CGTM 94
Bob Russell
October 1970

Model for Deadlock-Free Resource Allocation

Section 2: Linear Algorithms for Deadlock Prevention

I. Prevention of Deadlock

In CGTM 93 we presented a model of a resource allocation system that enabled us to formulate linear algorithms for dynamic deadlock detection. By judicious choice of the scheduler data base, these algorithms attained a significant advantage in speed over deadlock detection algorithms that have previously appeared in the literature. In this memo we will show how the same model can be used, almost trivially, to prevent deadlocks, rather than to merely detect them. As in CGTM 93, these results present linear (i.e. directly proportional to the number of jobs) algorithms to do the calculation, and are extendable to the previously unexplored case of simultaneous resource-sharing among jobs. Once again it is the advantageous choice of the Scheduler data base that gives us these results.

II. Static vs. Dynamic Prevention

The model for deadlock detection presented in CGTM 93 is dynamic in the sense that no advance information is required about the resource needs of the jobs, and no restrictions are made upon the jobs' use of resources insofar as the number and order in which they are requested, used, and released is concerned. The behavioral assumptions about the system are listed in Section III of that memo. As has been indicated by Shoshani, and Havender, certain conditions on the job behavior are necessary if a deadlock is to arise at all. Therefore, by suitably restricting the possible behavior of jobs in the system, such that any violation of the allowed behavior is cause for immediate termination of the job, one or more of these necessary conditions will be violated, and hence the possibility of deadlock does not exist. We will therefore call any scheme that restricts job (or resource) behavior in such a manner that one of the necessary conditions is violated and hence deadlock becomes impossible, a static prevention method. When one of these methods is employed, every possible state of the system, given any combination of jobs and resources, will be safe (i.e. free from deadlock), simply because the type of behavior that would lead to an unsafe state is not allowed. These methods will be discussed in a subsequent memo.

A <u>dynamic prevention</u> method is one which will operate a system in such a manner that it will never enter an unsafe state, even though job behavior is not restricted a priori to eliminate these states. If we consider all the possible allocation states of the system to be represented as nodes in a directed graph, with the edges being all possible transitions between states, then it is the task of a dynamic prevention method to determine, solely from the current state of the system and some (probably limited) amount of advance information, which transition to use to take the system into a safe next state (i.e. one that is neither in a deadlock nor will lead to a deadlock). In our model each allocation state is represented by the state of the scheduler data base, and a transition by the system from one allocation state to another occurs whenever a change is made in the scheduler data base. We have constructed the model so that only the scheduler, in response to either a REQUEST or a RELEASE primitive by one of the jobs, can make changes to the data base. Therefore we will aim for algorithms that operate as part of the normal scheduling primitives to prevent any changes in the data base that will lead to an unsafe state.

III. Other dynamic prevention models

In order to prevent deadlock in a dynamic method, some form of additional information about a job's future behavior or potential behavior must be available to the prevention algorithm. This information is supplied by the job at the time it enters the system, and it is expected that during its existence, the job will stay within the limits that it set for itself. Hence we have for each job a set of self-imposed restrictions on its behavior (rather than the system-imposed restrictions when a static prevention method is used). In Haberman's model, this information is in the form of the maximum number of elements of each resource class that will be needed simultaneously at any time in its existence. Since this information may not be known to the job until it has processed some data, each job must specify its need considering the worst possible case in all data dependent decisions. Since even this may not be known precisely, conservative estimates are often the only information that can be provided by the job in advance. Of course depending on the data, the job may never actually require the full maximum, or it may require it for only a very short length of time during the entire duration of the job. Hence specification of such maximums tends to lead an allocator into inefficient use of system resources in an effort to prevent deadlock in the (often unlikely) case that the maximum allocations are actually requested.

Shoshani's model attempts to improve the efficiency of the system resource allocator by requiring the job to specify in advance even more detailed information about its potential behavior. In his model, each job is required to specify its behavior as a series of discrete steps, each of which is of unknown duration but of known resource needs. The allocator can then piece together the small (and presumably less demanding of resources) steps from all jobs in a manner that utilizes the system resources more efficiently. In the degenerate case where each job consists of exactly one step, Shoshani's model is identical to Habermann's. There are several crucial assumptions worth noting about both of these models:

1) They both require information only about the number of resource elements needed by a job (in Shoshani's case, the job must specify a sequence of these numbers, corresponding to the sequence of job steps). However it is assumed that the time at which the actual need will arise, and the duration of the allocation, are unknown and unpredictable.

2) The information supplied must be accurate. Neither model considers the resource requirements to be probabilistic in any sense, and deadlock is prevented absoluately, rather than with only a high probability.

3) They both assume that a resource element can be allocated to,at most, one job at any time, and that once allocated, it cannot be deallocated by either the allocator or any other job until the job to which it is allocated, by its own volition, invokes a Release primitive to deallocate that specific resource element. Hence neither simultaneous resource sharing, of any kind, nor resource preemption, at any cost, is allowed.

