A CONCURRENT ALGORITHM FOR AVOIDING DEADLOCKS IN MULTIPROCESS MULTIPLE RESOURCE SYSTEMS

Rafael O. Fontao

Visiting Scholar *
Digital Systems Laboratory
Stanford University

Summary:

In computer systems in which resources are allocated dynamically, algorithms must be executed whenever resources are assigned to determine if the allocation of these resources could possibly result in a deadlock, a situation in which two or more processes remain in an idle or blocked state indefinitely.

In previous research, execution of the process requesting resources is suspended while an algorithm is executed to determine that the assignment could not cause a deadlock. In this paper, an algorithm is used to calculate all possible safe requests before they are made. This algorithm is executed concurrently with other processes between requests for resource allocations. If the determination of all safe requests has been completed and a process makes a request, the calculations required by the resource allocation are trivial.

In order to use this algorithm it is necessary to have a priori knowledge of the maximum resource requirements of each process. This is a standard requirement for deadlock avoidance algorithms (dynamic avoidance). In addition, requests are restricted so that a process may request only units from a single resource class. Given these requirements, the determination of the safe requests can be formed with a complexity of second-order in n, where n is the number of processes in the system.

A scheduler using this algorithm can reduce the in-line system time if there are sufficient idle CPU cycles available between requests to execute the algorithm as, for example, in the case of an I/O bound system. In addition effective deadlocks may be prevented and non-sequential processes can be accepted.

I. Introduction

In multiprocess multiple resource systems, if certain behavioral assumptions on the allocation of resources persist, a deadlock situation may arise in which two or more processes remain in an idle or blocked state indefinitely.

Solutions to the deadlock problem have been classified as prevention techniques and detection and recovery techniques[5]. For deadlock prevention, the scheduler allocates resources so that the deadlocks will never occur. With deadlock detection and recovery, the scheduler gives resources to the processes as soon as they are available and when a deadlock is detected the scheduler preempts some resources in order to recover the system from the deadlock situation. Deadlock detection and recovery techniques[6] will not be considered here. Prevention techniques, have been grouped according to two major classifications: static and dynamic[5].

Certain conditions can be shown to be necessary if deadlocks are to occur. By restricting the behavior of the processes so that one of the necessary conditions for the occurrence of a deadlock is violated, deadlocks will never occur. This approach[2,4] has been called static prevention[5], since the rule for allocating resources does not depend upon the current state of the system. Dynamic prevention methods, on the other hand, attempt to allocate resources depending upon the current state of the system. These methods lead to a better resource utilization; however, they need some information about the resource requirements for each process. The algorithm presented in this paper is a new approach for dynamic prevention.

II. Preliminaries

The resource allocation system:

A resource allocation system (RAS) is formed by a set of independent processes $P_1, P_2, \ldots, P_n$ ($n \geq 1$), a set of different types of resources $R_1, R_2, \ldots, R_m$ ($m \geq 1$) each with a fixed number of units (two units are considered to be equal if they can perform the same task), and a scheduler that allocates the resources to the processes according to certain rules fulfilling some

* The author is a Teaching Assistant at the Electrical Engineering Dept., Universidad Nacional del Sur, Bahía Blanca, Argentina. This paper was supported by a grant of Consejo Nacional de Investigaciones Científicas y Técnicas of República Argentina.

specified criteria.

The system state:

The system state of an RAS is defined by a 3-tuple ($W,A,\overline{f}$) where;

1) $W = (\overline{w}_1,\overline{w}_2,\ldots,\overline{w}_n)$ is the want matrix (nxm). The entry $W(i,j) = \overline{w}_i(j)$ is the maximum number of additional units of resource R_j that the process P_i will need at one time to complete its task. $\overline{w}_k$ is the want vector for the process P_k.

2) $A = (\overline{a}_1,\overline{a}_2,\ldots,\overline{a}_n)$ is the allocation matrix (nxm) The entry $A(i,j) = \overline{a}_i(j)$ is the number of units of resource R_j allocated to process P_i. $\overline{a}_k$ is the allocation vector for the process P_k.

3) $\overline{f}$ is the free resource vector. The jth component is the number of available units of the resource R_j.

