Section 3. Static Deadlock Prevention

I. Introduction

There are essentially 2 ways to prevent deadlock in a resource allocation system -- dynamically and statically. Dynamic prevention is achieved by formulating an algorithm that can be applied to the state of a system after every resource allocation to determine whether or not a deadlock was created by that allocation. If a potential deadlock is detected, the allocation is delayed until some later safer time. Such an algorithm and the manner of integrating that algorithm into the operation of the system was given in [Holl63].

The second method of deadlock prevention, which we will call static prevention, is achieved by suitably restricting the legal job behaviour so that it becomes impossible to get into a deadlock situation. That is, there are certain conditions pertaining to job behaviour within a system that are necessary in a deadlock to occur. Therefore by creating a system (i.e., restricting job behaviour within a system) that violates one or more of these necessary conditions, deadlock can never arise. One such set of conditions has been given by Shoshani in his thesis. This is obviously a fixed "static" prevention of deadlock, since the safety of the system is independent of its current state -- all possible threat states are known a priori, and hence no dynamic prevention algorithm need to run whenever an allocation is made. Since most previous dynamic prevention algorithms were extremely expensive to run, on the order of \( n^2 \), most current systems that have worried about preventing deadlock (some systems, like the B5500, do not attempt to prevent it -- they simply assume that it will occur so rarely that it will not be bothersome) have chosen a static prevention approach, as for instance OS/360 MVT.

This memo describes a scheme for representing resource allocation systems in which a static prevention method is applicable. This one representation appears
to be able to describe most static prevention methods as special cases. Therefore
it is necessary to only show that the most general system represented in this
manner is deadlock-free, and all subsets of this will then also have the same
property.

II. A Graph Representation

We can consider a resource allocation system to be represented as a
directed acyclic graph, which we will call a "sequencing graph" for reasons
which will become clear later. Each node in the graph consists of (1) a set
of one or more unique resource elements; (2) an input queue into which lead
all the edges of the graph directed into that node; (3) an output queue, from which
come all the edges of the graph directed out of that node; (4) two "primitive"
(i.e., noninterruptable and mutually exclusive) routines called "assign" and
"unassign". Each resource element in the system is located in exactly 1
node of the graph, and remains there at all times. Each job in the system is
represented by a single element which always resides in exactly 1 queue in the
graph. There is also assumed to be 1 unique initial node with no edges directed
into it, and at least 1 terminal node with no edges directed out of it (Figure 1).

The behaviour of a job is restricted (as far as its resource allocation
requests are concerned) such that it "travels on the edges" from node to node,
and at each node it can request for allocation to itself any subset of the
resource elements assigned to that node. The job, as represented by its unique
queue element, is always in 1 of 2 states - active or waiting. In the active
state, the job has all the resources it needs and is able to make progress at
its own rate without any assistance from the scheduler. In this state the job
element will reside in the output queue of the node where the last resource
allocation was made. In the waiting state, the job requires the allocation of
one or more resource elements before it can continue. The transition of a job
from active to waiting state results from scheduler action initiated by the
FIGURE 1. A Typical Sequencing Graph, Containing 6 Nodes and 4 Resource Classes: A (A₁-A₆), B (B₁-B₆), C (C₁-C₄) and D (D₁-D₃)
job itself when it issues a REQUEST primitive operation. The parameter to this
primitive is a vector representing the number and type of resource elements
being requested (if sharing is allowed, this vector can also specify the type
of allocation desired - exclusive or shared. This will be discussed later).
This demand vector is part of the queue element for that job and cannot be changed
except by the action of the scheduler invoking an allocate primitive. In this
state the job element will reside in the input queue of the node to which the
request was made. This must be one of the nodes connected by an edge in the
graph to the node in whose output queue the job resided when it issued the
request.

The transition of a job from waiting to active state can be caused only
by the action of the scheduler when it actually allocates resources to a job
as requested in an earlier REQUEST by that job. This transition is represented
in the graph by removing the job element from the input queue for that node
and placing it into the output queue for that node.

Each resource element exists in one of two states - free (unassigned) when
it is not allocated to any job, and busy (assigned) when it is allocated to at
least one job. If resource sharing is allowed, a single resource element can
be assigned simultaneously to more than one job, whereas if jobs are required
to have exclusive access to their resources, a busy resource element will be
assigned to, at most, one job. We will assume the latter case in the rest of
this discussion.

