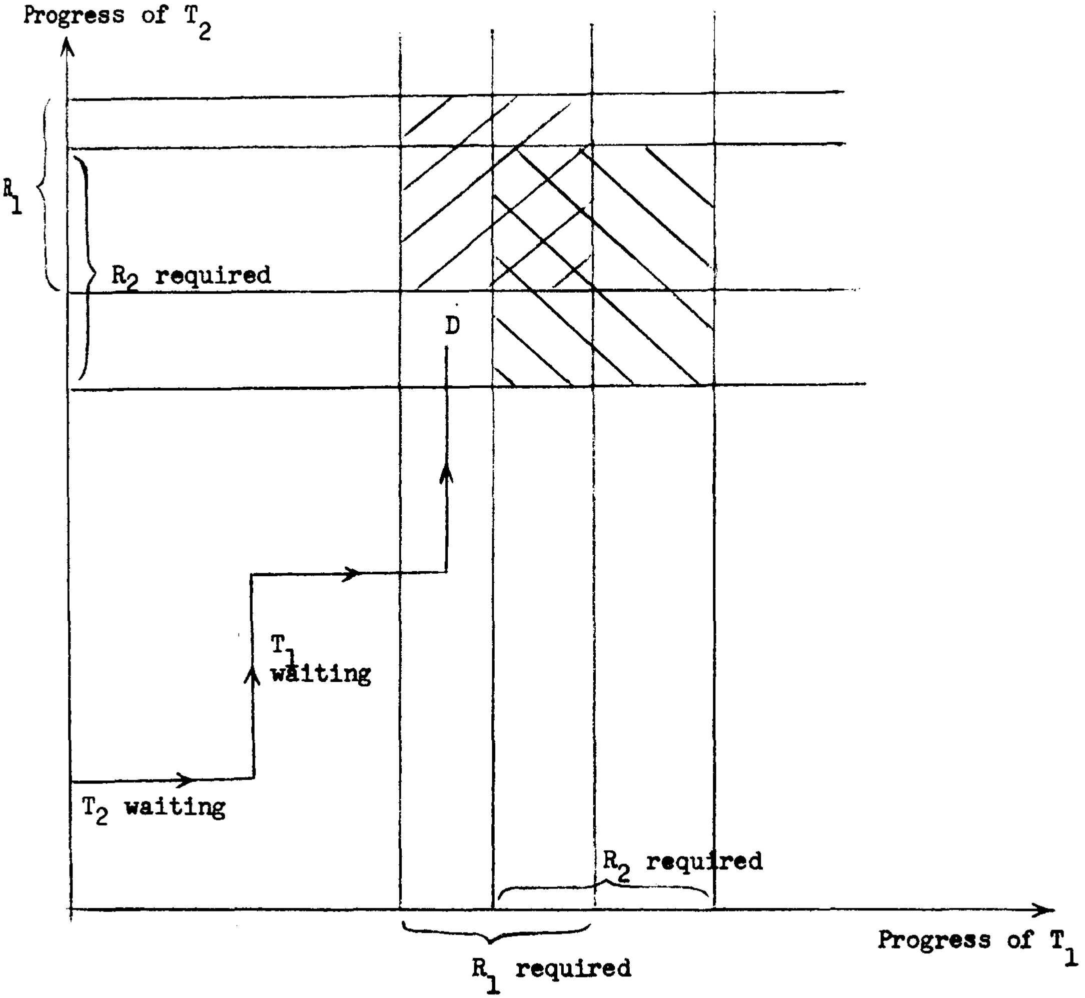

FIG. 2. Joint progress of tasks T_1, T_2.

interesting to examine again the example of traffic deadlocks in Figure 1. In this case there is only one resource of each type, since spaces occupied by cars are all distinguishable from each other. The "mutual exclusion" condition obviously holds since a space cannot be shared by two or more cars. The "wait for" condition exists, since a car cannot release the current space until it gets the next space. The "no preemption" condition clearly holds, assuming that cars are not removable by an outside agent. Finally, circuits in the state graph are possible because of the direction of flow around the square.