When A = 0 (matrix full of zeros) the system is in the initial state. In that case D = W is called the demand matrix and $\overline{c} = \overline{f}$ is the system capacity vector.

Basic assumptions:

1) Before it enters the system, a process is required to specify for each resource the maximum number of resource units it will ever need.

2) If a process is allocated resources, only the process can release them i.e., there is no preemption.

3) If a process is allocated all of the claimed resources it will release them after it completes its task.

4) The demand vector of every process is less than or equal to the system capacity vector.

Definition I: A sequence of processes $P_{e(1)}P_{e(2)}\cdots P_{e(k)}$ is called a terminating sequence for ($W,A,\overline{f}$) (where e(j) is the index of the process in the jth place) if:

$$1)\ \overline{w}_{e(1)} \leq \overline{f} \quad \text{and,}$$

$$2)\ \overline{w}_{e(i)} \leq \overline{f} + \sum_{1}^{i-1}{}_j\ \overline{a}_{e(j)} \quad \text{for } 1< i \leq k.$$

A terminating sequence is called complete if for all P_i there exists j ($1 \leq j \leq k$) such that e(j) = i. In other words, all processes are in the sequence.

Definition II: The system state ($W,A,\overline{f}$) is safe if there exists a complete terminating sequence for it.

In other words, the system is in a safe state if there is a way to allocate the resources claimed by the processes so that all of them can finish their task.

A. N. Habermann[1] has shown that when no process releases resources until the end of its exemption,processes will not get into deadlock if and only if the allocation state (or system state) is safe.

He also proved the following important theorem,

Theorem (Habermann)

Let ($W,A,\overline{f}$) be a safe state, and ($W',A',\overline{f}'$) the transformed state after a request by the process P_i is granted. If there exists a terminating sequence for ($W',A',\overline{f}'$) containing P_i, then the system state defined by ($W',A',\overline{f}'$) is safe.

The scheduler for Habermann's method works as follows: when a process wants additional resources, it calls the scheduler and goes into a wait state, the scheduler then decides if granting the request could cause a deadlock. If not, the process gets the resources requested. Otherwise, it remain waiting and later its claim is reconsidered by some scheduling rule. Figure 1 shows the timing of the procedure. It is assumed that only one process requires additional resources

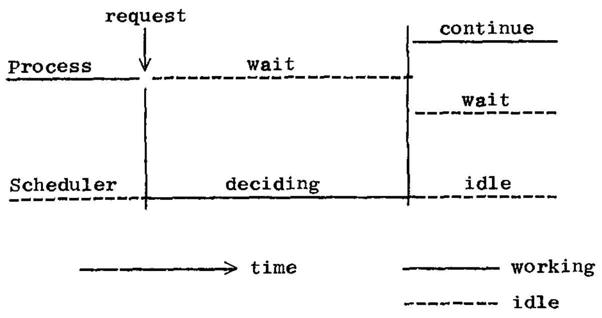


Figure 1

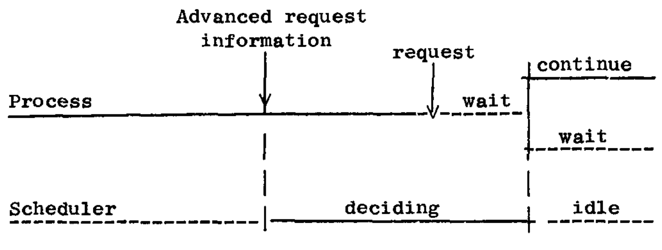
Timing of the use of process and scheduler in a previous method.

In a parallel sense, dynamic prevention methods may be classified as concurrent and non-concurrent, depending upon the concurrency of the working time between the process, involved in a request, and the scheduler.

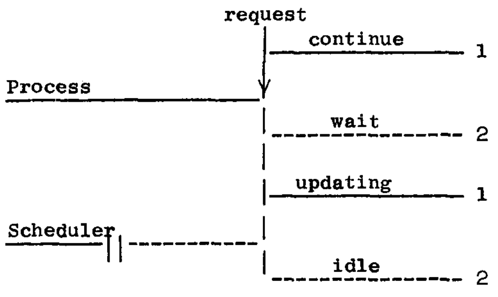
III. On Concurrent Algorithms

In order to devise concurrent algorithms, two alternatives may be considered: 1) know as early as possible when a request will be made, then run the scheduler concurrently with the process (concurrency-before-request), and 2) know before a request is made if it can be granted safely. If the request is granted, then run concurrently the process (continuing its task) and the scheduler (updating for new requests) (concurrency-after-request).