If we consider the initial node of our graph to be the "top", and the
terminal nodes to be the "bottom", then a job queue element will always reside
in a queue that is below the nodes containing the resource elements already
assigned to that job. This is due to two properties of the graph as we have
constructed it, (1) a job element can only travel from node to node along the
directed edges, and since the graph is acyclic, these all point downward;
(2) a job element "passes through" a node (i.e., goes from input queue to out-
put queue for that node) only after all the resource elements in that node
needed by a job are assigned to that job. Therefore, we can view the progress of a job to be represented in the graph, by the job queue element starting at the initial (top) node and "falling" through the graph structure along the edges, stopping at each node to "pick up" a (possibly empty) subset of the resource elements that are located in that node. The rate of the fall is dependent on the number of resource elements needed at each node, the intervals between the need for new resources by the job, and, of course, the availability of the resource elements due to the competition with other jobs.

The job element does not actually "pick up" the resource element at a node, but rather obtains the ability to access that resource element, either exclusively or in conjunction with other independent jobs if resource sharing is allowed. The allocate primitive, issued by the scheduler, establishes this access capability and changes the state of the resource element from free to busy. Access permission, once granted to the node to a job, can only be relinquished by that job through the use of a special RELEASE primitive. Although resource elements can be requested and allocated only according to the number and order imposed by the interconnection between nodes and the availability of the elements, they can be released by the job at any time and in any order provided the job is active (i.e., residing in an output queue of some node). The deallocated resource element is placed back into free state and becomes available for immediate assignment to any job waiting in the input queue for the node in which the resource is located. The job immediately relinquishes all access capabilities for that resource element.

The route by which a job element travels through the graph from the initial node to its current position in some queue will be called a path-history. Since a job can only request resources from nodes below its current position in the acyclic graph and connected to it by edges, it may not be able to request a large number of resource elements that are above or unconnected to its current node. In decisions about resource needs are to be made dynamically by each job,
some mechanism must be provided to enable the job element to go back and pick-up previously un seizeable but now un-accessible resource elements, or to choose a new path that will lead to the now inaccessible nodes. We will call this process "trailing," since the job will ascend in the graph, retracing its original path, but in the direction opposite to the directed edges. This process is initiated by the job through the use of the RETRACE primitive, which must be issued once for each node through which the path is being retraced. This primitive removes the job element from its current location in an output queue, deallocates (i.e., returns to free state) all resource elements assigned to the job which are located in that node, and then places the job element into the output queue for the node which preceded this node on the job's path history. Once in this new position, the job can select any of the edges out of that queue, including the one back to the node from which it just retraced, and can make a new request for any subset of the resource elements in the next nodes, including those elements just released in the previous retrace.

III. Primitive Definitions

Before showing that systems whose operation can be described by this graph model are free from deadlock, we need to define the synchronization properties of the components in the graph. By judicious definition of the three primitives (Request, Release, Retrace) available to the job, we can treat each node as an independent asynchronous "allocator" which neither knows nor cares about other allocators in the system. Since each job is already assumed to proceed independently and asynchronously to other jobs in the system, this will permit maximum simultaneous activity of all system components.

To do this, we must specify some "lowest level" primitives as the basis building blocks. We assume that a queue is a mechanism with two mutually exclusive (indivisible) operations defined on it - Enqueue to add a designated element to the queue according to some ordering rule (which may vary from queue to queue), and Dequeue to remove a designated job element from the queue,
regardless of its position within the queue. At any instant at most one of
these operations can be performed on any single queue, but, of course, simultaneous operations are possible on different queues.

We also postulate two job statuses primitives; Block to prevent further activity by the designated job by changing its state from active to waiting, and Unblock to return the state of the job to active. Two primitives are also needed to change the states of resource elements - allocate to establish the necessary accessibility of a designated resource element to a designated job, thereby changing the state of the resource element from free to busy, and deallocate to remove a job's access capability to a resource element which, thereby, returns to the free state.

Finally, we need two "historical" primitives in order to keep a correct path history for each job - Enter adds a designated node onto the path history of a job as the most recent node, and Remove removes the most recent node in the path history of the designated job.

Using these four lowest level primitive-pairs, we can define the two routines associated with each node as "intermediate level" primitive pairs. For any node, only one of these can be performed at any instant, but simultaneous operation of different nodes is, of course, permissible. We use the notation that "next (P,q)" refers to the next job element on queue q of node P, if any; "size (J)" refers to the current outstanding demand (if any) of job element J; "free (P)" refers to the vector of currently free resources in node P; "last (J)" refers to the last (most recent) node on the path-history of job J; and "quantity (P,J)" is the set of resource elements (if any) in node P currently allocated to job J.

