CGTM # 93
R.D. Russell
June 1970

A Model for Deadlock-Free Resource Allocation --
Preliminary Version

Section I: The Basic Scheduler and Linear Algorithms for Deadlock Detection

I. Introduction

This memo describes a model of a scheduler and its associated scheduler data base for use in dynamic resource allocation systems. Based upon this model, the following topics are covered:

(1) A linear algorithm for deadlock detection

(2) A linear algorithm to detect "effective" deadlocks (Holt)

(3) The specification of scheduling primitives that incorporate these algorithms into their normal operation.

(4) A basic cost analysis of the expense due to dynamic deadlock detection

(5) An extension of the model and the linear algorithm to handle the previously uncovered complications due to resource sharing between jobs.

We assume some familiarity with the work of Habermann, Dijkstra, Shoshani, Murphy, and Holt. This model serves 3 general goals: (a) to bring together the results of the previous work into one model, demonstrating how "all the pieces fit"; (b) presenting linear algorithms to perform the same functions as the n^2 or higher-order algorithms found in the previous works; (c) to extend the research into previously unexplored areas such as the case of resource sharing between jobs. The theoretical basis for these algorithms, and a detailed description of the general system model, especially of the synchronization mechanisms, will appear in subsequent memos.

The next memo, which will be the direct sequel to this memo, continues development of the ideaspresented here to the case of deadlock prevention.

II. The System

A system $\hat{S}$ consists of a set of jobs, $J = \{J_1, J_2, \ldots J_n\}$, a set of resource classes, $R = \{R_1, R_2, \ldots R_m\}$, a Scheduler S, and a scheduler data base B. The resources are the atomic building blocks whose substructure is of no interest at the level being considered (the so-called operating system level). Each resource class, R_j, consists of $RMAX_j$ functionally equivalent resource elements of type j. Jobs are users of resources, and represent the entities in the system to which resources are attached. A job is completely independent of all other jobs, and in fact, is unaware of the existence of any other job in the system. The Scheduler is a special mechanism designed to determine the resource needs of jobs and to satisfy them in a coordinated manner. The Scheduler itself, and the scheduler-job, scheduler-resource interfaces are the primary concern of this memo. We can largely ignore the behavior of the jobs or of the individual resources insofar as the results they produce (the work they do) is concerned. We therefore define 2 states for each job in the system, called active and waiting, and 2 states for each resource element, called free and owned. Transitions between these states are caused by "primitive" interactions with the Scheduler that are explained next.

The need for resources by a job is decided upon by the job itself, and may be data-dependent. There are 2 primitive functions available to a job, the request primitive by which a job informs the Scheduler of resources it needs to control before it is able to proceed, and the release primitive by which a job informs the Scheduler of resources it now controls but no longer needs. Jobs that control all the resources that they need are said to be active; otherwise they are waiting.

A wait will start when one of these primitives is invoked by a job, and ends when the Scheduler has completely satisfied the primitive.

A job can decide by its own internal logic what its resource needs are, but only the scheduler can satisfy these needs. When informed of such a need by a job invoking a request, the Scheduler must decide which resource elements (if any) are available to satisfy the job. There are 2 primitive operations used by the scheduler to affect the control of a job over a resource. Allocate is used to establish the control of a resource element, and deallocate removes this control. A resource is in the free state only if it is not controlled by (assigned to, allocated to, attached to) any job. Otherwise it is in the owned state.

III. Behavioral Assumptions

Aside from the 2 states for resources, the 2 states for jobs, and the 2 primitive functions for communicating between a job and the scheduler, the behavior of the jobs or the resources is constrained only by the following list of behavioral assumptions. Later we will examine the implications of dropping one or more of these assumptions. Often, the character of the system will change dramatically from the Scheduler's point of view.

1) At any time a job cannot request or be assigned more resource elements than the system contains ($RMAX_j$ elements of type j).

2) At any time there is at most 1 owner of a resource element.

3) Only active jobs can request and release resources.

4) A job can request resources for allocation to itself only, and can release only those resources that it already controls.

5) An owned resource can only be deallocated in response to a release by the job that owns it.

6) An active job can request and/or release resources in any order and at any time that it sees fit.

7) A job may ask for new resources without relinquishing control of any that it might already possess.

8) The Scheduler has no advance information about the number of resources and/or the sequence of requests for any job.

Assumption 1) means that all requests must be for "hard" resources -- virtual mappings and multiplexing are not allowed. Assumption 2) implies that only those resource elements in the free (unowned) state can be allocated by the scheduler to satisfy a new request. That is, there is no resource sharing of 1 element by 2 or more jobs. Assumption 5) says that if there are insufficient free resources, the Scheduler cannot deallocate resources from another job in order to satisfy the new request. This means there is no resource preemption allowed. Assumptions 3), 4) and 5) together imply

that there is no hierarchy between jobs, in which one job exercises control over the resource requirements of a subordinate. Assumptions 6), 7) and 8) together imply that the jobs operate in a completely dynamic, "laissez-faire" environment, where the resource needs of a job depend only on the job itself and are decided upon within the job's internal processing without restrictions by the system environment.

IV. Scheduler Data Base B_1

Figure 1 is a diagram of the Scheduler Data Base B_1. Corresponding to each job J_i, the ith row of the A matrix, called the job assignment matrix, represents the number of resources currently controlled by J_i (the jth entry in that row is the number of resource elements of type j assigned to J_i). At all times we must have the relationship

$$\sum_{i=1}^{n} A[i,j] \leq RMAX[j] \quad j=1,2,..,m.$$

This says simply that the number of resources assigned to jobs cannot exceed the number of resources in the system.

The vector RFREE is the number of unassigned resources and the jth component corresponds to the number of free resource elements of type j. We therefore have the relation

$$RFREE[j] = RMAX[j] - \sum_{i=1}^{n} A[i,j]$$

The D matrix is called the current demand matrix. If all elements in the ith row of D are 0, then job J_i is <u>active</u>, since it has no unsatisfied demand for resources. If however $D[i,j] > 0$ is true, then job J_i is in the <u>waiting</u> state, due to the unsatisfied requirement of D[i,j] resource elements of type j. If there are several positive components in the ith row of D, then job J_i must be assigned all of the corresponding resource elements before it can become active again. Initially the only restriction is

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

for all i and j (Assumption 1 in section III).