The figures 2(a) and 2(b) show the timing behavior of these alternatives The first alternative uses additional information (advanced request) and it will not be considered here. The algorithm that this paper deals with uses the second alternative.



(a)



1: request granted, 2: request denied

(b)

Figure 2

2(a); Timing of concurrency-before-request.
2(b); Timing of concurrency-after-request.

At first glance, it seems to be a tremendous task since all possible requests for each resource and for each process must be dealt with. However, if the requests are restricted to be single, i.e., for only one type of resource, then the task can be done with the same complexity as Habermann's method; second-order in n.

The safe request matrix

Clearly each process can safely request (single request) only a finite number of units (possibly zero). Let R be a matrix (nxm) defined by: the entry R(i,j) is the maximum number of units of resource j, that can be granted safely if the process i requests them. The matrix R is called the safe request matrix.

The single request restriction: An example.

Let (W,A,$\bar{f}$) be the state of a system defined by: (3 processes and 2 types of resources)

$$W = \begin{vmatrix} 1 & 0 \\ 2 & 2 \\ 0 & 1 \end{vmatrix} \quad A = \begin{vmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{vmatrix} \quad \bar{f} = (\ 1,1\ )$$

The system is in a safe state since $P_1P_3P_2$ is a complete terminating sequence. If P_2 requests only one unit of R_1, the system will be in a safe state since in that case $P_3P_1P_2$ is a terminating sequence containing P_2. Thus R(2,1) = 1. Similar, if P_2 requests only one unit of R_2 ($P_1P_3P_2$ would be a terminating sequence containing P_2). Thus R(2,2) = 1. However, if P_2 requires at the same time one unit of R_1 and one unit of R_2, the request can not be granted safely.

This example shows that the matrix R can be used only for single requests. If a process requires more than one resource, a rule stating the order in which the resources will be requested must be defined.

When a request of q units of a resource can be granted safely, in fact, it means that any request of k units ($k \leq q$) can be granted safely. Thus if the matrix R is known at the time a request is made, the process requesting only needs to check if the number of units claimed, is less than or equal to the corresponding entry in the matrix R. If the test is positive the process gets the claimed resources and continues its task. In this case, the scheduler runs concurrently (see Fig. 2(b)) in order to form the matrix R for the transformed state. If the test is not positive then; 1) the process gets the maximum number of units, given by the entry at R, and re-requests the difference later, or 2) it is put into the wait state.

The approach 1) may be used to prevent the effective deadlocks pointed out by R. Holt[3] and it will be discussed later. Now we present the algorithm.

IV. A Concurrent Algorithm

Let (W,A,$\bar{f}$) be the current state of an RAS. For all q ($q \leq f(j)$) let $\bar{f}^j_q$ be a vector formed from the free resource vector as follows:

1) $f^j_q(k) = f(k)$ for all $k \neq j$ ($1 \leq j \leq m$)

2) $f^j_q(k) = f(k) - q$ for $k = j$.

In other words, $\bar{f}^j_q$ is the free resource vector after q units of resource j are deleted from $\bar{f}$.

Definition 1: A process P_i is in the set H^j_q (head set) if there exists a terminating sequence for (W,A,$\bar{f}^j_q$) containing P_i.

In other words, H^j_q is the set of processes which can be still completed even though q units of resource j are deleted from the free resource vector.

Lemma 1: Let P_i in H^j_q and $P_{e(1)}P_{e(2)}\cdots P_{e(k)}$ be a terminating sequence for (W,A,$\bar{f}^j_q$) where e(k) = i. For all t ($1 \leq t < k$) $P_{e(t)}$ is in H^j_q.

Proof: It follows directly from definition.

Definition 2: A process P_i is in the set T_q^j (tail set) if;

1) P_i is not in H_q^j and,

2) $\overline{w}_i \leq \overline{f} + \overline{h}_q^j$ where $h_q^j = \sum_{P_k \in H_q^j} \overline{a}_k$

Lemma 2: If P_i is in T_q^j then the transformed state after allocation of q units of resource j to P_i is safe.

