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

This memo describes a model of a scheduler and its associated scheduler database 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 minimal cost analysis of the expense due to dynamic deadlock detection
5. An extension of the model and the linear algorithm to handle the previously unhandled complications due to resource sharing between jobs.

We assume some familiarity with the work of Habermann, Dijkstra, Shochani, Murphy, and Holt. This model seeks three general goals: (a) to bring together the results of the previous work into one model, demonstrating how "all the pieces fit"; (b) present 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 previous unexplored areas such as the cost of resource sharing between jobs.

The theoretical basis for these algorithms, and a detailed description of the general model and, especially of the synchronization mechanisms, will appear in a subsequent memo.

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

A system $S$ 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 Scheduler; 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 ($\text{RMAX}_j$ 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 $E_1$

Figure 1 is a diagram of the Scheduler Data Base $E_1$. Corresponding to each job $J_j$, the $i$th row of the $A$ matrix, called the job assignment matrix, represents the number of resources currently controlled by $J_j$ (the $j$th entry in that row is the number of resource elements of type $j$ assigned to $J_j$).

At all times we must have the relationship

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

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

The vector $E$FREE is the number of unassigned resources and the $j$th component corresponds to the number of free resource elements of type $j$.

We therefore have the relation

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

The $D$ matrix is called the current demand matrix. If all elements in the $i$th row of $D$ are 0, then job $J_i$ is active, since it has no unsatisfied demand for resources. If however $D[i,j] > 0$ is true, then job $J_i$ is in the waiting state, due to the unsatisfied requirement of $D[i,j]$ resource elements of type $j$. If there are several positive components in the $i$th 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

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

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

Corresponding to the $i$th row of the $D$ matrix is the element $D\text{NUMB}[i]$ in the $DNUMB$ vector. The value of $D\text{NUMB}[i]$ is the number of positive elements in the $i$th row of $D$, regardless of their value. In other words
\( D\text{NUMB}[i] \) is the number of different resource classes that must contribute some number of resource elements to satisfy \( J_1 \). Obviously if \( D\text{NUMB}[i] = 0 \), then the \( i \)th row of \( D \) is all 0, and job \( J_1 \) has no unsatisfied demand for resources. Hence \( D\text{NUMB}[i] = 0 \) implies \( J_1 \) active; \( D\text{NUMB}[i] > 0 \) implies \( J_1 \) waiting. Since there are \( m \) different resource classes, we must have
\[
D\text{NUMB}[i] \leq m \quad \text{for all } i.
\]

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 \( j \)th column, \( Q[1,j], Q[2,j], \ldots, Q[k,j] \) will be the indices of these jobs (i.e. \( i \) for job \( J_i \)), and the remaining elements in the \( j \)th column \( Q[k+1,j], \ldots, Q[n,j] \) will be 0. The value \( k \) will also be the value of \( Q\text{SIZE}[j] \), 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 \( \text{WAITCOUNT} \) is the number of jobs in the waiting state, that is, the number of \( D\text{NUMB}[i] > 0 \). 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 $\text{WAITQ}$, and are ordered according to the ordering rule $Y$.

Finally we will have $n$-component vectors associated with the $Q$ matrix, called $\text{PARK}$, $\text{THRESH}$, and $\text{ACC}$, one $n$-component vector $\text{DN}$ associated with $\text{DHU}$, and one single valued element $\text{WC}$ associated with $\text{WAITCOUNT}$. Their meaning and use will be explained as part of the deadlock detection algorithm.
Figure 1
Scheduler Data Base B₁
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 \textbf{finishing sequence}) of all jobs in \( S \) such that (renumbering the jobs so that \( j_k \) is the \( k \)th job in \( F \))

\[
\begin{align*}
D[1] &< R_{FREE} \\
D[i] &< R_{FREE} + \sum_{k=2}^{i-1} A[k], \quad i=2, 3, \ldots, n
\end{align*}
\]

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 in \( m \). 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 \( m \)) algorithm that requires only slight additional overhead (to maintain the \( Q \) matrix), and still generates a sequence \( F \) if one exists. Additional revision 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 detection algorithm.