The definitions are then:

\[
\text{Assign (P)}
\begin{align*}
\text{begin} \\
\text{Start: } J := \text{next (P, input)}; \\
\text{if } (J \neq \text{null}) \text{ and } (\text{size (J)} \leq \text{free (P)}) \text{ then} \\
\text{begin}
\end{align*}
\]
Dequeue P, input, J;
for each r, in size (J) do
    in
    Allocate (r);
    end;
    free (r) := free (P) - size (J);
    enter (P, J);
    enqueue P, output, J;
    unlock J;
    go to wrt;
    end;
end;

Request (P, N)
begin
for each r in N do
    begin
    deallocate (r);
    end;
    free (P) := free (P) + N;
end;

We have assumed that parameter N is the set of resource elements in node P that are no longer needed by some job J.

Finally, we can define five "highest-level" primitives. Only one of these can be performed at any instant by one job, but they can occur simultaneously for different jobs.

Request (P, N, r)
begin
    clock (J);
    enqueue (lock (J), output);
    dequeue (P, input, J);
    assign (P);
end

Release (P, N, r)
begin
    unassign (P, N);
    assign (P);
end
begin
  P:=last (j);
  unassign (P, quantity (P, j));
  enqueue (P, output, j);
  remove (j);
  enqueue (last (j), output, j);
  assign (P);
end

We also need another high-level primitive pair which give us the capability of entering new jobs into the system and removing old ones from it. The initiate primitive creates a new job element and enters it into the input queue for the initial node. The resources in this node will be the job initiators necessary for a job to possess before it can actively compete for resources. The terminate primitive releases all the resources assigned to the job and then destroys the job element. It is issued by a job after all its activities are completed.
IV. Properties of the Model

Inherent in the discussion so far and in the part that is to follow are five very basic assumptions about the systems being represented in the model.

Assumption 1: (Job independence) Each job is a unique sequential "process" or "computation" which is independent of all other jobs in the system. A job is totally unaware of the existence of other jobs, and has no way to detect their presence.

Assumption 2: (Non-virtual Resources) A job can only request and be granted a maximum of all the resources that are assigned to each node that is "visited" by the job. That is, there are no "virtual" resources accounted for in the model, and only those resources assigned to nodes along a single path in the graph can be allocated to a single job simultaneously.

Assumption 3: (Resource Equivalence) Resource elements are assigned to specific nodes in the graph, and within a node these elements are grouped into resource equivalence classes such that each element of a class performs identically as far as a job using it is concerned. Therefore, a job can only make requests for "any element" of a specified class (type). The allocator functions will determine which elements of the class are actually allocated to satisfy the request.

Assumption 4: (Finite Termination) If provided with all the requested resources, subject to the constraints imposed by the existence of paths in the graph, a job must terminate in a finite time. The termination process, of course, involves freeing all resources still allocated to the job being terminated.

Assumption 5: A resource element can be allocated to, at most, one job at a time. That is, we do not allow simultaneous sharing of a single resource element by two or more jobs.

This last assumption is not strictly necessary, in its entirety, since systems in which resource sharing is allowed can still encounter deadlock problems, as discussed in DVTM's 93 and 94. However, it simplifies the following discussion to make this assumption now, and then relax it later. Note, however, that if all resources were completely shareable at all times, no deadlock...
could arise since obviously all requests for resources could be immediately satisfied. Therefore, the essential part of this assumption consists of the following - it must be possible for a job to request and be granted exclusive allocation of at least a non-empty subset of the resource elements in the system. The effects of relaxing assumption 5 to this form will be discussed later.

With these assumptions in mind, we can delimit some of the properties inherent in the graph model and the set of primitives defined on it.

**Consequence 1:** The only source of delay to the progress of a job is lack of free resources to satisfy a request. This follows from the independence of a job from all others, and the fact that the only "communication" with the job's environment is by the use of the job primitives. Of these, only the Request primitive has the potential to block the progress of a job, and that is only due to insufficient free resources to satisfy the request.

**Consequence 2:** Since they are independent, jobs can be run to completion in any arbitrary order, such that one job must terminate before any resources are allocated to the next job. This would probably be extremely wasteful of resources since many would be idle much of the time, but due to assumption 4 and the previous consequence, it is guaranteed that the jobs can be processed somehow and that allocated resources can always be freed again.