Corresponding to the ith row of the D matrix is the element DNUMB[i] in the DNUMB vector. The value of DNUMB[i] is the number of positive elements in the ith row of D, regardless of their value. In other words

DNUMB[i] is the number of different resource classes that must contribute some number of resource elements to satisfy J_i. Obviously if DNUMB[i] = 0, then the ith row of D is all 0, and job J_i has no unsatisfied demand for resources. Hence DNUMB[i] = 0 implies J_i active; DNUMB[i] $>$ 0 implies J_i waiting. Since there are m different resource classes, we must have

$$DNUMB[i] \leq m \quad \text{for all } i.$$

The Q matrix is called the resource Queue matrix, each column corresponding to a resource class. Column j of the matrix represents an ordered list of pointers (or indices) of jobs with unsatisfied requests for resource elements of type j. (i.e. if index i is in column j of Q, then D[i,j] > 0). The rule Z by which jobs are ordered in the queue for each resource class is discussed later. Each column of Q is independent of the others, and may contain anywhere from 0 to n non-zero elements. Hence this matrix could in some implementations be more efficiently represented as a set of ordered lists, one list for each resource class. The choice between a vector and an ordered-list probably depends on the ordering rule Z, since linked lists are much easier to alter by adding or deleting elements in the middle. The matrix notation is chosen here simply for notational ease. We assume that if k jobs are waiting for resources of type j, then the first k elements of the jth column, Q[1,j], Q[2,j], ..., Q[k,j] will be the indices of these jobs (i.e. ℓ for job J_ℓ), and the remaining elements in the jth column Q[k+1, j], ..., Q[n,j] will be 0. The value k will also be the value of QSIZE[j], meaning the number of jobs waiting for resources of type j (i.e., the number of non-zero elements in column j of matrix Q).

The value WAITCOUNT is the number of jobs in the waiting state, that is, the number of DNUMB[i] $>$ 0. Obviously since there are only n jobs, we have

$$0 \leq \text{WAITCOUNT} \leq n$$

The indices of these waiting jobs are kept in the vector WAITQ, and are ordered according to the ordering rule Y.

Finally we will have 3 m-component vectors associated with the Q matrix, called MARK, THRESH, and ACC, one n-component vector DN associated with DNUM, and one single valued element WC associated with WAITCOUNT. Their meaning and use will be explained as part of the deadlock detection algorithm.

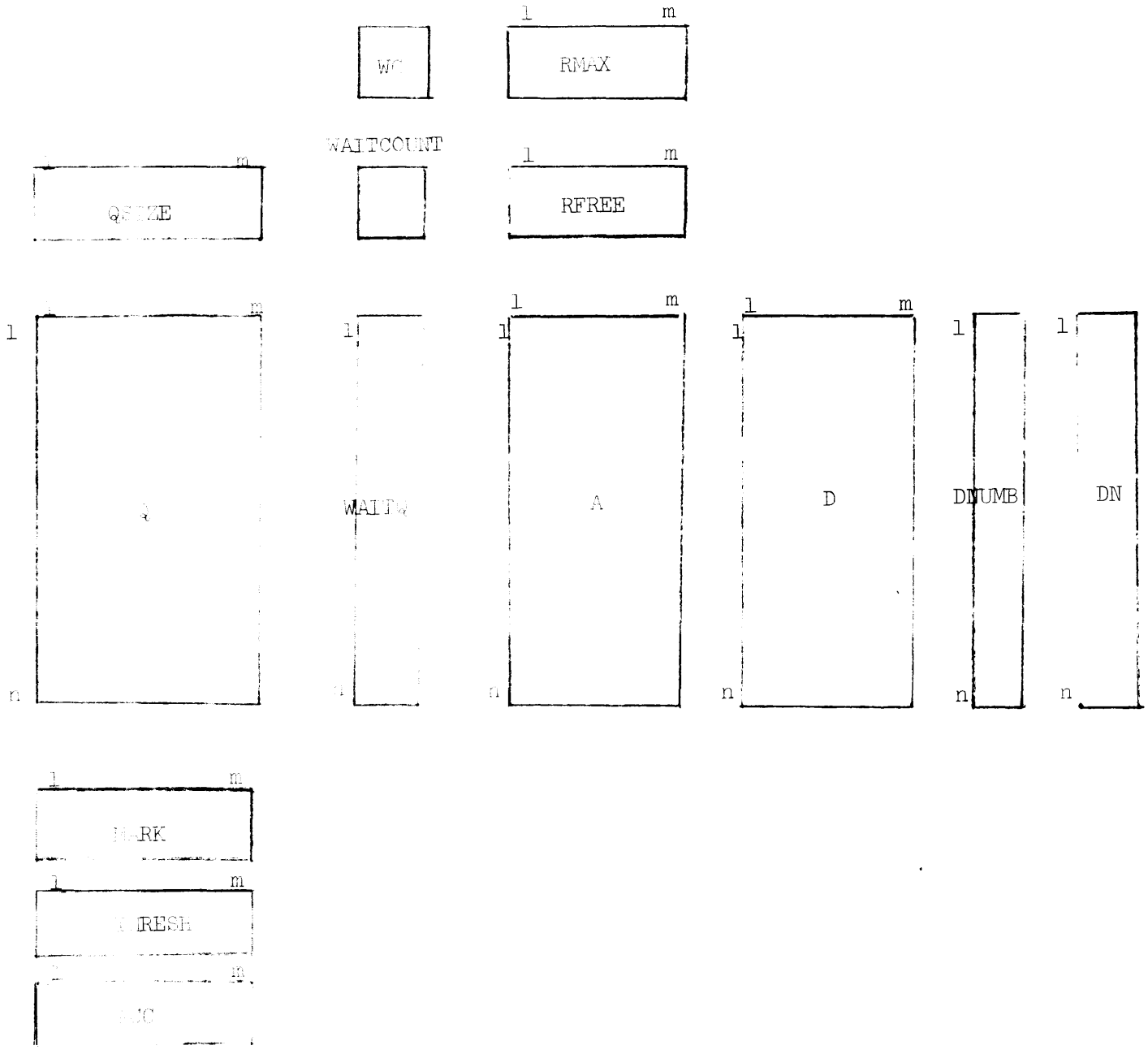

Figure 1

Scheduler Data Base B_1

V. Deadlock Detection Algorithm L_1

As has been shown by Shoshani, in the general case a System is safe (i.e. free from deadlock), if and 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)

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

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

where all operations are considered to be m-component vector operations.