We next state \( L_1 \) from \( L_1 \), 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 manner. 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 $\mathcal{E}$ 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,

\[ D(Q[i, j], j) \leq D(Q[i+1, j], j) \]

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

2) Initialization:

$\text{MARK}[j] \leftarrow 1 \quad j=1, 2, \ldots, m$

$\text{ACC}[j] \leftarrow 0 \quad j=1, 2, \ldots, m$

$\text{WC} \leftarrow \text{WAITCOUNT}$

$\text{DH}[i] \leftarrow \text{DNUMB}[i], i=1, 2, \ldots, n$

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

3) The algorithm:

\begin{verbatim}
RMAX = 1 for i := 1 until m do
  begin parallel
    I := random \text{MARK}[i], \text{ACC}[i] \;
    while (\text{MARK}[i] \leq \text{QSIZE}[j] \text{AND} \text{D}[I,j] \leq \text{THRESH}[j])
      do begin
        PUMP(I);
        \text{MARK}[i] := \text{MARK}[i] + 1;
        \text{ACC} := \text{ACC} + 1;
      end
  end parallel
  if \text{WC} = 0 then <no deadlock> \;
  else if \text{ACC} = 0 then <deadlock> \;
  else begin
    \text{THRESH} := \text{THRESH} + \text{ACC};
    \text{ACC} := 0;
    go to REPEAT;
  end
\end{verbatim}
Procedure BUMP (integer value I);
begin
DN[I] := DN[I] - 1;
if DN[I] = 0 then
begin
 Acc := Acc + ALT;
 WC := WC - 1;
end
end

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[1].

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[1] is the next DN to be found zero, then J_1 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 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 of page 10 necessary for valid Finishing Sequences.
We should note that only waiting jobs (those with $D > 0$ in some
components) are processed in this algorithm. Without any advance information,
active jobs (those with $D = 0$ 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 "backed 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 $F$.

1) 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 $D[K,j] < D[i,j]$, job $J_K$ can also be satisfied potentially.
The resource therefore "approves" all these jobs, by decrementing $DN[K]$,
and leaving 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 $ACC$ for use in the next step
of the algorithm. Since each job $J_i$ will contribute to $ACC$ at most once
when $DN[i]$ goes to 0), there can be at most $n$ changes to $ACC$. 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 $m \times n$ times (maximum of $n$ iterations,
$n$ values of $j$ per iteration). The test at $L2$ may be executed more however,
since each entry at L2 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 \( nC_2 \). Letting \( C_1 \) be the cost of a test at L2 that fails (one per iteration) plus the cost of executing the other statements in the block L3 for fixed \( j \), we obtain as the total cost per resource \( n(C_1 + C_2) \), or for all \( n \) resource classes, the cost \( n \times n(C_1 + C_2) \).

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 waiting (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 \((n-k)C_1 \). 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 \( \mu n \times C_1 \) 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 \( d \) different classes, \( 1 \leq d \leq m \). This means that the total number of indices in the Q matrix will average about \( d \times (n-k) \), and since there are \( m \) columns in Q, the average length of a column will be \( q = d \times (n-k)/m \). We can therefore define two more factors as \( \beta = d/m \) (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 $L_1$ 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 
\[(n-k)\times C_1 + n \times C_2 = \mu \times n \times C_1 + \gamma \times n \times C_2 = \mu \times n \times (C_1 + \gamma \times C_2).\]

Since there are $n$ resource classes, the total cost is $\mu \times n \times (C_1 + \gamma \times C_2)$. 

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 $L_1$ 
\[\mu \times n \times (C_3 + \mu \times (C_1 + \gamma \times C_2)).\]
VI. Scheduler Operation

The above algorithm \( L_1 \) simply determines from a given state of the scheduler data base \( E \), 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 \( E \) 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 \( E \). However \( E \) 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 \( E \), only at these times will a change in \( E \) 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.