**Consequence 3:** In keeping with the assumed independence of jobs, the set of job primitives is defined such that they can effect only the status of resources allocated to the job issuing the primitive. Therefore, a job cannot explicitly request or release resources on behalf of another job, nor can it do so implicitly by causing lower level primitives to change the allocation of other jobs in response to the needs of this job. This eliminates resource pre-emption.

**Consequence 4:** By construction, only active jobs can invoke primitive operations, and as stated above, these effect only resources allocated to that job. Therefore, once a request is made by an active job, no further requests and no releases affecting that job can occur until the request is satisfied and the job becomes active again.
**Consequence 1:** It is implicit in the graph that a job can validly retain control of all resources allocated at nodes on its path-history both when making a request to an earlier node and during any subsequent delay until that request is satisfied.

**Consequence 2:** There is no restriction on the order in which requests to a given node are satisfied, other than that all the resources in a single request must be allocated together as part of a single "assign" operation. This implies that each node can have its own queueing rule, such as FIFO, LIFO, Random, Priority, or any other non-unique ordering.

These assumptions are necessary and important to the problem of deadlock. For the following reasons: we may postulate the finite termination of jobs or of the entire system, whatever. In general, it is impossible to prove whether or not a queue or will terminate. The necessity for assuming independence and lack of any source of delay other than an unsatisfied resource request, is again one of making the problem solvable, since if forms a direct contradiction among the above, such that each depended on the other, then it is again impossible to prove, in the general case, that the job will, in fact, reach the state necessary to reactivate the other.

If virtual resources (assumption 2) can be created at will, then no deadlock is possible, since a request can always be satisfied by creating the necessary virtual resources. Therefore, any subset of virtual resources can be eliminated from consideration here. Similarly, resource elements which can only be shared simultaneously by jobs can be eliminated from discussion, as already mentioned in assumption 3. Consequence 2 implies that deadlock is not inherent in the use of resources to independent jobs, since a trivial allocation scheme always exists, but that it arises from the attempt to gain efficiency in allocating resources to 2 or more jobs simultaneously. Consequence 4 refers to the type of synchronization which exists between a job and the allocation scheme, and further indicates that only the actions of the allocator can cause a deadlock to arise.
Resource pre-emption also alleviates the problem of deadlock to a large extent since requests by priority jobs can always be satisfied by borrowing resources from lower priority jobs. The problem then reduces to allocating resources among jobs of equal priority (where pre-emption is impossible) or to reducing the cost of pre-emption to a minimum.
IV. Formal Properties

Notation: Associated with each job $J_i$ in the system $S$, will be a "demand vector" $D_i(t, P)$, which is a function of time $t$ and node $P$ to which it is directed. If $D_i(t, P) > 0$, then $J_i$ is active and $P$ is the node in whose output queue $J_i$ resides (the last node in which an allocation to $J_i$ was made). If $D_i(t, P) = 0$, then $J_i$ is waiting in the input queue to node $P$ for the number and type of resource elements assigned to node $P$ indicated by $D_i(t, P)$. $D_i(t, P)$ is the number of resource elements of type $J_i$ in the demand vector.

Each resource element of type $J_i$ is represented by $r_j(P)$, where $P$ is the node to which it is assigned. Then at time $t$, $r_j(P) \in J_i$ means that element $r_j(P)$ is allocated to job $J_i$ at time $t$. The notation $J_i \to J_k$ means that $D_i(t, P) > 0$ and $r_j(P) \in J_i$. In other words, $J_i$ is waiting for allocation of resource elements of type $J_k$, at node $P$, and at least one of these is presently allocated to $J_i$. The notation $P_i \to P_j$ means that there exists an arc in the graph from node $P_i$ to node $P_j$. $P_i \to P_j$ is defined recursively as $P_i \to P_j$ or $P_i \to P_k$, and $P_k \to P_j$. We first restate the finite termination condition (assumption 4) in terms of this notation.

Definition 1: (Finite termination) For each job $J_i$, if for every time $t$ such that $D_i(t, P) > 0$ there exists a time $t' > t$, such that $D_i(t', P) = 0$, then there exists a time $t' > t$ such that $J_i$ will terminate at time $t$.