Deadlocks can be expressed more precisely in terms of graphs. Suppose we have a set of tasks $\{T_1, T_2, \cdots, T_n\}$ in some arbitrary state of execution; let $\{R_1, R_2, \cdots, R_m\}$ be a set of distinct resources in use by these tasks (i.e., there is only one resource of each type). We define a directed graph (to be called a state graph) whose nodes correspond to the resources $\{R_j\}$ and whose arcs are defined as follows. If, at the time instant to which the graph applies, some task T is allocated (i.e., possesses) resource R_j while requesting R_k, then the graph contains an arc from node R_j to node R_k. An example is shown in Figure 3 for $m = 3$. As can be

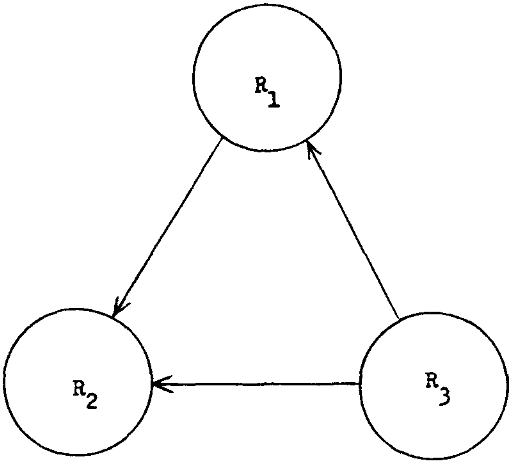

FIG. 3. Example of a state graph.

seen, there exists a task possessing R_1 and requesting R_2, a task possessing R_3 and requesting R_1 and R_2.

It has been shown [4, 6, 8, 14] that a circuit (directed loop) in the request graph is a necessary and sufficient condition for a deadlock, assuming the first three conditions given above are operative. As an example, suppose an arc from R_2 to R_3 is added to the graph in Figure 3. Clearly, there must exist at least three deadlocked tasks; each of the R_i is requested by one task but allocated to another so that none of the R_i can be effectively used.

The state graph described above is appropriate only for systems in which there is only one resource of a type. For the general case of more than one resource of the same type, a state graph was defined [14] as follows: resources are partitioned into types $r_1, r_2, \cdots, r_s$ with the number of resources of each type given by $w_1, w_2, \cdots, w_s$, respectively. The resources of a given type are identical and indistinguishable, and it is assumed that requests for a given resource type r_i can be for any number less than or equal to w_i. State graphs are defined in a way similar to that given earlier. The nodes now become the resource types $\{r_i\}$. There will be an arc from r_i to r_j if and only if there exists at least one task requesting one or more resources of type r_j and possessing one or more resources of type r_i. With this definition a circuit in the state graph is a necessary, but no longer a sufficient, condition for the existence of deadlocks. However, this result may still be meaningful in considering the prevention of deadlocks. If a restriction is imposed on the behavior of tasks in such a way that circuits in the request graph cannot arise, then deadlocks will never occur (such a restriction is discussed in section 3).

Two recent studies [16, 18] of deadlock problems have appeared in which different variations of graphical representations are investigated. Results based on graph structure that are similar to but more extensive than those discussed here under the headings of detection and avoidance are presented for these graphs.

3 PREVENTION OF DEADLOCK

If we undertake the design of a system in which the possibility of deadlock is to be excluded a priori, then we must ensure that at every point in time at least one of the necessary conditions is not satisfied. This implies certain constraints on the way in which requests for resources may be made.

The first of the four necessary conditions given in section 2 cannot be denied for all resources. For example, the sharing of a file by more than one task can only be permitted when neither task is updating the contents of that file.

The following approaches, suggested by Havender [4], in effect deny each of the three remaining conditions in turn.

a) Each task must request all its required resources at once and cannot proceed until all have been granted ("wait-for" condition denied).
b) If a task holding certain resources is denied a further request, that task must release its original resources and, if necessary, request them again together with the additional resources ("no preemption" condition denied).
c) The imposition of a linear ordering of resource types on all tasks; i.e., if a task has been allocated resources of type r_i it may subsequently request only those resources of types following r_i in the ordering. In the case of only one resource

of a type, the linear ordering is of resources. (It can be shown that under this condition the state graph cannot have circuits, and therefore circular wait conditions are prevented.)