4) All resources are considered to be "hard" resources which themselves do not require other resources of the system to be allocated to them in order to function. Hence there is no consideration of virtual resource mappings or multiplexing, and only one level of allocation is considered.

If time and duration of use as well as number of resource elements are known in advance, then the problem becomes one of optimization since complete information is now available. This then reduces a problem of operation research.

Models to avoid deadlock based on probabilistic predictions of need, rather than absolute limits do not exist as yet and might be a fruitful area for future research. Algorithms might be formulated such that the system would always be operated with a probability of deadlock less than some specified $\mathcal{E}$. Presumably these algorithms could be extremely fast and would be much more amenable to current systems where future resource requirements are often difficult to predict accurately.

All multiprogramming and multiprocessing systems violate assumption 3) since resource sharing, typically of data sets and core storage, and priority preemption, typically of the CPU resource, are a fundamental part of such systems. In a later section of this memo, we will present a dynamic prevention method that permits resource sharing and hence is more amenable to use in current systems.

Assumption 4) is violated by most time-sharing systems and many multiprocessing systems, since virtual resources that themselves compete for resources are becoming a more and more common phenomenon. Later we will also look into some of the implications of these systems for deadlock prevention schemes.

IV. Data Base B_3

The new data base B_3 is shown in Figure 3. It is identical to B_1 shown in Figure 1 of CGTM 93, but with the addition of matrix M.

We assume that at entry into the system, each job J_i specifies a "maximum resource demand" vector $M_i = [M_{1i}, M_{2i}, \ldots M_{mi}]$ such that at no time during its existence will job J_i simultaneously need more than M_{ji} resource elements of resource class R_j, j=1, 2, ..., m. This is the only advance information required of any job, and no assumption is made about the sequence in which resources are required, or the duration of their use by job J_i. In this data base we assume that no resource sharing is allowed. That is, all resource elements requested by J_i must be exclusively controlled by J_i during the time J_i has any access to them. The next section relaxes this unduly harsh restriction and permits resource sharing by otherwise unrelated jobs. Because of assumption 4) above, we must have $M[i,j] \leq RMAX[j]$ j=1, 2, ..., m. In addition to the vector M_i, which for now we have assumed remains fixed for the entire duration of job J_i in the system, there will also be two other vectors A_i and D_i that describe the current status of J_i in exactly the same manner as they did in CGTM 93. That is, A_{ji} is the number of resource elements of type R_j currently assigned to J_i, and D_{ji} is the number of resource elements of type R_j that must be assigned to J_i before J_i can proceed. Since M_i represents the maximum resource requirement of J_i, we must have

$$A[i,j] + D[i,j] \leq M[i,j] \leq RMAX[j]$$

for all i and j at all times. As before, if $D[i,j] > 0$ for any j, then job J_i is in the wait state for an allocation of $D[i,j]$ elements of resource class R_j. If $D[i,j]=0$ for all j and fixed i, then J_i is active.

We will also define a vector U_i such that

$$U[i,j] = M[i,j] - A[i,j]$$

for all i and j. Thus U_i represents the "unassigned" portion of the maximum possible resource allocations to J_i, and is therefore the maximum size of any future resource demand by job J_i, given its current allocation A_i. Obviously

$$D[i,j] \leq U[i,j]$$

for all i and j.

Because U_i can be derived from A_i and M_i for any i whenever needed, it is not represented in the data base B_3. It is, however, used throughout the algorithms to be described next.

We have also replaced DNUMB in B_1 with UNUMB in B_3, due to the fact that DNUMB is defined relative to the D matrix, and in the prevention algorithms the role of the D matrix will be replaced by the U matrix. Hence, we will not need DNUMB, but will define UNUMB[j] as the number of elements in the ith row of U that are greater than 0. We will also redefine WAITCØUNT relative to UNUMB rather than to DNUMB, as the number of UNUMB[i] > 0.

V. Algorithm L_4

As has been shown by Habermann, a system S is safe if any only if there exists a sequence F (which we will call a finishing sequence) of all jobs in S such that (renumbering the jobs so that J_k is the kth job in F)

$$\overline{U}[1] \leq \overline{RFREE}$$

$$\overline{U}[i] \leq \overline{RFREE} + \sum_{k=1}^{i-1} \overline{A}[k] \qquad i=2,3,\ldots, n$$

where all operations are considered to be m-component vector operations. This formula is similar to the one given in section V of CGTM 93, and can be obtained from that one by simply replacing $\overline{D}$ with $\overline{U}$ whenever it occurs.

The reasoning for this is obvious, since in CGTM 93 we assumed that for all jobs only the current unsatisfied demands D were known to be impeding the progress of those jobs. Once its demand D was satisfied, a job was expected to resume making progress, and nothing was known about any future demands by that job. Hence we had to assume that nothing else would be needed by that job beyond its current demand. In the present situation however, we know in advance that job J_i can at no time, either currently or in the future, make a demand greater than $\overline{U}_i$ (unless it happens to first release some resources, which of course it may or may not do). Hence by guaranteeing that the job can be sequenced so that the maximum possible demand could be met, we are assured that any actual demand less than or equal to that maximum will also be met.