Proof;

Let ($W', A', \overline{f}_q^j$) be the transformed state after allocating q units of resource j to P_i. In this case W' and A' differ from W and A only in the (i,j) entry as follows;

$$W'(i,j) = W(i,j) - q$$
$$A'(i,j) = A(i,j) + q$$

Let $P_{e(1)}P_{e(2)}\cdots P_{e(k)}$ be a terminating sequence for ($W, A, \overline{f}_q^j$) containing all processes in H_q^j. From Lemma 1 can be proved that such a sequence always exists. In fact, that sequence is terminating for the transformed state ($W', A', \overline{f}_q^j$) since by hypothesis P_i is not in H_q^j. From definition, condition 2) may be rewritten as,

$$\overline{w}'_i \leq \overline{f}_q^j + \overline{h}_q^j \qquad (1)$$

Therefore, the sequence $P_{e(1)}P_{e(2)}\cdots P_{e(k)}P_i$ (P_i in the tail) is a terminating sequence for ($W', A', \overline{f}_q^j$) since after $P_{e(k)}$ finishes its task it will release all its allocated resources and the free resource vector will be $\overline{f}_q^j + \overline{h}_q^j$, thus by (1) P_i can request all its claimed resources. Therefore by Habermann's theorem the system state ($W', A', \overline{f}_q^j$) is safe.

Q.E.D.

Theorem

Let ($W, A, \overline{f}$) be a safe state. If a process P_i requires q units of resource j, then the request can be safely granted if and only if $P_i \in H_q^j \cup T_q^j$.

Proof: <--) If $P_i \in H_q^j$ then by definition there exists a terminating sequence for ($W, A, \overline{f}_q^j$) containing P_i. Clearly the same sequence will be terminating for ($W', A', \overline{f}_q^j$) where q units of resource j, deleted from $\overline{f}$, are allocated to P_i. Thus by Habermann's theorem the request can be safely granted. If $P_i \in T_q^j$ by Lemma 2 the request can be safely granted.

--->) If the request can be safely granted, then there exists a terminating sequence for ($W', A', \overline{f}_q^j$) containing P_i. Let $P_{e(1)}P_{e(2)}\cdots P_{e(k)}$ be that sequence, where e(k) = i. If $P_i \in H_q^j$ then the theorem holds. Let

$$P_i \notin H_q^j \qquad (1)$$

Clearly for all t ($1 \leq t < k$) $P_{e(t)} \in H_q^j$ since $P_{e(t)}$ may be completed desregarding the q units of resource j taken out from the free resource vector. Let H'^j_q be the set of those $P_{e(t)}$ and $\overline{h}'^j_q = \sum_{P_{e(t)} \in H_q^j} \overline{a}_{e(t)}$

Since $H'^j_q \subseteq H_q^j$ and all components are non-negative numbers,

$$\overline{h}'^j_q \leq \overline{h}_q^j \qquad (2)$$

From hypothesis,

$$\overline{w}'_i = \overline{w}_i - (0,0,\ldots,\underset{\text{jth}}{q},\ldots,0) \leq \overline{f}_q^j + \overline{h}'^j_q$$

may be rewritten as, $\overline{w}_i \leq \overline{f} + \overline{h}'^j_q$ and by (2)

$$\overline{w}_i \leq \overline{f} + \overline{h}_q^j \qquad (3)$$

from (1) and (3) P_i is in T_q^j.

Q.E.D.

The above theorem states that the head and tail sets play an important role in constructing the matrix R, i.e., the objective. Clearly, R(i,j) is the maximum number q so that $P_i \in H_q^j \cup T_q^j$.

In order to facilitate the understanding of the algorithm, ALGOL-like versions to calculate these sets are presented first with a correctness proof, then a complete program for calculating the matrix R.