When each of these strategies is only partially applicable it is clearly possible to use a combination of them in a given system, and also to implement them either by embodying the constraints in the design of the system (so that no task *can* violate the constraints), or by insisting that all application programs as well as system components follow certain conventions in requesting resources.

Approach (a) may be costly since some of the resources allocated to a task may remain unused for long periods. Approach (b) is convenient only when applied to preemptible resources whose state can be easily saved and restored later, as is the case with a processor. Approach (c) may be feasible and is used in the case discussed by Havender [4], where he describes the way in which the deadlock problem was tackled in the design of that component of the OS/360 (MVT) system responsible for initiating job-steps. Here certain possibilities of deadlock (between separate job-steps, at least) have been prevented by acquiring storage, devices, and data sets for each job-step in a fixed order. However, the possibility of deadlock between several tasks initiated by one or more job-steps has not been prevented completely, since the ENQ and DEQ facilities can be used (or misused) to create a "circular wait."

Other approaches to this problem are discussed in the report by Collier [5], and in the papers by Reiter [6], Murphy [7], and Merikallio and Holland [8].

4 DEADLOCK DETECTION AND RECOVERY

It was discussed in section 2 that in the case of one resource of a type, a deadlock exists at time t if and only if there exists a circuit in the state graph at time t. Thus, a deadlock detection mechanism in this case consists of a routine for maintaining a state graph each time resources are requested, acquired, or released by tasks, and a routine that examines the state graph to determine whether a circuit exists.

For the case of more than one resource of a type, a detection algorithm cannot be similarly based on the state graph since, in this case, a circuit in the state graph is only a necessary condition for a deadlock; a more elaborate state description mechanism is required.

Let $r_1, r_2, \cdots, r_s$ represent resource types and $w_1, w_2, \cdots w_s$ represent the number of resources of each type as in section 2. At an arbitrary time instant t let p_{ij} denote the number of resources of type r_j allocated to (possessed by) T_i and let q_{ij} denote the number of resources of type r_j requested by T_i in excess of those already allocated to T_i. Define the *allocation* and *request* matrices $P = ((p_{ij}))$ and $Q = ((q_{ij}))$ and let P_i and Q_i denote the row vectors giving the resources allocated to T_i and requested by T_i, respectively. Let $V = (v_1, v_2, \cdots, v_s)$ be an *available resources* vector whose *i*th element $(v_i \leqslant w_i)$ indicates the number of resources of type r_i that are currently available. Note that

$$v_j = w_j - \sum_{i=1}^{n} p_{ij}.$$

That is, the sum of the resources allocated and those available of type r_j must be equal to the total number of that type in the system. In the following, $\underline{0}$ will indicate a (row) vector each of whose s elements is 0. Also, $x \leqslant y$, where x and y are vectors, is defined to hold if and only if it holds for each pair of corresponding elements from x and y.

The algorithm below, which is presented in [14], is designed to reveal a deadlock by simply accounting for all possibilities of sequencing the tasks that remain to be completed. Suppose at time t the state of the system is available in the matrices P and Q, and suppose V is also given. We have the following algorithm for determining the existence of a deadlock at time t.

Algorithm A

1) Initialize $W \leftarrow V(t)$. Mark all rows for which $P_i(t) = \underline{0}$. (All rows are assumed to be "unmarked" at the outset.)
2) Search for an unmarked row, say the *i*th,

such that $Q_i(t) \leqslant W$. If one is found, go to Step 3; otherwise terminate the algorithm.

3) Set $W \leftarrow W + P_i(t)$, mark the ith row, and return to Step 2.

It is not difficult to verify that a deadlock exists if and only if there are unmarked rows at the termination of the algorithm. Moreover, the set of unmarked rows corresponds precisely to the set of deadlocked tasks.

