Improvements on Habermann's Algorithm for Deadlock Avoidance

Zhao, Wei

University of Massachusetts
Department of Computer Science
Amherst, MA

Technical Report 83-17

May 1983

A paper submitted in fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

at the

Department of Computer and Information Science

University of Massachusetts

Amherst, MA 01003

Acknowledgements

The author is much indebted to Professor David Stemple and Professor Krithivasan Ramamritham, Department of Computer and Information Science, University of Massachusetts, for their continual encouragement, enthusiastic support, and clever guidance. Professor Xu Guoding, Computer Science Department, Shanghai Normal University, read an earlier draft and made valuable comments. Mrs. Carol Angus, Division of Continuing Education, University of Massachusetts, painstakingly corrected the author's English.

Contents

Abstract

A well-known problem encountered in the design of operating systems is deadlock. Generally, three strategies are used to deal with it: prevention, avoidance, and detection/recovery. This paper focuses on deadlock avoidance for nonpreemptable but reusable resources. First, Habermann's algorithm is reviewed and discussed. Then suggestions for improving concurrency and efficiency are introduced. Finally, the relationship between deadlock avoidance and process schedulers is analyzed as a related and important aspect of deadlock avoidance.

1. Introduction

Because the problem of deadlock is a critical issue in operating systems and data base systems, it is one of the most researched areas in computer science. Various reseachers have formalized and analysed it [Beauqqier 1980, Coffman 1971, Habermann 1969, Havender 1968, Holt 1971, Sugiyama 1977]. Although some results of this research have been applied to actual systems, a wholly satisfactory solution has yet to be formulated. This paper intends to contribute towards resolving this important problem.

A system becomes deadlocked if in the system there is a set of processes unable to proceed since each process in the set is waiting for another. Generally speaking, deadlocks are caused by competition among processes for nonpreemptable resources. A nonpreemptable resource is a resource which cannot be removed from the process holding it until the resource is used to completion. Resources may also be divided into another two classes: nonreusable (or consumable) and reusable. This paper will focus solely on nonpreemptable but reusable resources.

At the present time, three strategies to deal with deadlock exist: prevention, avoidance, and detection/recovery. In deadlock prevention, a system is designed to prevent the occurrence of deadlock. One trivial method of deadlock prevention is called "maximum acquisition". This method will not allow a process to begin its run until all resources it will request during its run time are available and have been acquired. Since each process of this kind is guaranteed a completed run, deadlock is prevented; however, this method lowers the concurrency of the system. A second method of deadlock prevention is called "ordered resources" [Havender 68]. This method prevents deadlock in a system by allowing any process to request resources in some pre-specified order, thereby effectively eliminating the circular waiting chains among processes and resources in the system. Concurrency in this kind of system, however, still remains low.

Although the <u>deadlock avoidance</u> method allows deadlock states to loom, whenever deadlock is approached, it is carefully sidestepped. Habermann formulated an algorithm for deadlock avoidance [Habermann 69], which is based on the banker's algorithm [Dijkstra 65]. Since then some improvements have been made on it [Holt 71, Habermann 72, Habermann 78, Minoura 82]. This paper intends to suggest further improvements. Through either the strategy of deadlock prevention or deadlock avoidance, a system can never fall into a deadlock state. In deadlock prevention, a system need not test system states once it has started; in deadlock avoidance testing is necessary. On the other hand, a deadlock avoidance method may offer a higher degree of concurrency than a deadlock prevention method.

In a system using the strategy of <u>deadlock detection/recovery</u>, a deadlock may occur, but the system is able to detect the deadlock occurrence and take action appropriate to recover the system from deadlock. This kind of system offers the highest degree of concurrency, but sometimes the recovery cost may be very high, and even prohibitive. Holt set up a model to detect deadlock, and discussed its properties in detail ([Holt 71], [Holt 72]).

From the system point of view, the following table summarizes the three strategies referred to in the preceding discussion:

	Deadlock Occurrence	Test Required At Run Time	Concurrency
Prevention	never	unnecessary	low
Avoidance	never	necessary	moderate
Detection /Recovery	possible	necessary	high

Table 1 Three Strategies for Deadlock Avoidance

The remainder of this paper will deal primarily with several aspects of deadlock avoidance. In section 2, Habermann's algorithm for deadlock avoidance will be introduced, with slight alterations from the original in order to provide a clearer understanding of Habermann's intent. Introducing the concept of "process graph", section 3 provides a method which improves the concurrency realizable using Habermann's original algorithm. Section 4 discusses improvements in efficiency. In section 5, the issue of the relationship between deadlock avoidance and the scheduler is introduced, which is easily ignored by system designers. Finally, section 6 presents a summary and discussion of what this study has accomplished.