The problem is to determine (by construction) the existence of such a sequence F given any arbitrary matrices D and A. Shoshani's algorithm, which is similar to that of Habermann's for detecting deadlock, is second-order linear in n. By taking advantage of the fact that the state of a system changes slowly over time (in fact, only the state of 1 job changes between applications of the algorithm to generate the sequence), we have arrived at a linear (in n) algorithm that requires only slight additional overhead (to maintain the Q matrix), and still generates a sequence F if one exists. Judicious choice of the data base B_1 permits increased speed at the cost of the extra storage needed by matrix Q, and a slightly more complicated determination algorithm.

We next state algorithm L_1, explain intuitively why it works, and point out certain special properties of the system that are being taken advantage of. The proof that in fact the algorithm is correct will be given in another memo. This algorithm, called the "sequence finder", is only one part of the total scheduling procedure which is explained immediately afterward in section VI.

1) We assume that the ordering rule % is defined so that each column j of Q is ordered according to increasing numbers of resource elements of type j requested by the job. That is,

$$D[Q[i,j], j] \leq D[Q[i+1,j], j]$$
$$\text{for } i = 1, 2, \ldots, QSIZE[j]-1$$

2) Initialization:

MARK[j] ← 1 j=1, 2, ..., m

ACC[j] ← 0 j=1, 2, ..., m

WC ← WAITCOUNT

DN[i] ← DNUMB[i] i=1, 2, ..., n

$$THRESH[j] \leftarrow RFREE[j] + \sum_{1} A[i,j] \text{ such that } DNUMB[i]=0 \quad j=1, 2, .., m$$

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
                   (D[I,j] ≤ THRESH[j])
                  do begin
L3:                   PUMP( 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
```