Definition 2: At time $t$, job $J_i$ is in permanent wait if $D_i(t, P) > 0$ and for all $t' > t$, $D_i(t', P) > 0$.

Definition 3: A system $S$ is deadlocked at time $t$ if and only if there exists at least one job $J_i$ in permanent wait.

This is a stronger definition of deadlock than that proposed by Shoshani, since it requires only that demands could be satisfied at some time, whereas this definition requires that they, in fact, be satisfied. Therefore, "effective deadlocks", as described by Holt, are included in this definition, but excluded by Shoshani's. An example of this will be given shortly.
Definition 4: A finishing sequence $F(t)$ at time $t$ is a sequence of all jobs \( \{ J_i \} \) in $S$ such that for $t = t_1 \leq \ldots \leq t_n < \infty$

\[ t_i(t_i^*, P_i) = 0 \quad i = 1, 2, \ldots, n \]

Theorem 1: A system $S$ is deadlock-free at time $t$ if and only if there exists at least one finishing sequence $F$ of all its jobs.

Proof:

If: obvious, since the existence of $F$ implies that no job is in permanent wait.

Only if: since $S$ is not deadlocked, no job $J$ is in permanent wait by definition 3. Therefore, for each $J_i$ there exists some $t_i < t$ such that $D_i(t_i, P_i) = 0$, by definition 2. We, therefore, arrange the jobs $J$ in a sequence $F$ according to increasing order of their associated $t_i$, and this forms a finishing sequence, by definition 4.

Definition 2: The path history $H_i$ of job $J_i$ is the set of all nodes in which allocations have been made to job $J_i$ in the same order as the order in which the allocation was made.

A job can only make requests to nodes connected by an arc in the graph to the node in whose output queue it resides, and this is the node in which the last allocation was made, so that $H_i = \{ P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_i \}$ is simply the path followed by $J_i$ through the graph. Note that no node $P_k$ can appear in $H_i$ more than once, since there is no path through the graph which involves a cycle.

Definition 3: A circular wait condition exists at time $t$ if there exists a subset of jobs $J_1, J_2, \ldots, J_k$ in $S$ such that $J_1 \rightarrow J_2 \rightarrow \ldots \rightarrow J_k \rightarrow J_1$.

Theorem 2: A circular unit is impossible in systems described by the graph model.

Although this should be obvious from the acyclic nature of the graph, we first need 2 lemmas before proceeding with the formal proof.
Lemma 1: \( J_i \leftarrow J_k \) implies either \( P_i = P_k \) or \( P_i \rightarrow P_k \)

Proof:
if \( P_i = P_k \) there is nothing to prove
if \( P_i \not= P_k \), \( J_i \leftarrow J_k \) means there exists at least one element \( r_j(P_i) \in J_k \).
Thus, an allocation must have been made to \( J_k \) at node \( P_i \), so that \( P_i \)
is in \( J_k \) by definition 5. Let \( P_k' \) be the last node in \( H_k \); there are
2 cases.
1) \( E(P_i, P_k) = 0 \) then \( P_i \rightarrow P_k \) since when active, \( J_k \) resides in
the output queue of the last node in \( H_k \). Since \( P_i \not= P_k \), we must
have \( P_i \rightarrow P_k = P_k' \), by definition 5.
2) \( E(P_i, P_k) \) then \( P_i \not= P_k \), since \( J_k \) is waiting for a request, and
requests can only be made to nodes connected by an arc in the graph
to the node in which the job making the request resides. That is,
the node where the last allocation was made, which is \( P_k' \) by def-
nition 1. Either \( P_i = P_k' \) or \( P_i \not= P_k' \) by definition of \( H \), so
in both cases \( P_i \rightarrow P_k \).

Lemma 2: \( J_i \leftarrow J_k \) implies either \( P_i = P_k \) or \( P_i \rightarrow P_k \).

Proof:
if \( J_i \not= J_k \) then lemma 1 holds
let \( J_i \leftarrow J_k \) and suppose it holds for \( J_i \not= J_k \), then either \( P_i = P_k \)
or \( P_i \not= P_k \), by assumption. By lemma 1, \( J_i \leftarrow J_k \) implies either \( P_i = P_k \)
or \( P_i \not= P_k \).

There are four possibilities
1) \( P_i \rightarrow P_k \) then \( P_i = P_k \).
2) \( P_i = P_k \) then \( P_i \not= P_k \).
3) \( P_i \not= P_k \) then \( P_i \not= P_k \).
4) \( P_i \not= P_k \) then we can simply write out the meaning of \( \rightarrow \)
as \( P_i \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots P_i \rightarrow P_k \) and then reapply the
definition of \( \rightarrow \) to get \( P_i \rightarrow P_k \).