2. Habermann's Algorithm

This section introduces Haberman's algorithm for deadlock avoidance. The statements of some of the theorems are slightly different from Habermann's original paper [Habermann 1969]. The author believes that these are really Haberamnn's intent, as represented in [Holt 71] and [Habermann 72].

2.1 System and its States

Systems discussed in this paper will always consist of a set of processes: P1, P2, ..., Pn; and m types of resources: R1, R2, ..., Rm; where n >= 0 and m >= 0. Each type of resource has several identical units. Each process Pk claims its maximum requirements for each type of resource. Once this is done, the system allows the process to take its claimed resources concurrently.

The system state is described by two vectors AVAIL and REM, and two matrices GOAL and ALLOC, where

$$\text{AVAIL} = \begin{bmatrix} a1 \\ a2 \\ : \\ : \\ am \end{bmatrix}$$

is the numbers of resources available in the system,

$$\begin{aligned} \text{AVAIL[i]} &= ai \\ &= \text{number of resource Ri} \qquad \text{(i=1..m);} \end{aligned}$$

and $$\text{MAXC} = (\ \text{MAXC1} \ldots\ldots \text{MAXCn}\ )$$

$$= \begin{bmatrix} mc11 & \ldots\ldots & mc1n \\ mc21 & & mc2n \\ \vdots & \vdots & \vdots \\ mcm1 & \ldots\ldots & mcmn \end{bmatrix}$$

is the maximum resource claims made by processes,

MAXC[i,j] = MAXCj[i]

= maximum number of units of Ri claimed by Pj;

and ALLOC = (ALLOC1 ...... ALLOCn)

$$= \begin{bmatrix} al11 & \ldots\ldots & al1n \\ al21 & & al2n \\ \vdots & \vdots & \vdots \\ alm1 & \ldots\ldots & almn \end{bmatrix}$$

is the allocation matrix,

ALLOC [i,j] = ALLOCj[i] = alij

= the number of Ri allocated to process Pj.

Obviously, ALLOC varies with time so that ALLOC(t) is used to denote the ALLOC at time t when necessary.

REM is not independent, which is defined by

$$\text{REM} = \text{AVAIL} - \underset{i=1..n}{\text{SIGMA}}\ \text{ALLOCi}$$

indicating the numbers of resources remaining in the system, ready to be allocated to processes upon request. Here "SIGMA" stands for the arithmetic sum of the arguments following it. This notation will be used throughout this paper. When necessary, REM(t) will be used to indicate the REM at time t.

To state the problems precisely, it is necessary to define the following relations:

<u>Definition</u>

$$\text{Let} \quad U = \begin{bmatrix} u1 \\ u2 \\ : \\ : \\ um \end{bmatrix} \quad \text{and} \quad V = \begin{bmatrix} v1 \\ v2 \\ : \\ : \\ vm \end{bmatrix}$$

be two integer vectors, we say U <= V if

U[i] = ui <= vi = V[i] for all i=1..m.

<u>Definition</u> Let F = (F1, F2, .., Fn) and G = (G1, G2, .., Gn) be two (m rows and n columns) integer matrices, we say F <= G if Fi <= Gi for all i=1..n, where Fi and Gi are m-dimension integer vectors.

Similarly, we may define relations >= on vectors and matrices. In the case where the context is clear 0 may be used as a 0 vector or 0 matrix, which means all the elements of the vector or matrix are 0.

The above relations may be used to represent the conditions that must hold in the systems being proposed:

```
For all j=1..n,
   R1:  MAXCj  <= AVAIL  (Claims must not be more than AVAIL);
   R2:  ALLOCj <= MAXCj  (No process can get more resources
                          than it claims);
   R3:  REM    >= 0      (System never issues loans).
```

<u>Definition</u> A system state (AVAIL, MAXC, ALLOC(t), REM(t)) is realizable if R1, R2, and R3 hold.

2.2 Safe State and Deadlock Avoidance

It is obvious that a realizable system state does not guarantee that a deadlock cannot occur. A system state free from the deadlock danger is called as a "safe" state. Within the definition of deadlock, a state is considered "safe" when each process is enabled to complete its run from this state in some way.