The running time of this algorithm is proportional to the square of the number of tasks. It is shown in [16] (see also [17]) that by ordering the resource requests by size and associating with each task a count of the number of types of resources being requested, a detection algorithm can be devised which has a running time that varies linearly with the number of tasks. Moreover, the additional cost of ordering requests and maintaining counts is offset by the fact that this method also facilitates finding blocked tasks to activate when other tasks release resources. A number of simplifications are possible in important special cases (e.g., a single resource of each type), and these are also discussed in [16].

Given a detection mechanism, perhaps the simplest approach to recovery from a deadlock situation would involve aborting each of the deadlocked tasks, or, less drastically, aborting them in some sequence until sufficient resources become released to remove deadlocks in the set of remaining tasks. Obviously, we could also design an algorithm that searches for a minimum-sized set of tasks which, if aborted, would remove the deadlocks.

A more general technique [14] has been devised that assigns a fixed cost c_i to the removal (forced preemption) of a resource of type r_i from a deadlocked task that is being aborted. Thus, the cost of removing resources from a deadlocked task T_i is:

$$\sum_{j=1}^{s} c_i \cdot g(q_{ij} - v_j) \quad \text{where } g(x) = \begin{matrix} x; x > 0 \\ 0; x \leqslant 0 \end{matrix}$$

An algorithm has been designed that finds a subset of resources that would incur the minimum cost if preempted. The algorithm finds a minimum cost solution by an efficient tree-search procedure that can be characte ized as a branch-and-bound technique. (A gorithm A is used to isolate new sets deadlocked tasks in the sequence of tria removals made by the algorithm.)

Note that by using this detection-reco ery mechanism the notion of deadlock ca be extended to include resource types whi are actually preemptible but which ha varying preemption costs. In other word with this extension, deadlocks do not nece sarily involve tasks that must be aborte rather, it may only be necessary to remo certain resources from tasks that incur su costs as supervisor overhead, swapping, et but not a loss of input, output, or the pro ress that has been made in a computatio Clearly, forced resource preemptions invol ing the latter losses will be assigned hig costs so that the recovery algorithm w avoid them.

5 AVOIDING DEADLOCKS USING INFORMATION ON RESOURCE REQUIREMENTS

To avoid deadlocks in a multiprogrammin system in which the necessary conditions f deadlocks can exist, it is usually necessary t have some advance information on the r source usage of tasks. A number of interes ing models can be envisioned, each differin in the amount of information assume available. A comprehensive theoretical stud of a number of task models can be found i [10]. In this section we shall briefly examir two such models: the first will be the mo basic model (full information assumed) an the second will be the more practically or ented model of Habermann [11].

In the basic model a task is assumed t consist of a sequence of *task-steps* durin each of which the resource usage of the tas remains constant. The execution of a task step first involves the acquisition of thos resources needed by the given task-step bu not passed on by the previous task-ste Next follows a period of execution durin which the resource requirements do no change. Finally, at the completion of execu tion, all those resources not needed by the subsequent task-step are released and returned to a pool of available resources.

Before describing how deadlocks are avoided with the task-step model we shall find it convenient to introduce the notion of *safe* states. As indicated in the previous section, the state of the system as it relates to the requesting and allocation of resources at time t is reflected in the two matrices $Q(t)$ and $P(t)$, respectively. For simplicity let us denote this resource-state by $S(t)$. We shall say that $S(t)$ is *safe* when, using the currently available resources and those which will be returned by currently executing task-steps, it is possible to find a valid sequence of the uninitiated task-steps in the currently initiated but incomplete tasks such that all tasks in the system can be run to completion. A sequence of this kind will be valid if the implied sequence of resource requests is such that at the time each request is made there are sufficient available resources to satisfy it.

As an illustration of this definition, consider Figure 2 in which the region D corresponds to points in the joint progress of T_1 and T_2 from which a deadlock is inevitable: As we can see, all points in the region D correspond to unsafe states and all points outside this region (and the shaded region) correspond to safe states.