Another way to look at this would be as follows: suppose at the current instant every job J_i were to simultaneously request all the remaining resources $\overline{U}_i$ which it declared it might need but has not as yet been assigned. Then $\overline{U}_i = \overline{D}_i$ for all i and the situation is identical to the one already discussed in CGTM 93, since if deadlock is not detected in this extreme situation, it cannot possibly be any worse at any time in the future and hence the system is safe for all future times as well.

As might well be expected, algorithm L_4 to prevent deadlocks is derived from algorithm L_1 to detect them by simply substituting $\overline{U}_i$ for $\overline{D}_i$ everywhere, and, since DNUMB was originally defined relative to $\overline{D}$, by replacing it with a corresponding UNUMB defined relative to $\overline{U}$. That is, as far as being able to ascertain whether or not a deadlock can ever occur, we need only consider the remaining potential demand $\overline{U}_i$ and can ignore the current demand $\overline{D}_i$.

Essentially the algorithm maintains its "deadlock detection" characteristics but only as applied to the worst possible demand at any future time rather than simply to the current demand situation. The rules for applying this algorithm to prevent deadlock as part of the Request and Release primitives must also be modified somewhat. However, as was demonstrated by Habermann, finishing sequences F for systems with known maximum resource demands also exhibit the "early cutoff" property described in section VII of CGTM 93. Hence we can incorporate the modified BUMP procedure from L_2 into L_4 directly. The final algorithm L_4 is as follows:

1) We assume the ordering rule Z is defined so that each column j of Q is ordered according to <u>increasing</u> numbers of resource elements of type j that remain for <u>potential</u> demand by the job. That is:

   U[Q[i,j],j] ≤ U[Q[i+1,j],j]

   for i=1, 2, ..., QSIZE[j]-1

2) Initialization:

   MARK[j] ← 1   j=1, 2, ..., m

   ACC[j] ← 0   j=1, 2, ..., m

   WC ← WAITCOUNT

   DN[i] ← UNUMB[i]   i=1, 2, ..., n

   THRESH[j] ← RFREE[j] + $\sum_i$ A[i,j] such that UNUMB[i]=0 j=1, 2, ..., m

3) The algorithm:

```
        REPEAT:  for  j := 1 until m do
        L1:           begin parallel
                      I := Q[MARK[j], j] ;
        L2:           while (MARK[j] ≤ QSIZE[j] ) AND
                                 (U[I,j]  ≤  THRESH[j])
                                 do begin
```

```
L3:                        BUMP(I);
                           MARK[j] := MARK[j] + 1;
                           I := Q[MARK[j], j] ;
                           end
                end parallel
L4:       if WC = 0 then  < no deadlock >
      else if ACC = 0 then < deadlock >
      else    begin
              THRESH := THRESH + ACC;
              ACC := 0;
              go to REPEAT;
              end
```

Since procedure BUMP does not involve either D or U, it is identical to that on page 21 of CGTM 93, and is reproduced here for completeness only.

```
Procedure BUMP (integer value I);
      begin
      DN[I] := DN[I]-1;
      if DN[I] = 0 then
          begin
          if I=K then  < stop, no deadlock > ;
          ACC := ACC + A[I];
          WC := WC-1 ;
          end ;
      end;
```

VI. Scheduler Primitives

As before, our objective is to maintain the system at all times in a deadlock-free condition. Now with the additional advance information about maximum resource demands, it becomes possible to guarantee that an allocation will never be made that leads to a deadlock situation. In the case of no advance information, deadlocks could be detected only at the instant when they occurred, that is, when a job made a request that could not be satisfied from the current "free resources" pool and it had to be placed into the wait state. If a deadlock did not exist previously, the entrance of a job into wait state might have created one, and the deadlock detection algorithm L_2 had to be executed to decide. However, if the demand could be satisfied, a new deadlock was impossible.

With advance information however, it is not the refusal of a demand due to insufficient free resources but rather the assignment of free resources to satisfy a current demand that can create a deadlock situation where none existed previously. Although this may seem strange at first, it becomes obvious when we note that the prevention algorithm L_4 is defined in terms of $\overline{U}_i$, not $\overline{D}_i$, and $\overline{U}_i$ changes only when $\overline{A}_i$ changes, i.e., when an allocation or deallocation is made. In other words, the prevention algorithm foresees the possibility of any demand $\overline{D}_i$ less than or equal to $\overline{U}_i$, so that the actual occurrence of such a demand does not change the safeness of the situation in any way. However, the satisfaction of a demand, either at the time it is made or some later time when resources became available, changes the remaining potential demand of that job and reduces the resources available to the system for satisfying potential demands of other jobs in the system. We therefore must execute algorithm L_4 to see if the reduced resource availability is still sufficient to satisfy the remaining potential demands of all the jobs according to at least one sequencing F.