In order to achieve such a situation, REM(t) must be sufficient to serve at least one process, for example, say Pk1,

$$MAXCk1 - ALLOCk1(t) <= REM(t)$$

so that process Pk1 can be completed. When Pk1 is completed, it will release all resources that it holds. In this way, more resources are free to serve another process, say Pk2,

$$MAXCk2 - ALLOCk2(t) <= REM(t) + ALLOCk1(t)$$

If the system state is "safe", this action may be repeated until all processes are completed.

In addition, it is necessary for a safe state to be realizable although not every realizable state may be safe. Formally, a safe state is defined as follows:

<u>Definition</u> A system state is safe if it is realizable and there is at least one sequence S of all processes such that:

R4: $\text{MAXCs}(i) - \text{ALLOCs}(i)(t) <= \text{REM}(t) + \underset{k < i}{\text{SIGMA}}\ \text{ALLOCs}(k)(t)$

for all i=1..n; where i and k are indices in S, and s(i) or s(k) indicates the original id number of the process which has indices i or k in S.

We call such a sequence S, which leads all processes to completion, as a safe sequence. The definition "safe state" indicates the fact that starting from a safe state there is at least one way to run the processes to completion. Therefore:

<u>Theorem</u> <u>1</u> The system will not become deadlocked if two conditions holds:

1. the concurrent system state is safe; and
2. the scheduler will not reject at least one of the safe sequences.

Proof: Because the state is safe, there is a safe sequence satisfying R4; and if the scheduler does not reject at least one safe sequence, the processes may be completed in the order of that sequence. []

The key to test the safety of a system state lies in finding one safe sequence. However, there are n! sequences. In the worst case, it is necessary to test as many as n! sequences. That requires too much work! Fortunately Habermann proved:

<u>Theorem</u> <u>2</u> If a subsequence S fulfills condition R4 and S cannot be extended into a safe sequence, then the state is not safe.

Proof: There may be many ways to try to extend S. Let S' be the longest extension of S and

T = {all processes} - {those in S'}.

If S cannot be extended into a safe sequence, which must contain all processes, then, S' cannot be extended either. Therefore, if the state is safe, the existing safe sequence must not begin with any process in S'.

By the definitions of S' and T,

$$\begin{aligned} MAXCt(i) - ALLOCt(i) &> REM + \sum_{\text{all } j \text{ in } S'} ALLOCs'(j) \\ &>= REM. \end{aligned}$$

Therefore, the safe sequence cannot begin with any process in T. Because T unions S' = {all processes}, so that if a safe sequence existed, no process could be the first process in the sequence. It follows that there exists no safe sequence. Consequently, the system state is not safe. []

Since an empty subsequence fulfills R4, verifying the safety of a system state may necessitate starting from the empty subsequence and trying to extend it. At any point if the subsequence cannot be further extended, according to Theorem 2, rather than looking for another sequence, it is possible to conclude that there is no safe sequence at all. Because Theorem 2 significantly reduces computation cost, it remains one of the basic theorems used to construct the deadlock avoidance algorithms..

2.3 State Transitions

In practice, a system utilizing the deadlock avoidance strategy may work in this way: whenever a process comes into the system, the system will test whether it satisfies R1: no process claims more resources than available in system. If it does not fulfill the requirement, the process is rejected. When the system starts, because no resource has yet been allocated, the system state is safe. Each time a process requests some resource(s) and sufficient unallocated resources exist to fulfill the request, the system state resulting from the proposed allocation will be checked. If the state after allocation is safe, then the resource(s) is allocated to the process, otherwise, this request has to be rejected until a later time when the process may make another attempt. It is necessary to check the state resulting from the proposed allocation because decreasing REM may make the system state unsafe, even if the original state is safe and there are enough resources to allocate. The procedure becomes simpler, however, when a process releases some resource(s). At this time REM will be increased, therefore the state following the deallocation must be safe if the original state is safe. In this way the system state always moves from one safe state to another safe state, guaranteeing that deadlock can never occur.

From the above discussion, it can be concluded that the test of safety for the system state following an allocation must be based on the fact that the original state is safe. Computation cost for this kind of test can be reduced by utilizing the following theorem:

Theorem 3 The system state resulting from an allocation will be considered safe if the original state is safe and Pk, which makes its request to cause this allocation, appears in one of the subsequences S fulfilling R4.

Proof: See [Haberman 69]. []

Intuitively, if Pk appears in one of the subsequences fulfilling R4, Pk is able to complete. When Pk completes, it will release all the resources it holds, including the one allocated this time. The resources available in the system will at least the same as those prior to this allocation. Because the system state prior to this allocation is safe, every other process is able to complete. Therefore the system state after this allocation is safe.