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

   1) If \( \bar{N} \leq \text{FREE} \) (in each component) then do the following:
      a) \( \bar{A}_i \leftarrow \bar{A}_i - \bar{N} ; \)
      b) \( \text{FREE} \leftarrow \text{FREE} - \bar{N} ; \)
      c) for each resource element requested by \( \bar{N} \) invoke the Allocate primitive for that resource class ;
Send a "go ahead" signal back to $J_1$, indicating "request satisfied".

The resulting system will be deadlock-free (this will be proved later), and Job $J_1$ is not delayed.

2) If it is not true that $T_i \leq RFREE$ for all components, then the request cannot be immediately satisfied and Job $J_1$ will be forced to wait. We must do the following:

a) $T_i \leftarrow T_i$

b) For each component $j$ of $D_i$ that is $> 0$, invoke the Enqueue primitive to put the index to Job $J_1$ into column $j$ of the $A$ matrix according to the ordering rule $Z$. We also add $J_1$ to the system waiting list WAITQ, according to ordering rule $Y$, and increment the value of WAITCOUNT by 1.

c) Invoke 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 $J_1$, so that recovery can be effected by either aborting $J_1$ 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 $J_1$

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

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

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

Send a "go ahead" signal back to $J_1$ indicating "release complete".

We have shown that resources can always be released safely, and that the above operations are the complement of those done in the Request, step 1.
Job $J_k$ 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 $\text{WAITCOUNT} > 0$, select the next job from the waiting list WAITQ, that is, if $K = \text{WAITQ}[1]$ then select $J_K$ and if $D_K \leq \text{RFREE}$ (in each component) then do the following:

a) Remove $J_K$ from the WAITQ, decrement WAITCOUNT by 1
b) $\overline{A}_K \leftarrow \overline{A}_K + \overline{D}_K$
c) $\text{RFREE} \leftarrow \text{RFREE} - \overline{D}_K$
d) For each resource class $J$ indicated in $\overline{D}_K$, invoke Dequeue to remove the job index $K$ from the $j$th column of the $Q$ matrix
e) For each resource element indicated in $\overline{D}_K$, 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 $\overline{D}_K \leq \text{RFREE}$, 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 2c 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 2c). 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 B1) 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 L1. 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 are 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 overall overhead is \( 2 \times m \times n \times C_h \) operations per request, so if \( C_h \) is the cost of an enqueue (or dequeue), the total cost is \( 2 \times m \times n \times C_h \). 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 \( \alpha \) 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 \( 2 \) classes,
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 \( 0 < \gamma < 1 \).

Thus an average figure for the overhead per request to maintain data base \( B_1 \) is

\[ 2^\alpha \times \beta \times \gamma \times m \times n \times O_1 \]

with \( \alpha \leq \alpha', \beta \leq \beta', \gamma \leq 1 \).

The more loaded (i.e. closer to saturation) the system, the closer \( \alpha', \beta', \) and \( \gamma \) 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

\[ 2^\alpha \times \mu \times (2q^2 \times O_1) \]

which will be more useful in the next section.
VII. Cost Considerations and Algorithm L₂

The cost reduction factor \( \lambda \), defined in section VI, can also be applied to the cost of executing algorithm \( L₁ \), 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 \( \lambda \times 100 \) percent of the time. This reduces the cost of \( L₁ \) to \( \lambda \times m \times (C₃ + m \times (C₁ + \beta \times C₂)) \) per request, so that together with the cost of maintaining the data base \( B₁ \) derived in section VI, the average cost of maintaining the system in a deadlock-free condition is

\[
\lambda \times m \times (C₃ + m \times (C₁ + \beta \times C₂ + 2\gamma \times C₄)) \text{ per request, } 0 \leq \gamma, \beta, \mu \leq 1.
\]

We can improve this even further by taking advantage of a special property of the \( P \) sequences. If the system at time \( t \) is safe and \( Tₖ \) 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 \( Jₖ \).