An algorithm for calculating the head set runs as follows:

```
BEGIN   H := $\phi$ ;
        P := { set of all processes } ;
      $\overline{acc}$ := $\overline{f}$ ;
    acc(j) := acc(j) - q ;
   NumberP := n ;
     FailP := 0 ;
 Loop:  BEGIN
             candidate := index member in P ;
          IF ( $\overline{w}_{[candidate]} \leq \overline{acc}$ ) THEN
            BEGIN  H := H $\cup$ { $P_{[candidate]}$ } ;
                   P := P - { $P_{[candidate]}$ } ;
                 $\overline{acc}$ := $\overline{acc} + \overline{a}_{[candidate]}$ ;
             NumberP := NumberP - 1 ;
               FailP := 0
            END
          ELSE FailP := FailP + 1 ;
          IF ( FailP = NumberP ) THEN GO TO Hdone ;
          GO TO Loop
        END
 Hdone: ( H = $H_q^j$ )
 END.
```

Correctness: H is intended to be the head set (initially empty), P is the set of potential candidates (initially is the set of all processes). The vector $\overline{acc}$ is used to simulate the free resource vector after each process terminating its task releases its allocated resources (initially $\overline{acc}$ is $\overline{f}$). After the instruction acc(j) := acc(j) - q is executed, $\overline{acc}$ will contain $\overline{f}^j_q$ as required to evaluate the head set H^j_q.
NumberP is used to point out the cardinallity of the set of potential candidates (at first it is n). FailP is used to count the number of processes failing to be in H^j_q after an increasing in the vector $\overline{acc}$. It is assumed that the rule for choosing "candidate" is done in such a way that no failing process will be chosen again, unless an increase in $\overline{acc}$ has occurred. Initially FailP is set to zero. Therefore the initialization of the algorithm is correct. Now, assume that H, P, $\overline{acc}$, NumberP and FailP are true before enter to Loop. At this point the set of potential candidates is not empty (clearly at first $P \neq \phi$). The fact that the want vector of the candidate may be satisfied by the free resource vector ($\overline{w}_{[candidate]} \leq \overline{acc}$) means that this particular process can be completed disregarding the q units of resource j deleted from $\overline{f}$, in other words it is a member of H^j_q. Thus it must be added to H and deleted from P. Since the process will release all of its allocated resources, they are summed to $\overline{acc}$. Clearly NumberP is one unit less and FailP is set to zero. When the test ($\overline{w}_{[candidate]} \leq \overline{acc}$) fails, the candidate may not be in H^j_q. In that case H, P, and NumberP remain the same, indeed FailP is increased by one. Thus the state of affairs is correct after the first IF instruction is executed.

If (FailP = NumberP) is true it means that all members in the set of potential candidates fail to be in the head set, then by Lemma 1 the set H^j_q (in H) has been calculated. Otherwise, there are (NumberP - - FailP) processes in P as possible candidates to be in H^j_q . Thus P is not empty when Loop is entered again. Since the last test does not change the state of H, P, NumberP and FailP, then they are true when Loop is entered again. Therefore the algorithm is correct.

Now an algorithm for calculating the tail set is presented. Since T^j_q must be evaluated after H^j_q the following algorithm runs after the algorithm for calculating H^j_q. Therefore it is assumed that this algorithm is inserted at the label Hdone of the previous one.

```
BEGIN        T := $\phi$ ;
  BEGIN
             acc(j) := acc(j) + q ;
    WHILE ( P $\neq \phi$ ) DO
        BEGIN
             candidate := index member in P ;
          IF ( $\overline{w}_{[candidate]} \leq \overline{acc}$ ) THEN
                        T := T $\cup$ { $P_{[candidate]}$ } ;
          P := P - { $P_{[candidate]}$ }
        END
  END
Tdone:  ( T = $T^j_q$ )
END.
```

Correctness: By previous calculation (evaluating H^j_q) q units of resource j were deleted from $\overline{f}$. The instruction acc(j) := acc(j) + q is intended to restore them. Assume that $P \neq \phi$ (if $P = \phi$ so is T^j_q), clearly a candidate is chosen from the set of processes not in H^j_q (condition 1 from definition). If the test ($\overline{w}_{[candidate]} \leq \overline{acc}$) is true (condition 2 from definition) the candidate is a member of T^j_q, otherwise it is not a member of T^j_q and in both cases it must be deleted from P since in this algorithm $\overline{acc}$ is never increased. Therefore the algorithm is correct.