Using this theorem makes it unnecessary to construct a full safe sequence to prove the state after allocation is safe. Only a sub-safe-sequence need be constructed to check if process Pk, which makes this state transition of the system, has been included in the subsequence.

Combining Theorem 2 and Theorem 3, the following algorithm emerges to verify whether the system state is safe after the resources are allocated to Pk:

```
Function SAFE (Pk : process) : boolean;

Var Safe-sub-seq,
    T            : set of process;

   Begin
      Safe-sub-seq := empty;
      T            := all processes;

      While (Pk is not in Safe-sub-seq) and (T is not empty) do
         Begin
            if not EXTEND(Pk, T) then
               EXTEND(another member in T, T);
         End;

      If Pk is in Safe-sub-seq then
         SAFE := true
         else
         SAFE := false;
   End;
```

Where EXTEND is a subroutine:

```
Function EXTEND(Pi : process; T : set of processes) : Boolean;

   Begin
      Subtract Pi from T;

      If Safe-sub-seq is extendable with Pi then
      Begin
         extend Safe-sub-seq with Pi;
         T := {all processes} - {Those in Safe-sub-seq}
         EXTEND := true;
      End
      else
         EXTEND := false;
   End;
```

3. Improvement in Concurrency

In this section, we first introduce the concept of process graph and define process states. Based on the process graph and process states, then we propose a system in which the concurrency is improved. The results represented in this section are formalized.

3.1 Process Graph and States

It should be noted that if some processes do not acquire their maximum claims for resources concurrently, then even in some "unsafe" state of the system, deadlock may not occur. This fact suggests that more concurrency can be gained by providing to the system more "dynamic" information about how processes acquire resources. To explain how such "dynamic" information can be given to the system, it is first necessary to illustrate how a process acquires resources explicitly. As a user of resources, a process may be represented as a set of tracks recording relatively when the process needs and holds the resources. This set of tracks is called as the "graph" of the process. For example, the following figure is the graph of a process that holds R1 from t1 to t6, R2 from t2 to t3 and t4 to t5, and also holds another R2 from t3 to t6, and so on.

```
        R3+----                              +--------
        R2+----     +----
             R2+--------------          +----+----
 . R1+----+----+----+----+----.    .    .    .    .   .
 t0  t1   t2   t3   t4   t5   t6   t7   t8   t9   t10 t11 ...-->t
```

Figure 2   The graph of a process

Notice that even between t6 and t8, no resource is held. For purposes of this discussion, on the graph of process, the following notations may be defined:

Definition The function "Number_of_Resource" is defined as

NR(Ri, Pj, t) = the number of units of Ri held by Pj at time t.

NR(*, Pj, t) = the sum of units of all types of resources held by Pj at time t.

Where there is no ambiguity, process name Pj and/or time t in the arguments in NR may be omitted. For example, in the above graph, NR(R2, t4)=2 and NR(*, t4)=3.

Definition A local block in a process graph is an area between two points, tb and te, where NR(*,tb) = NR(*,te) = 0 and there exists at least one point tm such that tb < tm < te and NR(*,tm) <> 0.

Definition A local block is considered to be basic when it cannot be further divided into local blocks.

Generally, an entire process graph is a local block, but often it is not the basic one. In the Figure 1, the area from t0 to t11 is a local block; however the areas from t0 to t7 and from t7 to t11 are also local blocks, and are the basic local blocks.

Definition The up point in a local block of a process graph is that point from the beginning point up to which NR(*)=0; and at which NR(*)<>0. That is, in the local block, the process makes its first request at the up point.

Definition The down point in a local block of a process graph is that point after which NR(*) is decreasing; and before which it may not. In other words, in the local block, the down point is the first point beyond which no more allocations are made.

Definition The completed point in a local block of a process graph is that point after which, NR(*)=0; and at which, NR(*)<>0. Stated simply, in the local block, the process makes its last release at the completed point.

For example, in Figure 2, in the local block form t0 to t7, t1 is the up point; t4 is the down point; and t6 is the completed point. As pointed out in the definitions, the up point, down point and completed point in a given local block are the first request point, the last request point, and the last release point of resources, respectively. Therefore,

Corollary The up point, down point and completed point in a given local block are unique.

Based on the above notations defined on process graphs, the states of a process may be defined as follows:

Definition Given a local block partition of its process graph, a process has three states in a system: (1) the up state, when it runs from the up point to the down point in a local block; (2) the down state, when it runs from the down point to the completed point in a local block; (3) the free state, when it is neither in an up state nor in a down state.