With this in mind we can proceed to redefine the Request and Release primitives as defined in section VI of CGTM 93.

A. Request ($\overline{N}$) where $\overline{N}$ is the m-component vector that specifies the new resource elements needed by J_i. We assume $\overline{N} \leq \overline{U}_i$, in each component, since otherwise we would be violating the assumption that M_i is the maximum possible demand of J_i.

1) If $\overline{N} \leq \overline{RFREE}$ (in each component) then proceed as follows:

a) $\overline{A}_i \leftarrow \overline{A}_i + \overline{N}$; (so that $\overline{U}_i \leftarrow \overline{U}_i - \overline{N}$;)

b) $\overline{RFREE} \leftarrow \overline{RFREE} - \overline{N}$;

c) For each component j of $\overline{N}$ that is > 0, re-order the jth column of the Q matrix (which is ordered by rule Z on the value of U_i) by removing J_i from its current position and re-inserting it according to rule Z.

d) Execute the sequence finding algorithm L_4 to find a finishing sequence F. If it succeeds, then no deadlock has been created and the system is still safe. In this case, we must invoke the Allocate primitive for each resource element requested by $\overline{N}$, and then read a "go ahead" signal back to J_i.
If algorithm L_4 fails, then this allocation would have created the deadlock and must not be made. Therefore we must "undo" it by

[1] restoring $\overline{A}_i(\overline{U})$, and $\overline{RFREE}$ to their original values

[2] removing J_i from its new slot in the Q matrix and replacing it in its original position.

[3] to indicate that job J_i must now wait, we set $\overline{D}_i \leftarrow \overline{N}$ and add J_i to the system waiting list WAITQ according to ordering rule Y.

2) If it is not true that $\overline{N} \leq \overline{RFREE}$ in every component, then the job J_i must be placed into wait state at once. Since there is no possibility of an allocation being made, the checking algorithm need not be executed, and the position of J_i in the Q matrix will remaing unchanged (remember, the rule Z for ordering the columns of Q is based only on U_i and this only changes when an allocation is made). Hence we need only perform

a) $\overline{D}_i \leftarrow \overline{N}$;

b) add J_i to the system waiting list WAITQ according to ordering rule Y.

B. Release $(\overline{N})$ where $\overline{N}$ is the m-component vector that specifies the resource elements no longer needed by J_i. Obviously $\overline{N} \leq \overline{A}_i$ since J_i can only release resources already assigned to it.

1) a) $\overline{A}_i \leftarrow \overline{A}_i - \overline{N}$; (so that $\overline{U}_i \leftarrow \overline{U}_i + \overline{N}$;)

b) $\overline{RFREE} \leftarrow \overline{RFREE} + \overline{N}$;

c) For each resource element requested by $\overline{N}$ invoke the Deallocate primitive for that resource class.

d) For each component j of $\overline{N}$ that is > 0, re-order the jth column of the Q matrix by removing J_i from its current position and re-inserting it according to rule Z.

e) Send a "go ahead" signal back to J_i. Note that even though $\overline{U}_i$ changes and the queue matrix Q is partially reordered, the detection algorithm L_4 need not be executed, since it will be shown later that resources can always be released without endangering the safety of the system.

2) If there is any job in wait state, as indicated by the system wait queue WAITQ, then select the next job from WAITQ, say J_k.

a) If $\overline{D}_k \leq \overline{RFREE}$ in each component then proceed exactly as in steps a) - d) of item 1) under the Request primitive, but replacing i with k and $\overline{N}$ with $\overline{D}_k$ throughout.
In step d), if the algorithm L_4 succeeds, then we must also remove J_k from the waiting list WAITQ, whereas if it fails we can simply leave it in place (i.e. eliminate step d)[3]).
If this assignment is successful, we can repeat this step for the next job on WAITQ, say J_p.

b) If $\overline{D}_k > \overline{RFREE}$ in at least 1 component, or if algorithm L_4 fails, we can either terminate the scheduler operation at once or continue checking all jobs on WAITQ, depending on the rule Y used to order WAITQ.

VII. Deadlock Prevention with Resource Sharing

A. Introduction

In the last section of CGTM 93 we derived an algorithm to detect deadlock in the case where there existed 2 modes of resource control by a job: exclusive control, in which a resource element could be assigned to at most one job at any time, and shared control, in which a resource element could be assigned simultaneously to several jobs. The data base, B_2 , and detection algorithm, L_3, presented there were derived from the original data base, B_1, and algorithm, L_2, in a straightforward manner. Since we have developed in this memo data base B_3 and algorithm L_4 by analogy to B_1 and L_2, is is fairly obvious how to construct a new data base B_4 and a deadlock prevention algorithm L_5 that will permit resource sharing. Since the reasoning is essentially identical to that in section IX of CGTM 93, we will present the extensions here with a minimum of explanation.