```
Procedure BUMP (integer value I);
    begin
    DN[I] := DN[I]-1;
    if DN[I] = 0 then
        begin
        ACC := ACC + A[I];
        WC := WC - 1;
        end
    end
```

4) Explanation

The algorithm allows a parallelism of m, since each resource can be testing its own column of Q independent of all others. The procedure BUMP is the only common Critical Section which can be called by at most 1 of the parallel blocks at any time. If the machine has a "decrement and branch on 0" instruction, then really only the inner block of BUMP need be made into a critical section. Algol unfortunately is incapable of expressing such an operation on DN[I].

The finishing sequence F for the system is given by the order in which DN[I] = 0 becomes true. That is, if I = 9 and DN[9] = 0 is the first DN to be found zero, then J_9 will be the first job in the sequence F. If DN[4] is the next DN to be found zero, then J_4 follows J_9 in F and so forth. Each pass through the REPEAT loop will find 1 or more reachable jobs in a safe system, and the order among these jobs in F is irrelevant. Hence, this algorithm can find many F's depending on the order in which the m iterations of the j loop are performed. Any one of these F's is sufficient, but if only 1 exists this algorithm is guaranteed to find it, as will be shown later. It should be obvious that any sequence F generated by this algorithm will satisfy the conditions on page 10 necessary for valid Finishing Sequences.

We should note that only waiting jobs (those with $\bar{D} > 0$ in some component) are processed in this algorithm. Without any advance information, active jobs (those with $\bar{D} = 0$ in all components) are assumed to be finishable with no further resource allocations. Strictly speaking, these jobs must be included in a complete finishing sequence F, and this algorithm assumes they are all "tacked on the beginning", by virtue of the initial value of THRESH. Thus we can consider F to be "initialized" to all active jobs, in any order, before executing L_1 to determine the rest of F.

f) Cost Considerations

Initially DN[i] is set to DNUMB[i], the number of different types of resources needed by J_i in order to become active again. The columns of Q are ordered by increasing unsatisfied demand D, and the algorithm takes advantage of the obvious fact that whenever $D[i,j] < THRESH[j]$ for some i, indicating that the demand by job J_i for resource j can be satisfied potentially, then for all K with $D[K,j] \leq D[i,j]$, job J_K can also be satisfied potentially. The resource therefore "approves" all these jobs, by decrementing DN[K], and moving the MARK value for that resource class up in the Queue (down column j in Q). Whenever all resources have "approved" a job (DN[K] = 0), that job is said to be "reachable" from the current system state. Hence it will eventually return all its currently assigned resources A[K] for use by other jobs. These resources are accumulated in $\overline{ACC}$ for use in the next step of the algorithm. Since each job J_i will contribute to $\overline{ACC}$ at most once (when DN_i goes to 0), there can be at most n changes to $\overline{ACC}$. Since at worst only 1 job will be reached on a complete execution of the "Repeat" loop, the algorithm will require at most n repetitions of this loop. Hence, the block labeled L1 will be entered at most m x n times (maximum of n iterations, m values of j per iteration). The test at L2 may be executed more however,

since each entry at L1 will always cause the test at L2 to fail once, but pass anywhere from 0 to n times. However, each "pass" causes the pointer $MARK_j$ to be incremented, so that the total "passes" over all entries to L1 for fixed j cannot exceed n, the maximum size of $QSIZE_j$, since as soon as $MARK_j$ exceeds $QSIZE_j$, the test at L2 must always fail. Hence over all iterations, for each j there can be at most n executions of the test at L2 that "pass", and thus at most n executions of the block at L3. If C_2 is the cost of a successful test at L2 plus the cost of executing the block at L3, each resource class contributes at most $n \times C_2$. Letting C_1 be the cost of a test at L_2 that fails (once per iteration) plus the cost of executing the other statements in the block L1 for fixed j, we obtain as the total cost per resource $n \times (C_1 + C_2)$, or for all m resource classes, the cost $m \times n \times (C_1 + C_2)$.

In a practical situation, the average cost of this algorithm will be considerably less, due to the following considerations. The "REPEAT" loop will never be executed a full n times, since only <u>waiting</u> (inactive) jobs must be checked by this algorithm. If k jobs are active, then the maximum number of repetitions is n-k, so that the cost per resource class for failures of test L2 is only $(n-k) \times C_1$. Obviously if k=0, every job is in wait state and the system is completely deadlocked. Therefore we can define the factor $\mu = (n-k)/n$, $0 \leq \mu < 1$, and state the reduced cost of the tests that fail as $\mu \times n \times C_1$ per resource class.

A second reduction is achieved by noting that jobs will not require elements from all m resource classes on each request, but on the average will need only ℓ different classes, $1 \leq \ell \leq m$. This means that the total number of indices in the Q matrix will average about $\ell \times (n-k)$, and since there are m columns in Q, the average length of a column in Q will be $q = \ell \times (n-k)/m$. We can therefore define two more factors as $\beta = \ell/m$ (the fraction of the total number of resource classes needed on any request), and $\gamma = q/n$ (the fraction of all jobs in each queue). This gives us the relationship $\gamma = \beta \times \mu$, which will be useful later.

Since the test at L_2 can succeed only on the average q times for each column of B, (instead of the full n as stated above) each resource class will incur the cost C_2 an average of q times. Hence the total cost to one resource class is now $(n-k)\times C_1 + q\times C_2 = \mu\times n\times C_1 + \gamma\times n\times C_2 = \mu\times n\times C_1 + \mu\times\beta\times n\times C_2 = \mu\times n\times(C_1 + \beta\times C_2)$. Since there are m resource classes, the total cost is $\mu\times m\times n\times(C_1 + \beta\times C_2)$.

We should also include a cost C_3 for the execution of the inner block of BUMP and the tests at L_4, including the vector additions to THRESH, each of which occurs at most once for each of the n-k waiting jobs, for a net cost of $(n-k)\times C_3 = \mu\times n\times C_3$. This gives us as the total cost of executing algorithm L_1 $\mu\times n\times(C_3 + m\times(C_1 + \beta\times C_2))$.

VI. Scheduler Operation

The above algorithm L_1 simply determines from a given state of the scheduler data base B_1 whether or not a finishing sequence F exists and hence whether or not the system is safe (deadlock-free). In this section we will show (a) the operations performed by the scheduler to change the data base, (b) how the ordering rule Z can be complied with efficiently, and (c) where in the normal course of scheduler functioning the algorithm L_1 is performed.

Operationally the objective is to maintain the System in a deadlock-free condition at all times. This is most easily done by use of a recursion relation. By assuming the system is safe at time t, we need only show what need be done to insure that it will remain safe at some time t' later ($t' > t$) after a change occurs in the system's state, as represented by the data base B. However B will change state only in response to a Request or Release primitive invoked by one of the jobs. The Scheduler exists for the sole purpose of responding to these primitives, and is idle (inactive) otherwise. Since only the Scheduler can access the data base B, only at these times will a change in B occur. We therefore simply give the procedures which the Scheduler follows in responding to each primitive (as invoked by arbitrary job J_i), and show how these will preserve the safety of the entire system.

A. Request ($\bar{N}$) where $\bar{N}$ is the m-component vector that specifies the new resource elements needed by J_i.

1) If $\bar{N} \leq \overline{RFREE}$ (in each component) then do the following:

   a) $\bar{A}_i \leftarrow \bar{A}_i + \bar{N}$;

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