3.2 Concurrency-improved System

This subsection proposes an improved system, which may use information on process graphs in order to achieve more concurrency than the system described in section 2.

In this concurrency-improved system, a process must report to the system the transitions of its states and the maximum requirements for resources in the current local block when the process state transits from "free" to "up". The latter requirement is called "local maximum claims". Now the "global maximum claims" may not be provided. Such a procedure implies that the graph of a process must be dynamically partitioned in some way. It is the responsibility of the process itself to decide how to partition its own process graph; however, the smaller the local blocks are, the more concurrency the system can acquire and the more efficiency the process itself can have. If the result of partition were only a whole local block, this system would degenerate to, rather than serve as an improved version of, the original one.

The system then requires a new matrix LMAXC to record information about the Local MAXimum Claims.

<u>Definition</u> Matrix LMAXC,(m rows and n columns) consists of n m-dimension integer vectors LMAXC1 ...... LMAXCn,

LMAXC = (LMAXC1 ...... LMAXCn)

where LMAXCj, j=1..n, is defined by

$$\text{LMAXCj} = \begin{cases} 0 & \text{if Pj is in free state;} \\ \text{the local maximum claims,} & \text{if Pj is in the up state;} \\ \text{ALLOCj,} & \text{if Pj is in down state.} \end{cases}$$

Obviously,

R2': ALLOCj <= LMAXCj for all j=1..n

should be always true. R2' is more restricted than R2, but reasonable.

<u>Definition</u> A system state (AVAIL, LMAXC, ALLOC(t), REM(t)) is <u>quasi-realizable</u> if R1, R2' and R3 hold.

<u>Definition</u> A system state is <u>quasi-safe</u> if it is quasi-realizable and there is at least one sequence S' of all processes such that

$$\text{R4': LMAXCs(i) - ALLOCs(i)} \leq \text{REM(t)} + \underset{k<i}{\text{SIGMA}}\ \text{ALLOCs(k)(t)}$$

for all i=1..n, where R4' is the condition parallel to R4; i and k are indices in S.

With these notations, the theorem about deadlock avoidance parallel to theorem 1 in section 1 can be proved. Such a proof, first, requires:

<u>Lemma 1</u> If in a system, all processes are in a free state, then the system is in a safe state.

Proof: At this time, P1, P2, ..., Pn is a sequence leading all processes to completion, hence, the system state is safe. []

<u>Lemma 2</u> If the system is in a quasi-safe state, then there is a full sequence Q, following which each process is able to attain a free state.

Proof: Sequence Q is nothing more than the one found in the definition of quasi-safe state. []

<u>Theorem 4</u> The system will not become deadlocked if two conditions hold:

1. the system is in a quasi-safe state; and
2. the scheduler will keep at least one of the sequences which leads every process to a free state; and keep at least one of the sequences which leads processes running from their free state to completion.

Proof: By means of Lemma 2 and the condition regarding the scheduler in this theorem, every process can be free. Then the result follows from Lemma 1 and Theorem 1. []

This improved system retains the essential framework of the original one, although it differs in some respects from it. It functions in this way:

1. Processes need to report to the system their state transitions; in addition, when they get to up state, they must report their local maximum claims, which will be updated in LMAXC;

2. Each time when a process requests some resource(s), and sufficient resources exist in the system, the system state resulting from the proposed allocation must be tested to determine whether or not it is quasi-safe. If it is not, then this request is rejected and the process may try at a later time;

3. If a process Pj transits into a down state, the LMAXCj will be updated to be equal to ALLOCj. During the down state, when it releases some resource(s) then LMAXCj will be decreased correspondingly. A check of the system state is, however, unnecessary;

4. When a process gets into a free state, LMAXCi will decrease to 0 immediately;

5. Starting with all ALLOCj = 0, the system is in a quasi-safe state. It remains in a quasi-safe state, always avoiding the non-quasi-safe state.

The theorems, parallel to Theorem 2 and Theorem 3 in section 2, are stated below, and because the proofs are similar, they are omitted here.

<u>Theorem 5</u> If a subsequence T fulfills condition R4' and it cannot be extended into a full quasi-safe sequence, then the system state is not quasi-safe.

<u>Theorem 6</u> The system state after an allocation is quasi-safe if the original state is quasi-safe and Pk, which makes its request to cause this allocation, appears in one of

the sub-quasi-safe-sequences.