F. Data Base B_4

Figure 4 presents a schematic description of the scheduler data base B_4 that will be required in order to prevent deadlock when resource sharing is allowed. When compared to figure 3, we see that the assignment matrix, A, the current demand matrix, D, and the maximum demand matrix, M, have each been replaced by 2 matrices AE and AS, DE and DS, and ME and MS, respectively. ME, DE, and AE represent the maximum demand, the current demand, and current assignment of resources in exclusive control mode, and MS, DS, and AS represent the maximum demand, current demand, and current assignment of resources in shared control mode. We also replace the derived matrix U by 2 matrices UE and US as follows:

$$\overline{UE} = \overline{ME}-\overline{AE}, \quad \overline{US} = \overline{MS}-\overline{AS}$$

As in CGTM 93, we define a new m-component vector RSHARE as the total number of resource elements under shared control by any job:

$$RSHARE[j] = \max_i AS[i,j] \text{ for } j=1, 2, \ldots, m$$

We then have

$$RFREE[j] = RMAX[j] - RSHARE[j] - \sum_i AE[kmj] \quad \text{for } j=1, 2, \ldots, m.$$

In order to keep track of the shared resource elements we need to define a new k by m matrix called QSHARE, where $k = \max_j RMAX[j]$ is the maximum number of resource elements in any one resource class R_j. The number in position $[i,j]$ of this matrix will be the number of jobs that currently have shared control of exactly i resource elements of type j. The algorithm L_4 also requires an auxiliary k by m matrix, QS, corresponding to QSHARE, and an m-component vector QMAX.

Finally we need to redefine vector UNUMB such that for i=1, 2, ..., n,

UNUMB[i] = the number of elements in the ith row of ($\overline{UE}$ + $\overline{US}$) that are > 0.

Thus the ith element in UNUMB represents the number of different resource classes that may be potentially demanded by job J_i, regardless of the number of elements or the mode of control required.

. Finishing Sequences

As will be proved later, a system with resource sharing will be safe if and only if there exists a finishing sequence F of all jobs in S such that, renumbering the jobs so that J_k is the kth job in F and defining Δ_i for i=2, 3, ..., n as

$$\overline{\Delta}_i = \overline{RFREE} + \overline{RSHARE} - \overline{W}[i,F] + \sum_{k-1}^{i+1} \overline{AE}[k]$$

then we require

(1) $\overline{UE}[1] \leq \overline{RFREE}$

$\overline{US}[1] \leq (\overline{W}[1,F]-\overline{AS}[1]) + (\overline{RFREE}-\overline{UE}[1])$

(2) $\overline{UE}[i] \leq \overline{\Delta}_i$

$\overline{US}[i] \leq (\overline{W}[i,F]-\overline{AS}[i]) + (\overline{\Delta}_i-\overline{UE}[i])$

for i=2, 2, ..., n.

The n by m matrix W, which is also a function of the sequence F, is defined exactly as in CGTM 93, so that W[i,j,F] represents the number of resource elements of type j that would remain under shared control if all jobs preceding J_i in F were to finish and no other job, including J_i, changed its status. As in algorithm L_3, it is not necessary to compute W for use in algorithm L_5, since the QSHARE matrix will accomplish the same purpose, as shown next.

E. Algorithm L_5

1) We assume that the ordering rule Z is defined so that each column j of Q is ordered according to increasing numbers of resource elements of type j that remain for potential demand in exclusive control mode by the job, and, in case of ties, by increasing numbers of resource elements of type j remaining for potential demand in shared control mode. That is:

UE[Q[i,j],j] ≤ UE[Q[i+1,j],j]

for i=1, 2, ..., QSIZE[j]-1

and in case of equality

US[Q[i,j],j] ≤ US[Q[i+1,j],j]

2) Initialization -- same as for algorithm L_4 except for the following:

for j = 1, 2, ..., m

$$\text{THRESH}[j] \leftarrow \text{RFREE}[j] + \sum_i \text{AE}[i,j] \text{ such that UNUMB}[i] = 0$$

and additionally

QS[i,j] := QSHARE[i,j] for i = 1, 2, ..., RMAX j = 1, 2, ...m

QMAX[j] := largest i such that QSHARE [i,j] > 0

3) The algorithm is then:

```
Repeat:    for j := 1 until m do
L1:            begin parallel
               I := Q[MARK[j], j] ;
L2:            while  (MARK[j] ≤ QSIZE[j])
                 AND (UE[I,j] ≤ THRESH[j])
                 AND (US[I,j] ≤ (QMAX[j] - AS[I,j] +
                                 THRESH[j] - UE[I,j]))
                 do begin
                    BUMP( I );
                    MARK[j] := MARK[j] + 1;
                    I := Q[MARK[j], j] ;
                    end ;
               end parallel ;
```

```
            if WC = 0 then <no deadlock>
        else if ACC̄ = 0̄ then <deadlock>
        else begin
            THRESH̄ := THRESH̄ + ACC̄ ;
            ACC̄ := 0̄;
            go to REPEAT ;
            end;
Procedure  BUMP (integer value I);
    begin
    DN[I] := DN[I] - 1;
    if DN[I] = 0 then
        begin if I = K then <stop, no deadlock>;

        ACC̄ := ACC̄ + AE[I]̄;
        WC := WC - 1;
B1 :    for J := 1 until m do
            if AS[I,J] > 0  then
                begin
                QS[AS[I,J], J] := QS[AS[I,J], J] - 1;
B2 :            while  (QMAX[J] > 0)  and
                    QS[QMAX[J], J] = 0 do
                    begin
                    ACC [J] := ACC[J] + 1 ;
                    QMAX[J] := QMAX[J] - 1;
                    end ;
                end ;
        end ;
    end ;
```