The following remarks should be made about the definition of a safe state. First, we note that the initial state in which no resources are yet allocated and all are available is always a safe state. Hence, there always exists an initial sequence of all task-steps according to which all tasks are completed. In particular, we can always execute the tasks in strict serial sequence if necessary. In part it follows from this observation that if there is a way to complete the currently executing tasks without getting into a deadlock, we can clearly complete the remaining uninitiated tasks (again, executing them in serial order if necessary). Finally, with an appropriate formalization of the ideas in this and the previous section, it has been shown that the ability to avoid deadlocks subsequent to time t can be guaranteed if and only if $S(t)$ is a safe state [10].

We are now in a position to state precisely the problem confronted by a supervisor designed to avoid deadlocks. Suppose the system is in a safe state $S(t)$ and there exists a set of outstanding resource requests given by the contents of $Q(t)$. The supervisor must be able to determine whether any given one of these requests (if any) is safe in the sense that it is less than or equal to the currently available resources and would leave the system in a safe state if granted. Since, in general, there can be more than one safe request, some other scheduling criterion has to be invoked to determine which safe request is to be granted. Common criteria include first-in-first-out, shortest (or smallest) job first, etc. It is worth emphasizing perhaps that these other scheduling criteria must be applied to the set consisting only of safe requests. Otherwise, if they are applied to all requests, a request that is not safe may be selected, thus artificially creating a deadlock arising from conflicting scheduling policies.

The problem of determining whether a state $S(t)$ is safe following the granting of a request generally requires a procedure to search for an appropriate sequence of task-steps. As implied earlier, the procedure begins by assuming that the currently available resources are augmented by those allocated to currently executing task-steps. This is equivalent to assuming that the first task-step in the sequence sought is not begun until all currently executing task-steps have been allowed to complete. The search time can be improved by the following result [10].

At some stage in the search procedure let $x = p_1p_2 \cdots p_r$ be an initial (trial) subsequence of task-steps. Assuming that this sub-sequence were realized, suppose that the number of resources available following any given task-step is, for each type, greater than or equal to the number of resources available just following the previous task-step. Under this condition it has been shown that if the state $S(t)$ being tested is in fact safe, then there must exist a valid sequence of all uninitiated task-steps in the executing tasks that has x as a prefix (i.e., $p_1p_2 \cdots p_r$ will be the initial task-steps in the sequence). Thus, in organizing a conventional tree-search for appropriate sequences it is never necessary to back-up beyond the last element of an initial sub-sequence having a corresponding sequence of available resource vectors with the above property.

From practical considerations it is likely to be necessary to assume that less is known about the resource usage of tasks than is assumed in the task-step model. We shall now briefly discuss Habermann's model [11] in which we deal with entire tasks rather than task-steps, and in which it is assumed that only the maximum number of resources required by each task is known. In particular, for each task T_i we have a resource vector that gives the maximum number of resources of each type that will be required at any time during execution of T_i.

A state $S(t)$ will be safe in this model if and only if there exists some sequence according to which the currently executing (i.e., initiated but incomplete) tasks can complete, *assuming they will still need their maximum resource requirements* at some time, and assuming that we have only the currently available resources at the beginning of the sequence. The search for sequences of the executing tasks can be significantly more efficient than the search described with the task-step model, even after the fact that there are generally fewer tasks than task-steps is taken into account. The reason for this is as follows. Since each task will return as many resources as it requests, the sequence of available resource vectors corresponding to a sequence of the executing tasks will always have the monotonically non-decreasing property mentioned earlier. Thus, we can show that in the search procedure it is never necessary to back-up to try new initial sub-sequences; and it follows that the search time is proportional to the square of the number of executing tasks.

A further technique mentioned by Habermann for shortening the search time is described as follows. Suppose a request has just been made by task T_i, the system is in a safe state, and we wish to determine if the request is safe. If at *any* point in developing a trial sequence we find that the corresponding available resources are sufficient to complete T_i, we may terminate the search with the decision that the request is safe. The reason is that if T_i, the source of the request, can complete and release all the resources it controls, all other tasks can certainly be completed, since the state previous to the request was safe and they could therefore have been completed before the request was made.*

In summary, the applicability of the basic task-step model means that we can determine precisely when deadlocks are unavoidable on granting a request. Although with Habermann's model false threats of deadlocks may degrade resource utilization, it has the significant advantages of requiring less information about task behavior and a more efficient algorithm for testing whether states are safe. For models that moderate the disadvantages (and the advantages)of both the above models see [10].