Thereby the computation cost of the test for the safety of the system following an allocation is similarly reduced as in section 2.

We conclude this section with the proof of our claim at the beginning of this section: This improved system offers a higher degree of concurrency than does the original one.

<u>Lemma</u> <u>3</u> If the system is in a safe state, then it is in a quasi-safe state also.

Proof: Because LMAXCi <= MAXCi for all i=1..n,

$$\text{LMAXCi-ALLOCi} \le \text{MAXCi-ALLOCi} \quad \text{for all } i=1..n.$$

Since the system state is safe, there is a safe sequence S such that

$$\text{MAXCs}(i) - \text{ALLOCs}(i) \le \text{REM} + \underset{k<i}{\text{SIGMA}}\ \text{ALLOCs}(k)$$

for all i = 1..n. Then

$$\begin{aligned}\text{LMAXCs}(i) - \text{ALLOCs}(i) &\le \text{MAXCs}(i) - \text{ALLOCs}(i)\\ &\le \text{REM} + \underset{k<i}{\text{SIGMA}}\ \text{ALLOCs}(k)\end{aligned}$$

for all i = 1..n, therefore S is the sequence fulfilling R4'. It follows that the system state is quasi-safe. []

<u>Theorem</u> <u>7</u> In our new system, the concurrency is more than that in the original one.

Proof: When a process requests some resource(s), if after an allocation the system state is safe, then by the above Lemma, the system state is quasi-safe also. Therefore, the new system has at least equal concurrency with that of the original. Nevertheless, it is possible for the system to be in a quasi-safe state, and not in a safe state; therefore, we conclude that of the new system has more concurrency than the original one. []

In the following section, when some aspects of system states are discussed without an explicit connection with concurrency, the term "safe" may be understood as "quasi-safe" as well as "safe", unless it becomes necessary to make a distinction.

4. Improvement on the Efficiency

To improve the efficiency, it is necessary to find a more effective way to prove the safety of a system state, that is, whether a safe sequence exists. An existence proof may be constructive or non-constructive. The original algorithm given in section 2 is constructive. In the case of one type of resources only, a non-constructive algorithm was given by [Habermann 78]. Habermann also concluded that his non-constructive method for one type of resources could not be extended to the case of multiple types of resources. However, He also presented an improved constructive algorithm for multiple types of resources [Habermann 78]. Based on this improved algorithm and using the property of this particular problem, this section intends to provide a partial constructive algorithm for multiple-types of resources. Needless to say, our new algorithm must have a higher degree of efficiency than any of its predecessors.

The object of a constructive algorithm is to construct a safe sequence, that is, to arrange the processes in some sorted order. In computer science, sorting is used to arrange a group of unordered elements into order. Therefore this method is nothing more than sorting! The key of sorting in this case is the amount of resources that a process may need in the (near) future. Hence, the efficiency of the algorithm seems to depend on the efficiency of the sorting algorithm chosen. In fact, Habermann has chosen "heap sort", one of the quickest sort algorithms available, to be used when attempting to improve the efficiency of the algorithm of multiple-types of resources.

On the other hand, it should be noted that, the basic requirement of the safety of the system state is the existence of the safe sequence. It may not be necessary, however, to find such a full safe sequence to prove the existence of the safe sequence. In the meantime, it is important to observe that to insure that the system state always remains safe, the system state must be checked at and only at the time when a process makes a request. Therefore Theorems 2 and 5 and Theorems 3 and 6 are always applicable no matter what sort algorithm is chosen. Combining these facts with the techniques of "heapsort" produces the intended efficiency-improved algorithm for verifying the safety of a system state.

Data Structures:

In addition to the data structures already in the original system, m n-dimension arrays, H1, ...,Hm, and one m-dimension array SUM are needed to do the sort. Each Hi corresponds to a type of resources, Ri. The elements of Hi are processes. The processes will be partially ordered in "Hi"s. Processes will try to pass, in turn, H1, ...,Hm. A process, Pj, passing Hi means that there will be sufficient resource, Ri, to serve Pj at some point in the future. When a process, Pj, passes Hm, how Pj releases resources being acquired is simulated by adding ALLOCj to SUM, (SUM is initialized with REM); that is,

SUM = SUM + ALLOCj.

Consequently, the formal condition by which a process can pass Hi is:

R4" MAXCj[i] - ALLOCj[i] <= REM[i] + SUM[i].

This is a coordinate form of R4. The algorithm will try to let all process pass each Hi, i=1..m; and will terminate if either

1. no more process can pass Hm, or
2. Pk, the process making this request, has passed Hm.