An algorithm for constructing the matrix R

This algorithm uses a logical matrix related to R as follows: the semaphore matrix S is a logical matrix (nxm) where S(i,j) = true if R(i,j) has been calculated by the algorithm, and it is false otherwise. Furthermore a new vector has been introduced; the surplus vector . The surplus vector of an RAS is defined by; for j ($1 \leq j \leq m$) surplus(j) = q where q is the maximum number so that H^j_q = { set of all processes } . For any particular state, the jth component of this vector shows the number of "surplus" units of resource j i.e., it shows the number of units of resource j that can be deleted from the system without introducing a deadlock. Clearly, for deleting resource units according to the surplus vector information, the single resource restriction still holds.

The following algorithm calculates the matrix R, the surplus vector and the safeness for a system state. It may run each time there is a change in the system.

```
Repeat:   FOR j = 1 STEP 1 UNTIL m DO   PARALLEL
BEGIN
        $\bar{s}(j)$ := $\bar{f}$alse ;       ( vector full of false's )
        $\bar{r}(j)$ := $\bar{0}$ ;
   surplus(j) := 0 ;
           T := $\phi$ ;
           H := $\phi$ ;
           q := f(j) ;
         $\overline{acc}$ := $\bar{f}$ ;
     NumberP := n ;
  WHILE ( q $\geq$ 0  and  H $\neq$ {set of all processes})DO
  BEGIN
        acc(j) := acc(j) - q ;
             P := {set of all processes} - H ;
           FailP := 0 ;
Loop:  BEGIN
        candidate := index member in P ;
      IF( $\bar{w}_{[candidate]} \leq \overline{acc}$ )  THEN
        BEGIN
            H := H $\cup$ { $P_{[candidate]}$ } ;
            P := P - { $P_{[candidate]}$ } ;
          $\overline{acc}$ := $\overline{acc}$ + $\bar{a}_{[candidate]}$ ;
        NumberP := NumberP - 1 ;
          FailP := 0 ;
        IF( S( candidate,j ) = false ) THEN
            BEGIN    R( candidate,j ) := q ;
                     S( candidate,j ) := true
            END
        END
      ELSE    FailP := FailP + 1 ;
      IF( FailP = NumberP ) THEN GO TO Hdone ;
      GO TO Loop
    END
Hdone:  BEGIN
          P := P - T ;
      acc(j) := acc(j) + q ;
      WHILE ( P $\neq$ $\phi$ ) DO
        BEGIN  candidate := index member in P ;
          IF( $\bar{w}_{[candidate]} \leq \overline{acc}$ ) THEN
            BEGIN
                         T := T $\cup$ {$P_{[candidate]}$};
            R( candidate,j ) := q ;
            S( candidate,j ) := true
            END
          P := P - {$P_{[candidate]}$}
        END
      END
            q := q - 1
  END
    IF( H = {set of all processes} )THEN
                          surplus (j) := q + 1 ;
     ELSE
     " unsafe state, the processes not in H may be
                    blocked " ;
END .
```

Remarks: The algorithm starts calculating H = $H^j_{f(j)}$, then H = $H^j_{f(j)-1}$ and so on. When H = H^j_q, since for all q $H^j_q \subseteq H^j_{q-1}$, the algorithm attempts to evaluate H^j_{q-1} only considering "candidate" in the set of processes not yet in H^j_q. Furthermore, it may be easily proved that if a process P_t is in $H^j_q \cup T^j_q$, then P_t is in $H^j_{q-1} \cup T^j_{q-1}$ i.e., if P_t can safely request q units of resource j, it can safely request q-1 units as well. This fact has been considered in choosing "candidate" for evaluating the set T. Note that if a process is in T^j_q then it may be in H^j_{q-1} but if it is in H^j_q it is in H^j_{q-1} too.

The instruction "IF(S(candidate,j) = false) THEN ..." is intended to test if a process P_t just in H^j_{q-1} was in T^j_q. If the test is positive, it means that R(t,j) and S(t,j) had been evaluated.

The instruction "IF(H = {set of all processes}) THEN ...", just before the end of the algorithm, is intended to evaluate the surplus and the safeness of the current state as follows: if the algorithm terminates because H is the set of all processes before q $\geq$ 0 is false, then the system has a surplus of q+1 units of the resource being dealt (note that q was decreased by one). If the algorithm terminates because q $\geq$ 0 (by construction would be q = -1) is false, then there are two alternatives: 1) H = H^j_0 is the set of all processes, in this case the surplus is zero and it is correctly given by the algorithm, or 2) H = H^j_0 is not the set of all processes, then it means that there are processes (not in H) that could be deadlocked and the system state is unsafe.