Hebalkar [18] uses a graph model to represent processes of more general structure than the sequence of task-steps. In his model, nodes represent transitions of a computation and arcs represent demand vectors of resources. Thus, a computation can split into parallel subcomputations, and subcomputations can merge. A cut-set of a graph represents a state of the system. This model has been shown to facilitate the representation of safe states as well as deadlock states. Algorithms designed to preclude deadlocks are presented, and are based on the advance information in the graph model.

6 CONCLUSIONS

We have described the various strategies that can be used to deal with the problem of deadlock, under the headings of *prevention*, *detection* (and recovery), and *avoidance*. Algorithms for implementing the last two policies have been outlined briefly.

Where possible, prevention of the possibility of deadlock is better than cure, since in most current systems deadlock is an ex-

* For techniques leading to search times that vary linearly with the number of tasks and for applications to important special cases, see [16]. Such techniques arise from the more efficient detection algorithms mentioned in section 4.

ceptional occurrence, and in many cases it involves only a limited set of resources. Once a deadlock situation has been recognized as such, the problem can often be removed by suitable changes to that part of the system (for example, permitting preemption of a resource where this was not thought necessary before).

If this approach is not feasible—perhaps because the constraints it would impose on jobs are unacceptable—then detection or avoidance may be considered. However, it is not possible at present to predict accurately either the costs or the benefits that can be obtained by use of the techniques described. Good measures of the running time of the various search algorithms and of the resource utilization achieved by systems using them are not available.

Some refinements of the models presented here for the detection, recovery, and avoidance algorithms would be desirable in practice. The recovery model, for example, should use costs (for the preemption of resources from a task) that are functions of the time for which that task has been in progress, and perhaps of its priority. The selection of an appropriate avoidance model involves a balance between the efficiency of the algorithm (for determining whether granting a request is safe), the level of resource utilization possible, and the cost to the user in providing estimates of potential resource requirements.

In avoiding deadlock, an alternative to the search algorithms presented may be feasible when the number of safe sequences is relatively small (and the system may be operating inefficiently, having to deny most requests). This alternative approach is to store the current set of safe sequences, updating them as necessary, and to inspect the first element of each sequence to decide whether a request can be granted. However, as the number of safe sequences increases, the storage and updating problems will make this approach less efficient, and some scheme for switching from one method to the other might be necessary.

In practice, even though complete deadlock is avoided, it may be more important, for the sake of efficiency, to prevent the system from entering a state of "near deadlock," in which progress can be made only by granting requests from one task at a time. Again, when jobs with heavy resource requirements are discriminated against, the situation in which such a job is allowed to acquire control of several valuable resources, but is subjected to long waits as a result of further requests, must be avoided. The models described above might be adapted for this purpose by defining a cost for a given sequence of resource allocations (which would be high when a valuable resource is unavailable for a long sequence of task-steps); a search for a safe sequence having a low cost might be feasible.

On the other hand, one could focus on reducing supervisor overhead rather than improving resource utilization by adopting the following approach. Assuming advance resource-usage information, as in Habermann's model, for example, an operating system could be designed that never multiprograms two or more tasks whose resource requirements are such that circular wait conditions can arise. That is, deadlocks are avoided by inspection of resource requirements only at the beginning of task executions rather than each time a resource request is made. This approach clearly sacrifices potential losses in resource utilization for savings in supervisor execution times.

Needham and Hartley [12] have examined system designs partially involving this approach but have not attempted a complete removal of the possibility of deadlock (or "knotting," as the authors refer to it). In the interest of system operating efficiency, deadlocks (and the resulting "disasters") are allowed to occur, but measures are described that ensure that such events are kept at an acceptably low frequency.