The former condition indicates that system state is not safe; and the latter indicates a safe state.

One m-dimension vector, REQk, will be used, which is issued by process Pk when it requests some resource(s). REQk indicates the amount of each resource which Pk is requesting:

REQk[i] = the amount of resource Ri which Pk is requesting.

In formal terms:

Algorithm:

```
Procedure STATE_CHECKER(State: System state, Pk: process);

Begin
   Put all processes in H1;

   Heapsort H1;

   Checked := false;
   SUM := REM;

   While not Checked do
   Begin
      For i := 1 to n-1 do
         Begin
           Transfer those processes satisfying R4"
           from Hi to Hi+1;
```

```
               Heapsort Hi and Hi+1;
            End;

         Transfer those processes in Hm satisfying R4"
         out of Hm,
         and when each of them, Pj, leaves Hm, do
            SUM := SUM + ALLOCj;

         If Pk passed Hm then
             Begin
                State := Safe;
                Checked := True;
             End;

         If No process can pass Hm then
             Begin
                State := Deadlock;
                Checked := True;
             End;
      End(*while*);

   End(*state_check*);


   Procedure ALLOCATOR;

   Begin
      If REM >= REQk then
         Begin
            Update ALLOCk as if the resource had been
            allocated to Pk upon its request;

            STATE_CHECKER(State, Pk);

            If State = Safe then
               Allocation
               else
               Reupdate ALLOCk to the original one;
         End
         else
            take some proper action such as blocking Pk;
   End(*Allocator*);
```

As a subprocedure of ALLOCATOR, STATE_CHECKER checks the system state. ALLOCATOR will be called when a process, Pk, makes a request for some resource(s). ALLOCATOR will decide whether the resources are allocated to Pk; if the decision is positive, the allocation will be carried on.

For detail about the heap sort procedure, refer to [Aho 1974] and [Habermann 1978].

5. Relation between Deadlock Avoidance and the Scheduler

The system will not become deadlocked if the conditions of Theorem 1 hold. This section discusses the issue of the second condition: The scheduler must not reject at least one of the safe sequences. If the scheduler uses an improper priority policy, it may mistakingly reject all of the safe sequences. The following discussion will serve as an example of such a situation and provide an opportunity to explore the possible solutions.

Assume that a system contains two types of one-unit resources, R1 and R2, and three processes, P1, P2, P3. The process graphs of the processes are as follows:

```
R1+-------             :                    :      R1+-------
    R2+-------         :   R2+---           :  R2+-------
  t0  t1  t2  t3       :     t0  t1         :    t0  t1  t2  t3
                       :                    :
  P1's Graph           :   P2's Graph       :    P3's Graph
```

Figure 2   Graphs of Processes P1, P2 and P3

The time domains of processes are independent, so that the same time notations used in the above different process graphs do not necessarily mean those events must happen simultaneously.

Assume the system will let the processes run according to the following sequence: P1, P2, P3, P1,..., if possible. If three units of the time have passed since the system started, the current system state is:

$$REM = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$ALLOC = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$GOAL = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Ready: P1, P2
Block: P3;

Then P1 takes its turn to run. Because of the lack of R2, P1 becomes blocked. Now P2 runs, releasing R2 and completing its run. When R2 is released, some process blocked on R2 may be awakened. If the system takes the FIFO awakening policy, P3 will naturally be chosen to wake up. When P3 runs, and requests R2, a problem presents itself. Since the allocation R2 to P3 would cause deadlock, this request is rejected. However, at present, P3 is the only process eligible to run, it runs again, and is, of course, rejected again. This will continue indefinitely, unless some special action is taken. The system is thrashing! Neither P1 nor P3 can complete themselves and the system is deadlocked!! Notice that now a safe sequence, P1-P2, does exist, but the scheduler is unable to follow the only safe sequence because of its FIFO awakening policy.

To solve this problem, the scheduler has at least two alternatives:

1. wake up all processes blocked by the resource when that resource is available;
2. sort the processes, and select the relatively first blocked process in the safe sequence to wake up when the resource becomes available;

Each of these methods eliminates the possibility of causing all safe sequences to be rejected by improper block/awaken priority policy. This is accomplished, however, at the expense of efficiency because more work has to be done by the scheduler.

6. Conclusions and Suggestions

This final section concludes the discussion of Habermann's algorithm for deadlock avoidance and suggestions for possible improvements on it.