   c) For each resource element requested by $\bar{N}$ invoke the Allocate primitive for that resource class ;

c) Send a "go ahead" signal back to J_i, indicating "request satisfied".

The resulting system will be deadlock-free (this will be proved later), and job J_i is not delayed.

2) If it is not true that $\bar{N} \leq \overline{RFREE}$ for all components, then the request cannot be immediately satisfied and job J_i will be forced to wait. We must do the following:

a) $\bar{I}_i \leftarrow \bar{N}$;

b) For each component j of $\bar{D}_i$ that is > 0, invoke the Enqueue primitive to put the index to job J_i into column j of the Q matrix according to the ordering rule Z. We also add J_i to the system waiting list WAITQ, according to ordering rule Y, and increment the value of WAITCOUNT by 1.

c) Execute the sequence finder algorithm L_1 to find a finishing sequence F. If it succeeds, then no deadlock has been created and the system is still safe. If it fails, then a deadlock exists. As will be shown later, this deadlock must involve J_i, so that recovery can be effected by either aborting J_i or pre-empting resources from it. Of course this is not the only way to recover, and probably is not the cheapest (see Shoshani).

. Release ($\bar{N}$) where $\bar{N}$ is the m-component vector that specifies the resource elements no longer needed by J_i

1) a) $\bar{A}_i \leftarrow \bar{A}_i - \bar{N}$

c) $\overline{RFREE} \leftarrow \overline{RFREE} + \bar{N}$

d) For each resource element represented by $\bar{N}$ invoke the Deallocate primitive for that resource class.

e) Send a "go ahead" signal back to J_i indicating "release complete".

Note that resources can always be released safely, and that the above operations are the complement of those done in the Request, step 1.

Job J_i is then permitted to proceed, since as far as it is concerned the scheduler has completely processed the Release primitive. However, the scheduler will continue asynchronously as follows:

2) If WAITCOUNT > 0, select the next job from the waiting list WAITQ, that is, if K = WAITQ[1] then select J_K and if $\bar{D}_K \leq \overline{RFREE}$ (in each component) then do the following:

   a) Remove J_K from the WAITQ, decrement WAITCOUNT by 1

   b) $\bar{A}_K \leftarrow \bar{A}_K + \bar{D}_K$

   c) $\overline{RFREE} \leftarrow \overline{RFREE} - \bar{D}_K$

   d) For each resource class j indicated in $\bar{D}_K$ invoke Dequeue to remove the job index K from the jth column of the Q matrix

   e) For each resource element indicated in $\bar{D}_K$ invoke the Allocate primitive for that resource class

   f) Send a "go ahead" signal back to J_K indicating request satisfied

   g) go back and repeat all of step 2

3) If the next job J_K found in step 2 does not satisfy $\bar{D}_K \leq \overline{RFREE}$, we can either terminate the scheduler operation at once, or continue checking all jobs on the WAITQ. The choice will depend to a large degree on the ordering rule Y for the WAITQ.

We note that step B2 is the same as step A1 with the added dequeuing to remove job J_K from the WAITQ and the resource class queues in Q. This dequeuing is the complement of the enqueuing performed in step A2 when the original request by J_K was unable to be satisfied at the time it was issued by J_K. Between these 2 instants of time, J_K has been in the wait state, essentially "hanging up" until it receives the "go ahead" signal from the Scheduler (step B2 f). This can be paraphrased in the colloquial jargon suggested by Dijkstra as follows. When a job J_K issues a request it "goes to sleep". The scheduler responds to the request and if it can be satisfied at the time

of issue (step A1), then the job is immediately reawakened by a "go-ahead" signal. Otherwise (step A2) the job is "put into bed" by entering its sleeping condition into the queues of B. At some later time the scheduler (step B2) realizes that the sleeping condition can be satisfied, so it gets the job "out of bed" (removes its wait condition from the queues of B) and reawakens the job by sending it the "go-ahead" signal.

We note further that a deadlock can occur only when a job is put to bed (step A2). A proof of this is given later. The enqueue-dequeue mechanism to put a job to bed and to get it out again is the only overhead necessary to maintain the ordering rule Z for the resource class queues Q, as is required in the deadlock detection algorithm L_1. When a job must be put to bed, the Enqueue and Dequeue primitives are both executed once for each resource class to which the job has submitted a demand. This is at most m classes, so there is at most 2m uses of these primitives. Since each queue is at most n jobs long, and each entry and deletion will require on the average $\frac{n}{2}$ compares and $\frac{n}{2}$ reshuffle operations in that column of Q, the overhead is $2 \times m \times n$ operations per request, so if C_4 is the cost of an enqueue (or dequeue), the total cost is $2 \times m \times n \times C_4$. This is obviously a worst possible case, and is felt to be extremely conservative for practical situations due to the following:

1) Not every request will incur this overhead. If the resources are available at the time of the request there is no overhead. We will call α the ratio of times a request incurs overhead to the total number of requests: $0 \leq \alpha \leq 1$.

2) Jobs will seldom require new resources in all m resource classes, and only those columns in Q that correspond to classes needed by the job will require any enqueue or dequeue operations to be performed. In practice then, this will be ℓ classes,

$0 < d \leq m$, giving $\beta = \frac{d}{m}$, $0 < \beta \leq 1$.

3 The queues will seldom if ever have all n jobs in them. The average length will depend to some extent on the particular resource class, but will have an average $q \leq n$, so letting $\gamma = \frac{q}{n}$ we have $0 < \gamma \leq 1$.

Thus an average figure for the overhead per request to maintain data base B_1 is $\varepsilon \times \alpha \times \beta \times \gamma \times m \times n \times C_1$ with $0 \leq \alpha$, β , $\gamma \leq 1$.

The more loaded (i.e. closer to saturation) the system, the closer α , β, and γ will be toward 1, and hence the higher the overhead.

Using the fact that $\gamma = \beta \times \mu$ (from section V-5), we can rewrite this cost as $\alpha \times \mu \times m \times n \times (2 \times \beta^2 \times C_1)$, which will be more useful in the next section.

VII. Cost Considerations and Algorithm L_2

The cost reduction factor α, defined in section VI, can also be applied to the cost of executing algorithm L_1, defined in section V-5, since this algorithm is executed only on requests that cannot be immediately satisfied (step A2, section VI), and this occurs $\alpha \times 100$ percent of the time. This reduces the cost of L_1 to $\alpha \times \mu \times n \times (C_3 + m \times (C_1 + \beta \times C_2))$ per request, so that together with the cost of maintaining the data base B_1 derived in section VI, the average cost of maintaining the system in a deadlock-free condition is

$\alpha \times \mu \times n \times (C_3 + m \times (C_1 + \beta \times C_2 + 2 \times \beta^2 \times C_4))$ per request, $0 \leq \alpha, \beta, \mu \leq 1$.

We can improve this even further by taking advantage of a special property of the F sequences. If the system at time t is safe and T_K makes a request at time t' that cannot be immediately satisfied, the system is safe at time t' if we can find a partial finishing sequence F' that ends in job J_K. In other words, we only have to guarantee that J_K, the job just put to sleep, will not sleep forever. We do not have to find a complete sequence containing all jobs. It can be proved that this partial sequence, coupled with the known fact that the system was safe before J_K made its request, is necessary and sufficient to guarantee that the new system is safe. This will reduce further the number of iterations of the checking loop in L_1 by an average amount δ, $0 < \delta \leq 1$. The only change necessary to give this new less costly algorithm L_2 is as follows:

1) Declare a global variable K whose value is the index of the job that just made the unsatisfiable request.

2) Add a test in procedure BUMP to check whether or not a job just determined reachable is J_K. If it is, the algorithm should immediately terminate with no deadlock. The revised BUMP is then:

```
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
```

This modified algorithm L_2 is used identically to L_1, but now the total cost is reduced to $\alpha \times \mu \times n \times (\delta \times C_3 + m \times (\delta \times C_1 + \delta \times \beta \times C_2 + 2 \times \beta^2 \times C_4))$, per request.

Cost Summary

C_1 ::= cost of one execution of the block L1, including one "fail" of test L2.

C_2 ::= cost of one execution of the block L3, including the cost of"passing" the test L2 and calling procedure BUMP.

C_3 ::= cost of one execution of the inner block of BUMP plus the tests at L4.

C_4 ::= cost of one comparison or one reshuffle operation used by Enqueue and Dequeue

α ::= average fraction of the total requests that cannot be satisfied at time of issue and hence the job must be "put to bed".

β ::= average fraction of the total number of resource classes needed on any one request

μ ::= average fraction of all jobs in wait state at any one time

δ ::= average ratio of the number of iterations required by L_2 to the number of iterations required by L_1.

ν ::= $\beta \times \mu$, the average fraction of all jobs in the queue for a particular resource class.

VIII. Effective Deadlock

In a recent monograph from Cornell, Holt describes a phenomenon he called "effective deadlock". This is caused by allowing the Scheduler to superimpose an ordering on the manner in which resources are allocated to jobs such that even though the system is safe because a valid finishing sequence, F, exists, the scheduler "never finds" that sequence, but instead attempts to allocate resources according to another sequence, T, which is not a valid finishing sequence. Although Holt offered no way to detect such a condition (he did give a scheme for avoiding it), our model permits an almost trivial modification to check for effective deadlock.

This is done by simply changing the ordering rule Z that decides how the jobs are maintained in the Q matrix. For example, Holt demonstrates how a FIFO allocation scheme, such that resources are given to jobs according to when the request was made, oldest first, can cause a deadlock. This deadlock is not detected, since Shoshani's algorithm proclaims the system safe. In our model, we simply define the Z rule as FIFO, and in case of ties, smallest demand first. The same Scheduler operation and the same detection algorithm (L_1 or L_2) can then be used without further modification to detect whether or not an "effective deadlock" exists. That is, the algorithm will attempt to generate a finishing sequence that preserves the FIFO discipline, and if it fails, then a deadlock exists.

Obviously any sort of imposed queuing condition, such as priority, LIFO, etc., can cause an effective deadlock, and these can all be detected trivially by simply redefining the ordering rule Z. Murphy also discussed deadlock detection when a FIFO ordering is imposed on the allocation strategy for the special case of 1 resource element of each type, but with the possibility of sharing that element among several jobs concurrently. In our model,

isolation of the ordering rule Z permits the same algorithm to detect deadlock under any constrained scheduler ordering, and also generalizes to the case of any number of resource elements of each type(without sharing). The next section discusses how sharing can also be incorporated in this model.

IX. Resource Sharing

A. Removing Assumption 2

Assumption 2 in section III stated that a resource could have at most 1 owner at any time. This eliminates the possibility of resource sharing, and this has been true of all previous works on deadlock detection and prevention, except for Murphy's. He considers the special case where there is exactly 1 element of each resource class, but this element can be controlled in one of 2 modes - exclusive control (single owner) and shared control (multiple simultaneous owners). Since it has been shown, by Denning in particular that resource sharing can be extremely advantageous to the operation of a computer system, it is surprising that this concept has largely been ignored in the research on deadlock detection. Fortunately, our model permits a simple, efficient method of incorporating resource sharing into both the Scheduler data base and the deadlock detection algorithms L_1 and L_2. Before explaining the necessary modification to our model presented so far, we note that Murphy's scheme depended upon a FIFO queuing discipline for each resource class. As should be obvious, and has been shown by us elsewhere, the particular ordering rule chosen is irrelevant to the problem of deadlock detection in the case where resources are shared. As might be suspected, our present model subsumes the queuing considerations under the choice of ordering rule Z, as has already been discussed previously.

B. Changes to the data base B_2

Because a job can now request and be assigned resources in one of 2 modes, the first obvious modification will be the replacement of the assignment matrix, A, by 2 matrices - AE for exclusive assignments and AS for shared assignments. Similarly the demand matrix D will be replaced by 2 matrices DE and DS. We will then define a new m-component vector RSHARE as

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

This represents the total number of resource elements under shared control by any job.

We also need to redefine RFREE and DNUMB as follows:

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

DNUMB[i] = number of positive elements in the ith row of $(\overline{DE} + \overline{DS})$ for i = 1, 2, ..., n.

Hence DNUMB is still the number of different resource classes that are required, regardless of the mode of control requested.

Finally, we will define the new k by m matrix QSHARE, where k = $\max_j RMAX[j]$. The entry in QSHARE [i,j] will be the number of jobs that have been assigned shared control of exactly i resource elements of type j. Corresponding to QSHARE is a second k by m matrix QS, and a vector QMAX, both of which are used in algorithm L_3. The new data base is pictured in figure 2.

2. Modified definition of finishing sequences

It can be shown by an argument similar to that of Shoshani and Habermann that a system in which resource sharing is allowed will be safe (i.e. free from deadlock) if and only if there exists a sequence F of all jobs in S such that (renumbering the jobs so that J_K is the Kth job in F)

1) $\overline{DE}[1] \leq \overline{RFREE}$

and $\overline{DS}[1] \leq (\overline{W}[1,F] - \overline{AS}[1]) + (\overline{RFREE} - \overline{DE}[1])$

2) if $\overline{\Delta_i} = \overline{RFREE} + \sum_{k=1}^{i-1} \overline{AE}[k] + \overline{RSHARE} - \overline{W}[i,F]$

then $DE[i] \leq \overline{\Delta_i}$

and $\overline{DS}[i] \leq (\overline{W}[i,F] - \overline{AS}[i]) + (\overline{\Delta_i} - \overline{DE}[i])$

for $i = 2, 3, \ldots, n$

This definition includes a new matrix W which is a function of both the job i and the sequence F. This value is not in our data base because it is not necessary for the algorithm to be described shortly. It is however

a useful way to define the necessary and sufficient conditions for F to be a finishing sequence. It is defined conceptually as follows. Suppose each resource element that is under shared control by one or more jobs has associated with it a "control list"of indices to all the jobs that possess such shared control for that element. We can then define a vector $\bar{W}[i,F]$ such that the jth element of $\bar{W}$ is the number of resource elements of type j that are under shared control and have on their control list either the index i or the index of at least 1 job following J_i in F. Intuitively $\bar{W}[i,F]$ has as its value the number of resource elements that would be under shared control if all jobs preceding J_i in F were to finish and no other job changed its status. Since W obviously depends on the particular sequence F and must be recomputed for each F, it is easy to see that algorithms that generate F's would get extremely expensive if they also had to compute $\bar{W}$ to check the validity of the resulting F. Fortunately there exists an algorithm to generate F's that uses the matrix QSHARE (which is independent of F) in lieu of W to check for deadlocks. This will be presented next.

D. Deadlock detection with Resource sharing, Algorithm L_3

1) We assume that the ordering rule Z is defined so that each column j of Q is ordered according to increasing DE_j, and in case of ties, by increasing DS_j-- that is:

$$DE[Q[i,j], j] \leq DE[Q[i+1,j], j]$$

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

and in case of equality

$$DS[Q[i,j], j] \leq DS[Q[i+1,j], j]$$

2) Initialization -- same as for algorithm L_1 except for the following:

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

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

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 (DE[I,j] ≤ THRESH[j])
                  AND (DS[I,j] ≤ (QMAX[j] - AS[I,j] +
                                  THRESH[j] - DE[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 ACC = 0 then <deadlock>
              else begin
                   THRESH := THRESH + ACC ;
                   ACC := 0;
                   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>
```

```
                ACC := ACC + 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
```