E. Scheduler Primitives

The REQUEST and RELEASE primitives presented in section VI must be modified somewhat to account for the 2 possible modes of resource control, exclusive and shared. Once again, however, it is not the creation of a current demand ($\overline{DE}_i$, $\overline{DS}_i$) by any job J_i that creates the possibility of introducing a deadlock into a previously safe system, but rather an assignment of resources to satisfy a current demand, whether in shared or exclusive control mode. Once again this is due to the fact that the detection algorithm, L_5, depends only on the values of $\overline{UE}$ and $\overline{US}$, and these change only when $\overline{AE}$ and $\overline{AS}$ change, i.e., when resources are actually allocated or deallocated. It will be shown later that resources can always be deallocated safely, so that only an allocation needs checking.

1) REQUEST ($\overline{NE}$,$\overline{NS}$) where $\overline{NE}$ is the m-component vector of new resource elements needed by J_i in exclusive control mode, and $\overline{NS}$ the vector of new elements needed in shared control mode. We required $\overline{NE} \leq \overline{UE}_i$, $\overline{NS} \leq \overline{US}_i$ in each component.

   a) If $\overline{NE} \leq \overline{RFREE}$ and

      $\overline{NS} \leq (\overline{RSHARE}-\overline{AS}_i) + (\overline{RFREE} - \overline{NE})$ (in each component)

      then proceed as follows:

      (1) for each NS[j] > 0, decrement QSHARE[AS[i,j], j] by one, and increment QSHARE[NS[j]+AS[i,j],j] by one.

      (2) $\overline{AE}_i \leftarrow \overline{AE}_i + \overline{NE}$ (so that $\overline{UE}_i \leftarrow \overline{UE}_i - \overline{NE}$)

      $\overline{AS}_i \leftarrow \overline{AS}_i + \overline{NS}$ (so that $\overline{US}_i \leftarrow \overline{US}_i - \overline{NS}$)

      (3) $\overline{RFREE} \leftarrow \overline{RFREE} - \overline{NE} - \max(\overline{AS}_i - \overline{RSHARE}, \overline{0})$

      $\overline{RSHARE} \leftarrow \max(\overline{RSHARE}, \overline{AS}_i)$

      where the "max" of 2 vectors is the vector of the max applied to the 2 corresponding components.

(4) For each j such that either $NE[J] > 0$ or $NS[j] > 0$, remove job J_i from its current position in column j of the Q matrix and re-insert it in that column according to rule Z.

(5) Execute algorithm L_5 to find a finishing sequence. If one exists, this assignment will create no deadlock, so we invoke the Allocate Exclusive primitive for each resource element requested by $\overline{NE}$, and the Allocate Shared primitive for each resource element requested by $\overline{NS}$. Then send a "go ahead" signal back to J_i.

(6) If algorithm L_5 fails to find a finishing sequence in step (5), then the assignment would create a deadlock situation and cannot be made at this time. We must therefore

(a) "undo" steps (1)-(4) by restoring QSHARE, AE, AS, RFREE, RSHARE, and Q to their original state

(b) put job J_i into the wait state by setting $\overline{DE}_i \leftarrow \overline{NE}$, $\overline{DS}_i \leftarrow \overline{NS}$, and adding J_i to the system waiting list WAITQ according to rule Y.

(b) If both tests in step a) are not satisfied in each component then the new resource demands of J_i cannot be met at this time, and J_i is put into the wait state by setting $\overline{DE}_i \leftarrow \overline{NE}$, $\overline{DS}_i \leftarrow \overline{NS}$, and adding J_i to the system waiting list WAITQ according to rule Y.

2) RELEASE ($\overline{NE}$, $\overline{NS}$) where $\overline{NE}$ is the m-component vector of resource elements currently under exclusive control of J_i but no longer needed, and $\overline{NS}$ the vector of elements currently under shared control by J_i but no longer needed. We require $\overline{NE} \leq \overline{AE}_i$, $\overline{NS} \leq \overline{AS}_i$ in each component.

a) Resources can always be deallocated safely at the instant the release is invoked, so we proceed as follows:

(1) set $\overline{RX} \leftarrow \overline{RSHARE}$

(2) for each NS[j] > 0, decrement QSHARE[AS[i,j],j] by one, and increment QSHARE[AS[i,j]-NS[j],j] by one. If QSHARE[AS[i,j],j] is reduced to 0, set $\overline{RX}[j] = \max_i \text{QSHARE}[i,j]$.