Improvement in concurrency, made in section 3, is based on the fact that processes are able to provide more information, that is, more approximate information, about their requirements for resources at run time. If the processes cannot do so, and are only able to provide the "global maximum claims" then our improved system would degenerate to the original one. In this sense, the new system is compatible with the original one. It is interesting, however, that in modular programming, processes may more easily provide the local maximum claims than the "global" maximum claims. It appears impossible to avoid deadlock dynamically if processes are unable to provide any information about their resource requirements before making requests.

One suggestion for the practical implementation of the concurrency-improved system is to weaken the criteria of process's up state and down state in a local block. In other words, allow a process running in a local block to declare its "local maximum claim" at any point between the beginning point and the up point of the local block, where it makes its first request. Further, allow a process running in a local block to report "entering down state" at any point between the down point and the completed point, where it makes the last release in the local block. Obviously, the overhead of this proposal lowers the concurrency of the system, but it seems far easier to implement than does the theoretical method.

The principle of improving the efficiency does not depend upon any specific sorting algorithm. For this reason, this principle may be applied to any sort algorithm that may be developed in the future. Essentially, this study, in fact, suggests only a "partial" constructive algorithm for the proof of the existence of a safe sequence. Until a better non-constructive algorithm is developed, it is hoped that what this study proposes will prove useful.

Either the improvement in concurrency or efficiency implicitly suggests that the further work in this area will involve the specification of process, that is, a proper specification of process in some form may effectively direct processes to provide the information about resource claims needed by system, and consequently, reduce the difficulty in deadlock avoidance.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. [1974] "The design and analysis of computer algorithm." Addison-Wesley Publishing Company.

[2] Beauquier, J., and Nivat, M. [1980] "Application of Formal Language Theory to Problems of Security and Synchronization." Formal Language Theory [A. Book eds] Academic Press, (1980) p407-454.

[3] Coffman, E.G., Elphick, M.J., and Shoshani, A. [1971] "System deadlocks." Computer Survey 3, 2 (June 1971), p67-78.

[4] Deitel, H.M. [1983] "An introduction to operating systems" Addison-Wesley Publishing Company (1983).

[5] Dijkstra, E.W. [1965] "De Bankiers Algorithme." EWD116, Math. Dept., Technological Univ., Eindhoven, The Netherlands.

[6] Dijkstra, E.W. [1965] "Co-operating sequential processes." EWD123, Math. Dept., Technological Univ., Eindhoven, The Netherlands.

[7] Dijkstra, E.W. [1966] "The structure of the THE multiprogramming system." CACM 4,3 (Mar. 1966), p143-155.

[8] Habermann, A.N. [1969] "Prevention of system deadlocks." CACM 12,7 (July 1969), p373-377, 385.

[9] Habermann, A.N. [1974] "A new approach to avoidance of system deadlocks." Operating Systems (E. Gelenbe and C. Kaiser, eds.), Lecture Notes in Computer Science, Vol 16, Springer-Verlag, p163-170.

[10] Habermann, A.N. [1978] "System Deadlocks." Chapter 7 of Current Trends in Programming Methodology, Vol 3 (K. Mani Chandy and T. Yeh, eds.), Prentice-Hall (1978), p256-297.

[11] Havender, J.W. [1968] "Avoiding deadlock in multitasking systems." IBM syst. J. 7,2 (1968), p74-84.

[12] Holt, R.C. [1971] "On deadlock in computer systems." Ph.D. Thesis, Cornell Univ., (Jan. 1971).

[13] Holt, R.C. [1971] "Comments on prevention of system deadlocks." CACM 14,1 (Jan. 1971), p36-38.

[14] Holt, R.C. [1972] "Some deadlock properties of computer systems." Computing Surveys 4,3 (Sept. 1972) p179-196.

[15] Minoura, T. [1982] "Deadlock avoidance revised." J. ACM 29,4 (Oct. 1982) p1023-1048.

[16] Parnas, D.L. and Habermann, A.N. [1972] "Comment on a deadlock prevention method." CACM 15,9 (Sept. 1972), p840; with reply by R.C. Holt, p840-841.

[17] Spector, Alfred Z. and Schwarz, Peter M. [1983] "Transaction: A Construct for Reliable Distributed Computing." Operating Systems Review 17,2 (April 1983),

p18-34.

[18] Sugiyama, Y., Araki, T., Okui, J., and Kasami, T. [1977] "Complexity of the Deadlock Avoidance Problem." Transactions of the Institute of Electron. Comm. Eng. of Japan 60,4 (April 1977), p251-258.