In other words, we only have to guarantee that \( Jₖ \), 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 \( Jₖ \) 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 \( L₁ \) by an average amount \( \delta \), \( 0 < \delta \leq 1 \). The only change necessary to give this new less costly algorithm \( L₂ \) 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 \( Jₖ \). 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 L₂ is used identically to L₁, but now the total cost is reduced to \( m \cdot n \cdot (A \cdot C₂ + m \cdot (A \cdot C₁ + A \cdot C₂ + 2 \cdot A \cdot C₄)) \), per request.

Cost Summary

\( C₁ \) := cost of one execution of the block L₁, including one "fail" of test L₂.
\( C₂ \) := cost of one execution of the block L₃, including the cost of "passing" the test L₂ and calling procedure BUMP.
\( C₃ \) := cost of one execution of the inner block of BUMP plus the tests at I4.
\( C₄ \) := cost of one comparison or one reshuffle operation used by Enqueue and Dequeue.
\( \kappa \) := average fraction of the total requests that cannot be satisfied at time of issue and hence the job must be "put to bed".
\( \beta \) := average fraction of the total number of resource classes needed on any one request.
\( \alpha \) := average fraction of all jobs in wait state at any one time.
\( \delta \) := average ratio of the number of iterations required by L₂ to the number of iterations required by L₁.
\( \gamma \) := \( \beta \cdot \alpha \), 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 exists, the scheduler "never finds" that sequence, but instead attempts to allocate resources according to another sequence 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

$$RSHARE[j] = \max_{i} AS[i,j]$$

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:

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

\[ DNUMB[i] = \text{number of positive elements in the } i\text{th row of } (DE + DS) \]

for \( i = 1, 2, \ldots, n. \)

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 \( k = \max \ RMAX[i,j] \). 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 L2. 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 \( K \)th job in \( F \))

\[ DE[1] \leq RFREE \]

\[ AS[1] \leq (W[1,F] - AS[1]) + (RFREE - DE[1]) \]

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

then \( DE[i] \leq \Delta_i \)

\[ AS[i] \leq (W[i,F] - AS[i]) + (\Delta_i - DE[i]) \]

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 \( \overrightarrow{W}_{i, F} \) such that the \( j \)th element of \( \overrightarrow{W} \) 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 \( J_j \) in \( F \). Intuitively \( \overrightarrow{W}_{i, F} \) has as its value the number of resource elements that would be under shared control if all jobs preceding \( J_j \) 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 \( \overrightarrow{W} \) 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.

2. Deadlock detection with Resource sharing, Algorithm \( L_3 \)

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

\[
DE[C[i,j], j] \leq DE[C[i+1,j], j]
\]

for \( i = 1, 2, \ldots, \text{QSIZE}[j] - 1 \)

and in case of equality

\[
DS[C[i,j], j] \leq DS[C[i+1,j], j]
\]

2) Initialization — same as for algorithm \( L_1 \) except for the following:

\[
\text{for } j = 1, 2, \ldots, m
\]

\[
\text{THRESH}[j] \leftarrow \text{RFREE}[j] + \sum_{i} \text{AB}[i,j] \text{ such that } \text{DNUMB}[i] = 0
\]
and additionally
\[ QS(i,j) := QSHARE(i,j) \quad \text{for } i = 1, 2, \ldots, \text{RMAX} \quad j = 1, 2, \ldots, \text{m} \]
\[ QMAX[j] := \text{largest } i \text{ such that } QSHARE[i,j] > 0 \]

3) The algorithm is then:

\[ \text{Repeat: for } j := 1 \text{ until } m \text{ do} \]

\[ \text{L1: } \begin{align*}
& \text{begin parallel} \\
& I := Q[\text{MARK}[j], j];
\end{align*} \]

\[ \text{L2: } \begin{align*}
& \text{while } (\text{MARK}[j] \leq \text{QSIZE}[j]) \\
& \quad \text{AND } (\text{DE}[I,j] \leq \text{THRESH}[j]) \\
& \quad \text{AND } (\text{DS}[I,j] \leq (\text{QMAX}[j] - \text{AS}[I,j] + \text{THRESH}[j] - \text{DE}[I,j]))
\end{align*} \]