(3) $\overline{AE}_i \leftarrow \overline{AE}_i - \overline{NE}$ (so that $\overline{UE}_i \leftarrow \overline{UE}_i + \overline{NE}$)

$\overline{AS}_i \leftarrow \overline{AS}_i - \overline{NS}$ (so that $\overline{US}_i \leftarrow \overline{US}_i + \overline{NS}$)

(4) $\overline{RFREE} \leftarrow \overline{RFREE} + \overline{NE} + (\overline{RSHARE} - \overline{RX})$

$\overline{RSHARE} \leftarrow \overline{RX}$

(5) for each resource element in $\overline{NE}$ invoke the <u>Deallocate Exclusive</u> primitive, and for each resource element in $\overline{NS}$ invoke the <u>Deallocate Shared</u> primitive.

(6) for each component j of $(\overline{NE} + \overline{NS})$ that is > 0, remove job J_i from its current position in column j of matrix Q and re-insert it according to rule Z.

(7) send a "go ahead" signal back to J_i.

(b) If there is any job in wait state, as indicated by the system wait queue WAITQ, select the next job on that list, say J_k.

1) If $\overline{DE}_k \leq \overline{RFREE}$ and

$\overline{DS}_k \leq (\overline{RSHARE} - \overline{AS}_k) + (\overline{RFREE} - \overline{DE}_k)$

in each component, then proceed exactly as in steps (1)-(6) of item a) under the Request primitive, but replacing i with k, $\overline{NE}$ with $\overline{DE}_k$ and $\overline{NS}$ with $\overline{DS}_k$ throughout.

If algorithm L_5 succeeds in step 5 then we also remove J_k from the waiting list WAITQ, whereas if it fails we can simply leave J_k in place (thereby eliminating step (6)-(b)). If the assignment succeeds, repeat this procedure for the next job on WAITQ, if any.

2) if the above conditions are not true for J_k, we can either terminate the scheduler at once, or continue checking all jobs on WAITQ, depending on the rule Y being used to order WAITQ.