The algorithm permits a parallelism of m, since each resource can be dealt independently.

Complexity

Block Loop; Suppose that the system is in a state so unfortunate that H^j_q is the set of all processes and there is only one complete terminating sequence for it. Let $P_{e(1)}P_{e(2)}\cdots P_{e(n)}$ be that sequence. Assume in the worse case that "candidate" is chosen in such a way that it is the inverse order of the sequence. At first Loop will be entered n times to find $P_{e(1)}$, then n-1 times to find $P_{e(2)}$, then n-2 times to find $P_{e(3)}$ and so on to $P_{e(n)}$. In other words, in the worst case Loop will be entered n + n-1 + n-2 + ... + 2 + 1 = n x (n+1)/2 times. Since the blocks inside Loop have constant complexity, the complexity of this block is second-order in n.

Block Hdone: Clearly each time the test (P $\neq$ ϕ) is true a member in P is deleted. Thus the maximum number of true tests is n, then the complexity of this block is first-order in n.

Each time the test ($q \geq 0$) is true q is decreased by one, thus the maximum number of times that this test is true ($1 + f(j)$) is not dependent on n. Therefore the complexity of the algorithm, in the worst case, is second-order in n.

Example: Let ($W, A, \bar{f}$) be the system state defined by; (4 processes and 2 resources),

$$W = \begin{vmatrix} 5 & 2 \\ 3 & 1 \\ 2 & 2 \\ 4 & 4 \end{vmatrix} \quad A = \begin{vmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 0 \\ 3 & 1 \end{vmatrix} \quad \bar{f} = (3,4)$$

the application of the algorithm for this state yields,

$$R = \begin{vmatrix} 0 & 1 \\ 3 & 4 \\ 3 & 4 \\ 1 & 3 \end{vmatrix} \qquad \overline{\text{surplus}} = (0,1)$$

If one unit of resource 2 is deleted from the system, then the response of the algorithm would be,

$$R = \begin{vmatrix} 0 & 0 \\ 3 & 3 \\ 3 & 3 \\ 1 & 2 \end{vmatrix} \qquad \overline{\text{surplus}} = (0,0)$$

If two units of resource 2 are deleted from the system i.e., the free resource vector would be (3,2), then the algorithm yields,

" unsafe state, the processes 1 and 4 may be blocked "

V. The Scheduler

The above algorithm is only a part of a scheduler for an RAS. In order to design a scheduler using this algorithm, the designer must consider:

1) A rule for deciding what process will be considered first if more than one is waiting for resources.
2) A rule for deciding what resource will be considered first if a process requires more than one resource class.
3) A rule stating what to do when a request (single) is denied, i.e., if the number of units requested is larger than the corresponding entry in the matrix R.

The first rule is considered by any scheduler, the second one introduces no additional problems and may be handled without difficulty. Any scheduler needs the third rule. In those schedulers using the previous algorithms, this rule has the following form: if a request is denied send it into a queue for a later attempt. The requests are kept in their original form. A scheduler using the algorithm presented here will have the following features:

Reservation of units (effective deadlock prevention)

Although, the number of units requested by a process in a resource is larger than the corresponding entry in matrix R i.e., the request can not be safely granted, the process can get, without introducing a deadlock, the number of units specified by the entry in R. The process can enter a queue and re-request the difference at a later time. In other words, a process requesting units of a resource completes its demand by collecting all available units for it, each time it uses the algorithm. In that sense the use of the algorithm is never unsuccessful. This "put in reserve" property prevents the effective deadlocks pointed out by R. C. Holt[3].

As was shown by Holt the scheduler may introduce deadlocks not prevented by methods using the safe state concept alone. When the requests are ordered in a queue by a FIFO discipline this kind of deadlock may arise. Fortunately, these deadlocks can be easily avoided by breaking the priority rule; the scheduler passes down the queue granting those requests which are safe.