\[ \text{do begin} \\
& \text{BUMP}( I ); \\
& \text{MARK}[j] := \text{MARK}[j] + 1; \\
& I := Q[\text{MARK}[j], j]; \\
\end{align*} \]

\[ \text{end parallel} \]

\[ \text{if } WC = 0 \text{ then } < \text{no deadlock}> \]

\[ \text{else if } \text{ACC} = 0 \text{ then } < \text{deadlock}> \]

\[ \text{else begin} \\
& \text{THRESH} := \text{THRESH} + \text{ACC}; \\
& \text{ACC} := 0; \\
& \text{go to REPEAT}; \\
\end{align*} \]

\[ \text{end} \]

Procedure BUMP (integer value I);

\[ \text{begin} \\
& \text{DN}[I] := \text{DN}[I] - 1; \\
& \text{if } \text{DN}[I] = 0 \text{ then} \\
& \text{begin if } I = K \text{ then } < \text{stop, no deadlock}> \]
\[ \text{ACC} := \text{ACC} + \text{AE}[I]; \]
\[ \text{WC} := \text{WC} - 1; \]

31: \hspace{1em} \text{for } i := 1 \text{ until } m \text{ do}

\hspace{2em} \text{if } \text{AS}[I,J] > 0 \text{ then}

\hspace{3em} \text{begin}

\hspace{4em} \text{QS}[\text{AS}[I,J], J] := \text{QS}[\text{AS}[I,J], J] - 1;

\hspace{2em} \text{end}

32: \hspace{1em} \text{while } (\text{QMAX}[J] > 0) \text{ and } 

\hspace{2em} (\text{QMAX}[J] > 0) \text{ do}

\hspace{3em} \text{begin}

\hspace{4em} \text{ACC}[J] := \text{ACC}[J] + 1;

\hspace{4em} \text{QMAX}[J] := \text{QMAX}[J] - 1;

\hspace{4em} \text{end}

\hspace{2em} \text{end}

\hspace{1em} \text{end}

\hspace{1em} \text{end}

4) \text{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.
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.

1. Changes to the Scheduler primitives

The new algorithm $L_2$ 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 ($NE, NS$) -- 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 $NE \leq RFREE$ and $NS \leq (RSHARE - AS_i) + (RFREE - NE)$ 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) $AE_i \leftarrow AE_i + NE$

   $AS_i \leftarrow AS_i + NS$

   3) $RFREE \leftarrow RFREE - NE - \max(AS_i, RSHARE, 0)$

   $RSHARE \leftarrow \max(RSHARE, AS_i)$ where the max of a vector is the vector of the max applied to each component.
1) For each resource element requested by NE invoke
the Allocate exclusive primitive, and for those requested
by NS invoke the Allocate shared primitive.

2) 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. For each component j of (DE_i' + 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.

2) 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 (NE, 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[i,j], j] by one. This complements
   step 1-a-1 above.
(2) \[ AE_i \leftarrow AE_i - NE \]
\[ AS_i \leftarrow AS_i - NS \]
\[ RFREE \leftarrow RFREE + NE \]

(3) for \( j = 1, 2, \ldots, m \), \( RSHARE_j \leftarrow \) 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 \( j \)th component of \( RFREE \), since \( J_i \) was the
last job to use the resource element in shared mode.

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

(5) Send the "go ahead" signal back to \( J_1 \).

b) If \( WAITCOUNT > 0 \), select the next job on the waiting list
\( WAITQ \), say \( J_K \), and if both \( RSHARE - AS_K \)
\( \leq \) (\( RFREE - DE_K \))
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 \( (RFREE + DE_K)_j \) \( > 0 \).

(3) Proceed as in part 1-2, steps (1)-(5) for a request that
is immediately satisfiable, everywhere substituting
\( DE_K \) for \( NS \) and \( DE_K \) for \( NE \).

(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 B₂
Bibliography on Deadlock in Resource Allocation


2) Dijkstra, E.W.  "Cooperating Sequential Processes"  EWD-123, Department of Mathematics, Technological University Eindhoven, The Netherlands (Sept. 1965) 24 pages