4) Explanation

Except for procedure BUMP, only the test on line L2 is different from the basic L_1 algorithm given in section V. This expanded test simply reflects the extra work which must necessarily be incurred when there are 2 types of resource assignment possible (exclusive and shared control) instead of only 1. Likewise the new procedure BUMP is identical to the previous BUMP but with the block labeled B1 inserted. (We have also incorporated the early cutoff criterion of section VII.) This block has also been added to take care of those resource elements that are being shared among several jobs. The block labeled B2 tests for the significant case where the job just "approved" in the call to BUMP is also the last one to control a shared resource element in the sequence

F. In that case, the element becomes free for assignment in either mode of control to all subsequent jobs in F.

Although an additional test and another m iteration block has been added, the algorithm is still linear in n and the previous cost analysis remains a good approximation if the cost per operation figure of C_1 is adjusted accordingly.

E. Changes to the Scheduler primitives

The new algorithm L_3 requires some further changes to the 2 primitives REQUEST and RELEASE. This is to be expected since there are now 2 A matrices and 2 D matrices as well as a new QSHARE matrix, all of which will require updating at one time or another by the Scheduler.

1) REQUEST ($\overline{NE}$, $\overline{NS}$) -- In keeping with the 2 possible modes of control, NE is the number of resources for which exclusive control is requested, and NS is the number for which shared control is asked. The scheduler responds as follows:

a) If $\overline{NE} \leq \overline{RFREE}$ and $\overline{NS} \leq (\overline{RSHARE} - \overline{AS}_i) + (\overline{RFREE} - \overline{NE})$ then the assignment can be made immediately

1) for each component j such that $NS[j] > 0$, we must decrement QSHARE[AS[i,j], j] by one and increment QSHARE[NS[j] + [AS[i,j], j] by one. This updates the count in QSHARE of how many jobs possess what number of shared resources of each type.

2) $\overline{AE}_i \leftarrow \overline{AE}_i + \overline{NE}$

$\overline{AS}_i \leftarrow \overline{AS}_i + \overline{NS}$

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 a vector is the vector of the max applied to each component.

4) For each resource element requested by $\overline{NE}$ invoke the Allocate exclusive primitive, and for those requested by $\overline{NS}$ invoke the Allocate shared primitive.

5) Send a "go ahead" signal to J_i, indicating the request is satisfied.

b) If both tests in step a are not satisfied, then the resource demands of J_i cannot be met at the time the request is issued, and hence it must be enqueued for a later time. We must therefore do the following:

(1) $\overline{DE}_i \leftarrow \overline{NE}$; $\overline{DS}_i \leftarrow \overline{NS}$;

(2) For each component j of $(\overline{DE}_i + \overline{DS}_i)$ that is > 0, invoke the Enqueue primitive to put the index i to job J_i into column j of the Q matrix according to ordering rule Z stated above. We also add J_i to the list of waiting jobs, WAITQ, according to ordering rule Y, and increment the value of WAITCOUNT by 1.

(3) Perform algorithm L_3 to find a finishing sequence. If it succeeds, the system is safe. Otherwise this request has created a deadlock, and appropriate recovery (or termination) action must be taken.

2) RELEASE ($\overline{NE}$, $\overline{NS}$) -- operates identically with the previous version of RELEASE, but now accounts for the 2 modes of resource control.

a) The deallocation can always be performed safely at the time the RELEASE is invoked.

(1) For each component j of NS > 0, we must decrement QSHARE[AS[i,j], j] by one and increment QSHARE[AS[i,j] - NS[j], j] by one. This complements step 1-a-1 above.

(2) $\overline{AE}_i \leftarrow \overline{AE}_i - \overline{NE}$

$\overline{AS}_i \leftarrow \overline{AS}_i - \overline{NS}$

$\overline{RFREE} \leftarrow \overline{RFREE} + \overline{NE}$

(3) for $j = 1, 2, \ldots, m$, $\overline{RSHARE}_j \leftarrow$ largest i such that QSHARE $[i,j] > 0$. Note that this i will be no larger than the previous value of $RSHARE_j$, so that the test should start at $i = RSHARE_j$ and work backwards toward 0. Each time the test fails (QSHARE[i,j] = 0), we need also to add 1 to the jth component of $\overline{RFREE}$, since J_i was the last job to use the resource element in shared mode.

(4) For each resource element in $\overline{NE}$ and $\overline{NS}$ invoke deallocate exclusive and deallocate shared respectively.

(5) Send the "go ahead" signal back to J_i.

b) If WAITCOUNT > 0, select the next job on the waiting list WAITQ, say J_K, and if both $\overline{DE}_K \leq \overline{RFREE}$ and

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

then it is safe to restart J_K as follows.

(1) Remove J_K from WAITQ and decrement WAITCOUNT by 1

(2) Remove J_K from all the resource queues by invoking Dequeue on column j of matrix Q for each component of $(\overline{DE}_K + \overline{DS}_K)_j > 0$.

(3) Proceed as in part 1-2, steps (1)-(5) for a request that is immediately satisfiable, everywhere substituting $\overline{DS}_K$ for $\overline{NS}$ and $\overline{DE}_K$ for $\overline{NE}$.

(4) Go back and repeat all of this step (b).

c) As in the case of RELEASE for non-shared resources, the option of searching the entire WAITQ for any job that "fits" (i.e. passes the above test) will depend on the choice of ordering rule Y for the WAITQ.

These new primitives are very similar to the original ones discussed in part VI, the only changes being those obviously needed to test for the 2 types of resource control modes, and the extra expense of maintaining the QSHARE matrix.