More important, indeed, is the fact that a process may be blocked for a "long time" if its request is never granted because others processes claiming less units, make their requests in such a way that the units requested by the process are never available for it. In that case the process is a "victim" of an effective deadlock.

This necessary condition for an effective deadlock can be avoided if at the time a process requires units of a resource, it gets them according to the information given by the matrix R, instead of waiting for all of the required units. In other words, a process claiming a large demand will decrease it, by reserving all possible units, each time it uses the algorithm.

Consider the following example (from Holt[3]): assume a system containing only two units of a resource and three processes P_1, P_2 and P_3 demanding one, one and two units respectively. Assume initially that P_1 has allocated one unit and P_3 requests two units. Clearly, the request can not be granted, simply, because two units are not available. If P_2 requests one unit before P_1 releases its allocated unit, later P_1 releases it and re-requests the unit again before P_2 releases its unit and this behavior is mantained by the processes P_1 and P_2, then the process P_3 can never fulfill its task because at no time are two units available. In this case P_3 is "victim" of an effective deadlock.

In the notation of this paper the initial state is described by:

$$W = \begin{vmatrix} 0 \\ 1 \\ 2 \end{vmatrix} \quad A = \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} \quad \bar{f} = (\ 1\ )$$

The application of the algorithm to this example yields,

$$R = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix} \quad \overline{\text{surplus}} = (0)$$

If at the time P_3 requests two units, it gets at least one unit as given by the entry R(3,1), then the effective deadlock will never occur because P_3 will request later only one unit, just the same as P_2 will need.

Accepting non-sequential processes

Assume that there are processes in the system organized in such a way that their tasks depend on the availability of the resources. For example a process may want one unit, at least, but it will run faster if it gets more units. If the processes get the resources according to the information given by the matrix R, those processes can be accepted by the system and they will run with a resource utilization nearly optimal. Those processes must specify before entering the system the minimum number of units that they will need at one time in order to fulfill their tasks.

Added to the previous features, a scheduler using this algorithm will have the basic characteristic of the algorithm; it can reduce the in-line system time if there are sufficient idle CPU cycles available between resource requests to execute the algorithm as, for example, in the case of an I/O bound system.

VI. Conclusion

A new approach for avoiding deadlocks in computer system has been presented. By evaluating the safe requests before they are made, this method can increase the resource utilization, provided there are sufficient idle CPU cycles available between resource requests to execute the algorithm, as in the case of an I/O bound system.

With this approach effective deadlocks can be avoided and non-sequential processes can be accepted. Furthermore, the algorithm evaluates the units of "surplus" of the system. According to the definition, the surplus vector of a system shows the number of units of each resource that can be deleted from the system without introducing a deadlock. The information given by the surplus vector could be used by a scheduler of a network of computers to assign the resources among them.

Acknowledgment

The author wishes to thank Prof^s Edward J.McCluskey and Thomas H. Bredt of the Digital Systems Laboratory, Stanford University, for their helpful suggestions and discussions.

References

1) HABERMANN, A.N. Prevention of System Deadlocks. Comm. ACM 12,7 (July 1969), 373-377.

2) HAVENDER, J.W. Avoiding Deadlocks in Multi-tasking Systems. IBM Systems Journal 2,7 (1968), 74-84.

*3) HOLT, R.C. Comments on Prevention of System Deadlocks. Comm. ACM 14,1 (January 1971), 36-38.

4) MURPHY, J.E. Resource Allocation with Interlock Detection in Multi-task Systems. Proc. AFIPS 1968 FJCC, Vol. 33, Pt. 2, 1169-1176.

5) RUSSELL, R.D. A Model for Deadlock-Free Resource Allocation. Tech. Memos No. 93, 94 and 116 (June, Oct. and Dec. 1970 respectively), Stanford Linear Accelerator Center, Computer Group, Stanford University.

6) SHOSHANI, A. and E. G. COFFMAN Detection, Prevention and Recovery from Deadlocks in Multiprocess, Multiple Resource Systems. Tech. Rep. No. 80, Dept. of Elec. Eng., Comp. Sc. Lab., Princeton University, (Oct. 1969).

*Note: Reference 3 was published before in Tech. Report No. 70-5- (January 1970) Department of Computer Science, Cornell University, Ithaca, New York