In conclusion, it must be agreed that the problem of deadlock has not been of major (or at least continuing) importance in most current systems, mainly because it has amounted to little more than an isolated debugging problem. However, for future systems sharing an increasingly costly set of services and resources among an increasing number of individual users, these problems are likely to become more pressing.

They will be even more important for those systems that provide a common set of large files (or data bases) available to the many users of the system for both read-only access and updating. The typical resource usage here will be to obtain access to a small subset of records, from which other tasks must be excluded until the task at hand releases the subset. Some techniques for organizing this type of system to avoid deadlock are discussed by Shoshani and Bernstein [13].

REFERENCES

1. Dijkstra, E. W. "Co-operating sequential processes." In *Programming languages: NATO advanced study institute.* F. Genuys (Ed.), Academic Press, London, 1968.
2. *IBM System/360 operating system, supervisor and data management services.* Form C28-6646-2, IBM, White Plains, N. Y., 1968.
3. Dennis, J. B.; and Van Horn, E. C. "Programming semantics for multiprogrammed computations." *Comm. ACM* **9,** 3 (March 1966), 143–155.
4. Havender, J. W. "Avoiding deadlock in multi-tasking systems." *IBM Systems Journal* **2** (1968), 74–84.
5. Braude, E. J. "An algorithm for the detection of system deadlocks." IBM Technical Report: TROO. 791, IBM Data Systems Division, Poughkeepsie, N. Y., 1961.
6. Collier, W. W. "System deadlocks." IBM Technical Report TR-00. 1756, IBM Systems Development Division, Poughkeepsie, N.Y., 1968.
7. Reiter, A. "A resource-allocation scheme for multi-user on-line operation of a small computer." *Proc. AFIPS SJCC,* Vol. 30, pp. 1–7. AFIPS Press, Montvale, N. J., 1967.
8. Murphy, J. E. "Resource allocation with interlock detection in a multi-task system." *Proc. AFIPS FJCC,* Vol. 33, Pt. 2, pp. 1169–1176. AFIPS Press, Montvale, N. J., 1968.
9. Merikallio, R. A.; and Holland, F. C. "Simulation design of a multi-processing system." *Proc. AFIPS FJCC,* Vol. 33, Pt. 2, pp. 1399–1410. AFIPS Press, Montvale, N. J., 1968.
10. Shoshani, A.; and Coffman, E. G. "Sequencing tasks in multi-process, multiple resource systems to avoid deadlocks." In *Proc. 11th Annual Symposium on Switching and Automata Theory,* Oct. 1970, pp. 225–233.
11. Habermann, A. N. "Prevention of system deadlocks." *Comm. ACM* **12,** 7 (July 1969), 373–377, 385.
12. Needham, R. M.; and Hartley, D. F. "Theory and practice in operating system design." *Proc. 2nd ACM Symposium on Operating Systems Principles,* pp. 8–12. Brandon/Systems Press, Princeton, N.J., 1969.
13. Shoshani, A.; and Bernstein, A. J. "Synchronization in a parallel-accessed data base." *Comm. ACM* **12,** 11 (Nov. 1969), 604–607 [Also GE Report No. 69-C-138].
14. Shoshani, A.; and Coffman, E. G. "Prevention, detection, and recovery from system deadlocks." In *Proc. 4th Annual Princeton Conf. on Information Sciences and Systems,* March 1970. (See also Computer Science Lab. Technical Report No. 80, Department of Electrical Engineering, Princeton University, 1969.)
15. Dijkstra, E. W. "The structure of the THE-multiprogramming system." *Comm. ACM* **11,** 5 (May 1968), 341–346.
16. Holt, Richard C. "On deadlock in computer systems." (PhD Dissertation) Department of Computer Science, Cornell University, Ithaca, N.Y., Jan. 1971.
17. Russell, R. D. "A model of deadlock-free resource allocation—preliminary version." Memo CGTM #93, Department of Computer Science, Stanford University, Stanford, Calif., June 1970.
18. Hebalkar, Prakash. "Deadlock-free resource sharing in asynchronous systems." (PhD Dissertation) Electrical Engineering Department, MIT, Cambridge, Mass., Sept. 1970.