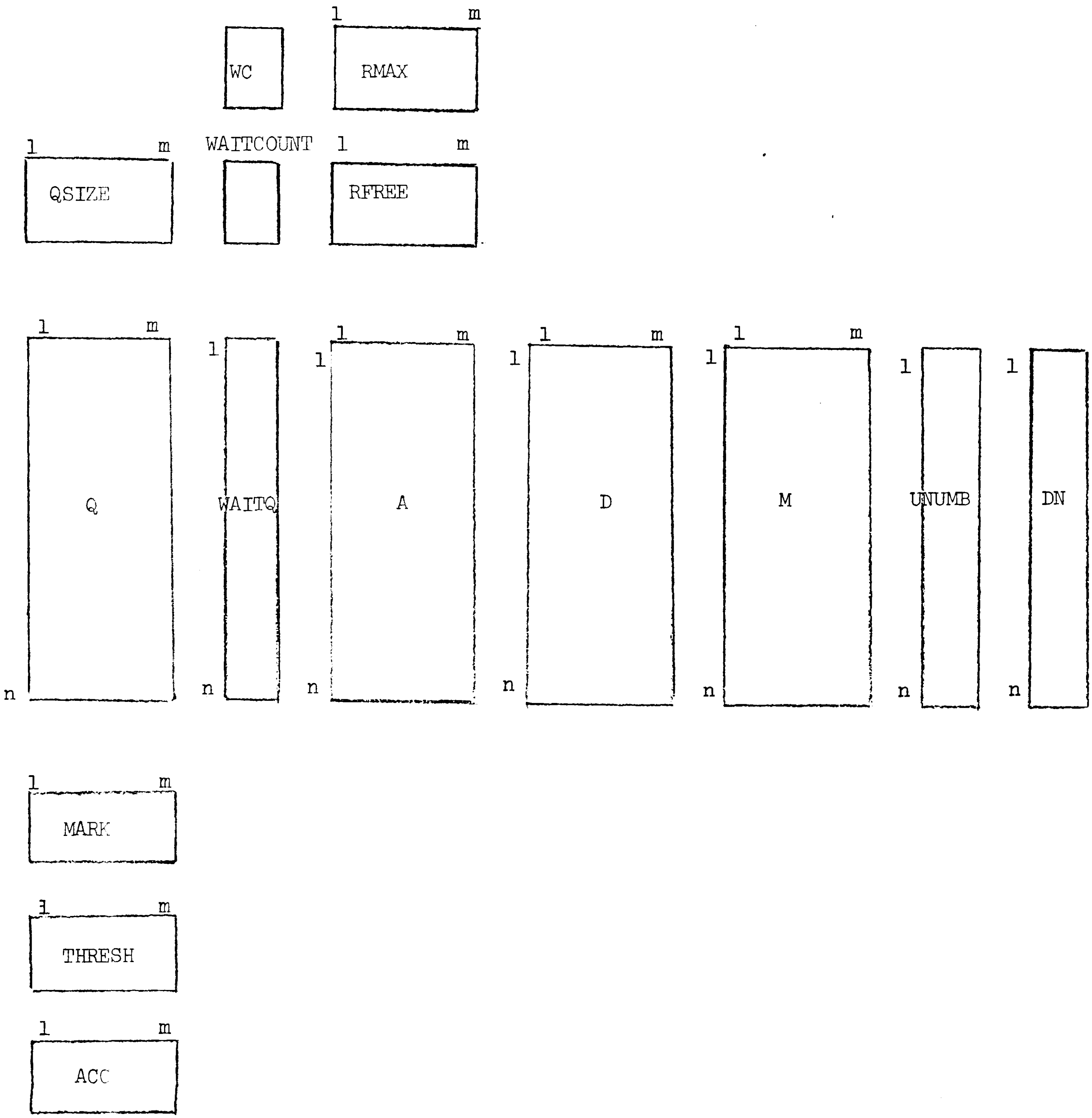

Figure 3

Scheduler Data Base B_3

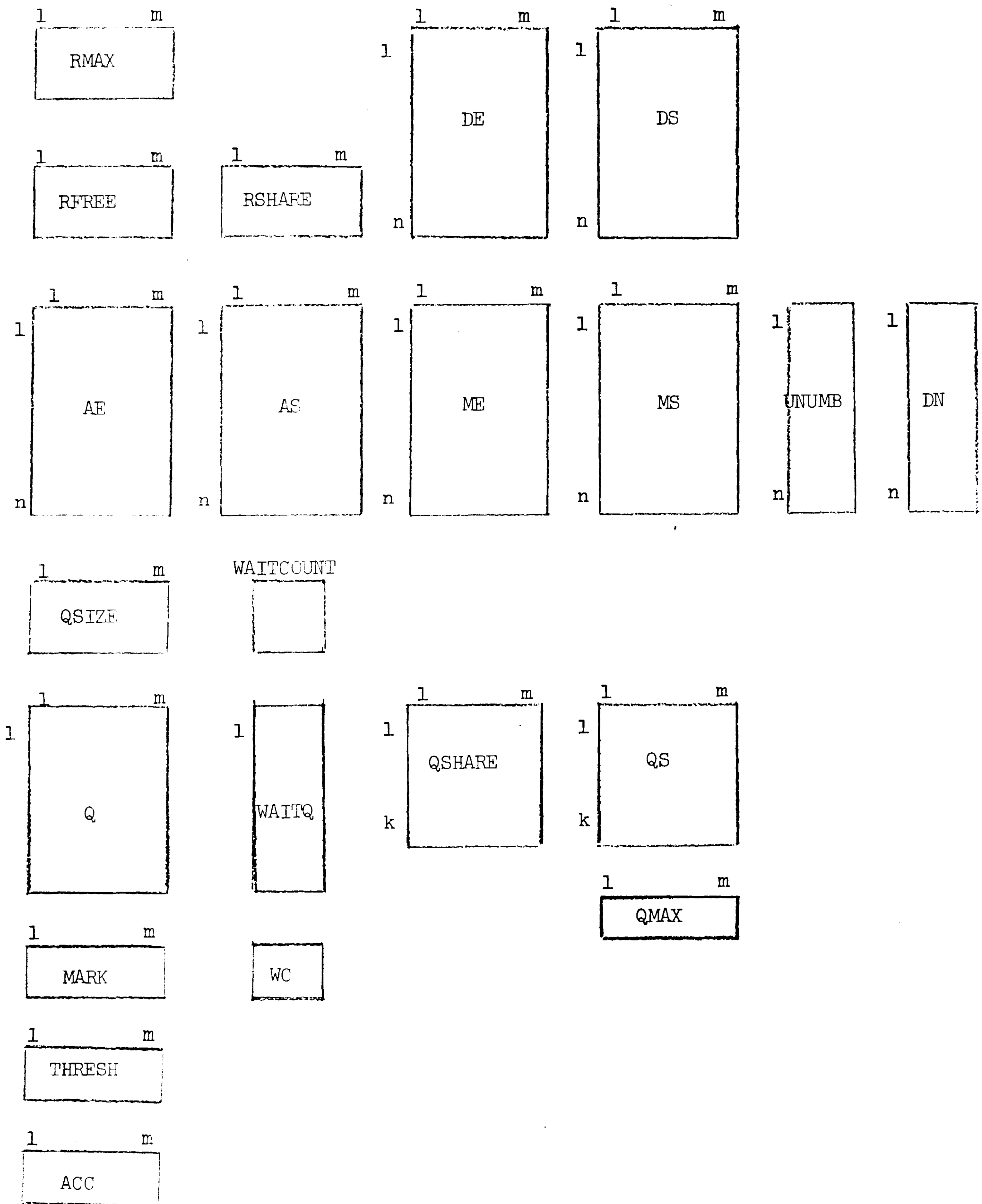

Figure 4

Scheduler Data Base B_4