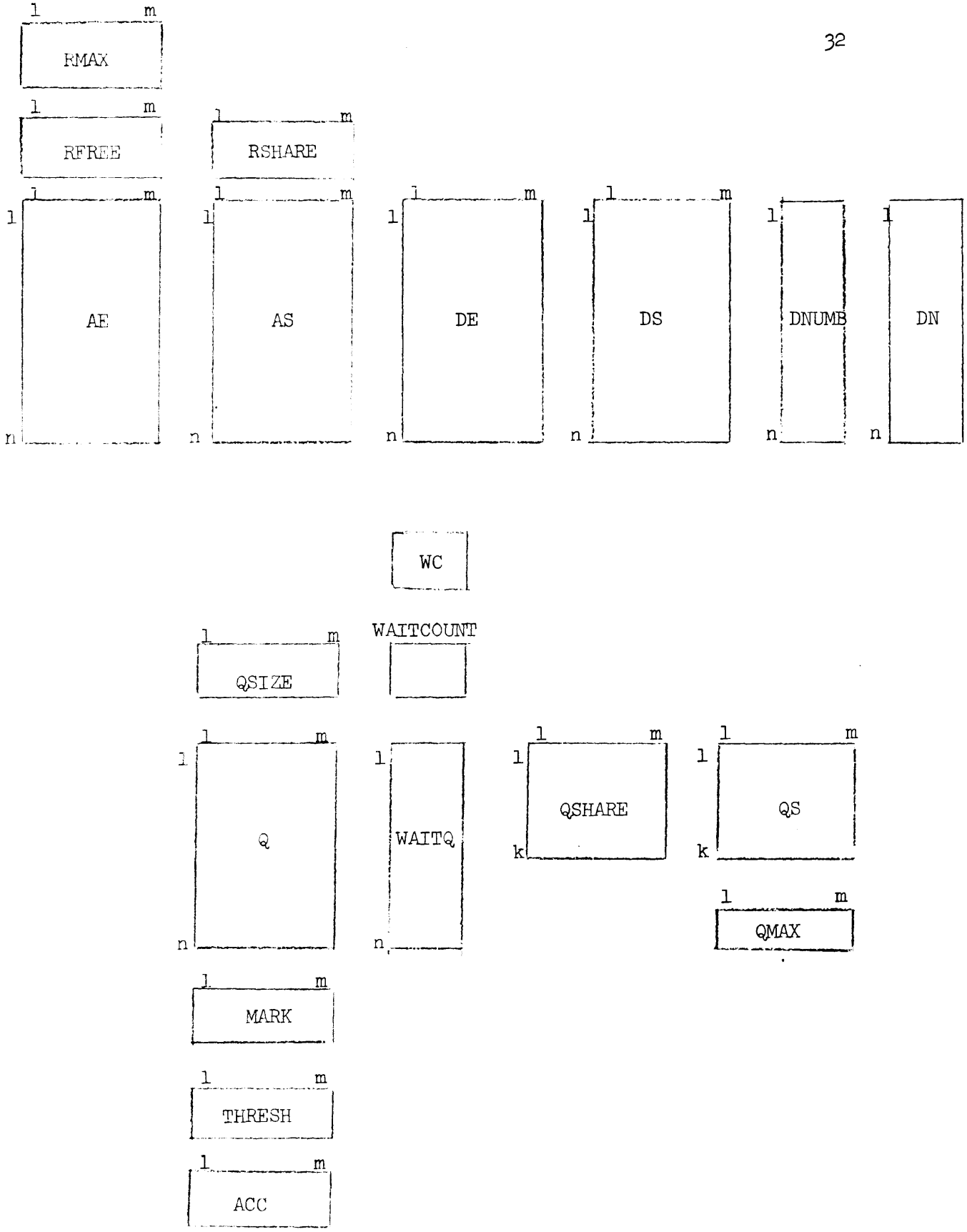

Figure 2

Scheduler Data Base B_2

Bibliography on Deadlock in Resource Allocation

1) Collier, W.W. "Systems Deadlocks" IBM Technical Report TR-001756 (1968)

2) Dijkstra, E.W. "Cooperating Sequential Processes" EWD-123, Department of Mathematics, Technological University Eindhoven, The Netherlands (Sept. 1965) 84 pages

3) Dijkstra, E.W. "The Structure of the "THE"-multiprogramming System" C.A.C.M. Vol. 11 No. 5 (May 1968) pp. 341-346

4) Habermann, A.W. "On the Harmonious Cooperation of Abstract Machines" Ph.D. Thesis, Mathematics Department, Technological University Eindhoven, The Netherlands (Oct. 1967)

5) Habermann, A.W. "Prevention of System Deadlocks" C.A.C.M Vol. 12 No. 7 (July 1969) pp. 373-377

6) Havender, J.W. "Avoiding Deadlock in Multi-Tasking Systems" IBM System Journal Vol. 2 No. 7 (1968) pp. 74-84

7) Hebalkar, Prakash G. "Coordinated Sharing of Resources in Asynchronous Systems" Computation Structures Group, Memo No. 45 Project MAC, MIT, Cambridge, Mass. (Jan. 1970) 29 pages

8) Hebalkar, Prakash G. "Some Structural Properties of Demand Graphs" Computation Structures Group Memo No. 46, Project MAC, MIT. Cambridge, Mass. (April 1970) 20 pages

9) Hebalkar, Prakash G. "Deadlock Avoidance in Multi-resource Systems" Computation Structures Group Memo No. 48, Project MAC, MIT Cambridge, Mass. (April 1970) 29 pages

10) Holt, Richard C. "Comments on Prevention of System Deadlocks" Technical Report No. 70-50 (Jan. 1970), Department of Computer Science, Cornell University, Ithaca, New York 12 pages

11) Murphy, James E. "Resource Allocation with Interlock Detection in a Multi-task System" Proceedings, Fall Joint Computer Conference (Dec. 1968) Vol. 33, part 2, pp. 1169-1176

12) Shoshani, Arie "Detection Prevention and Recovery from Deadlocks in Multiprocess Multiple Resource Systems" Thesis, Dept. of Electrical Engineering, Princeton University (Oct. 1969) 115 pages

14) Shoshani, Aire, and Bernstein, A.J. "Synchronization in a Parallel-Accessed Data Base" C.A.C.M. Vol. 12, No. 11 (Nov. 1969) pp.604-607

15) Shoshani, Arie, and Coffman, E.G. Jr. "Detection, Prevention, and Recovery from Deadlock in Multiprocess, Multiple Resource Systems" Technical Report No. 80, Electrical Engineering Dept., Computer Sciences Laboratory, Princeton University (Oct. 1967) 42 pages

16) Shoshani, Arie, and Coffman, E.G. Jr. "Sequencing Tasks in Multiprocess, Multiple Resource Systems to Avoid Deadlocks" Technical Report No. 78, Electrical Engineering Department, Computer Sciences Laboratory, Princeton University (June 1969) 50 pages