Proof of Theorem 2
Let the circular wait be \( J_i \leftarrow J_k \rightarrow J_1 \), with \( i \not= k \). By lemma 2, \( J_i \rightarrow J_k \)
implies either \( P_i = P_k \) or \( P_i \rightarrow P_k \).
If \( P_i \not= P_k \), then by lemma 1, \( J_i \rightarrow J_k \) implies either \( P_i = P_k \) or \( P_i \rightarrow P_k \)
Thus, either \( P_i \rightarrow P_k \) or \( P_i \not= P_k \), both of which form a cycle
in the graph. This is impossible since the graph is acyclic.

Similarly, if \( P_i = P_k \) and if lemma 1 applied to \( J_k \rightarrow J_1 \) gives the case
\( P_k \rightarrow P_1 \), then we also have \( P_i = P_k \rightarrow P_i \) which is an impossible cycle in
the acyclic graph.

- 15 -
This leaves only the case $P_{i_k} = P_{j_1} = P_i$, which must also be shown impossible. $J_{i_k} \rightarrow J_{i_1}$ means $r_{j_1}(t,p_{k(t)}) > 0$ and there exists $r_{j_1}(P_{j_1}) \in J_{i_1}$. Since $r_{j_1}(P_{j_1})$ is allocated to $J_{i_1}$ at node $P_{j_1}$, $P_{j_1}$ must be on the path history $K_i$ of $J_{i_1}$.

Now, look at $J_{i_1} \rightarrow J_{i_2}$, which means either $J_{i_1} \rightarrow J_{i_k}$ or $J_{i_1} \rightarrow J_{i_1} \rightarrow J_{i_2}$. In both cases, $D_{j_1}(t,p_{i_1}) > 0$ so that $J_{i_1}$ is waiting for an unsatisfied request to node $P_{i_1}$. Let $P_{k'}$ be the last node on the path history of $J_{i_1}$, which is necessarily connected by an arc to $P_{i_1}$, $P_{k'} \rightarrow P_{i_1}$. We cannot have $P_{k'} = P_{i_1}$, since this implies an arc in the graph from a node back to itself, and that does not exist. Since $P_{k'}$ is on the path history of $J_{i_1}$, we have either $P_{k'} \rightarrow P_{i_1}$ or $P_{k'} = P_{j_1}$. The latter is not possible since it implies $P_{i_1} = P_{k'} = P_{j_1}$, which we just noted is impossible.

However, $P_{i_1} = P_{k'} \rightarrow P_{i_1}$ gives us $P_{i_1} \rightarrow P_{i_1}$, which is a cycle in the graph, so that that is also impossible. Therefore, we cannot have $P_{i_1} = P_{k'}$ and hence a circular wait cannot exist.

We have not yet discussed the relationship between a circular wait and a deadlock. Although it should be obvious that a circular wait is one form of deadlock, there are, however, other forms of deadlock possible in the systems we are modeling, one of which we will call lockout. It is best explained by a simple example. Suppose a job $J_{i_1}$ is waiting in the input queue for some node, say $N_{j_1}$, for resources currently allocated to $J_{i_1}$. Before releasing these resources, $J_{i_k}$ creates a new job $J_{i_k'}$, which has resource demands identical to $J_{i_k}$ and which is immediately able to progress through the graph from the initial node to $N_{j_1}$. If the queueing scheme at $N_{j_1}$ is such that $J_{i_k'}$ enters the queue ahead of $J_{i_1}$ (say a LIFO rule, or a priority rule), then $J_{i_k'}$ will be allocated the resources released by $J_{i_k}$. If, in turn, $J_{i_k'}$ creates a new job $J_{i_k''}$ with the same characteristics, it is possible for $J_{i_1}$ to remain in the queue of node $N_{j_1}$ forever. Because such a situation does not "freeze" resources in the infinite control of one job, it is not considered to be a deadlock by Dijkstra, Habermann, or Shoshani, primarily because they require a finite number of jobs and job creations in their system. We will call such systems closed systems, and treat them as a special subcase.
A third form of deadlock, called "effective deadlock" by Holt, is discussed in [Holt]. It is caused by systems in which finishing sequences exist, but the allocation algorithm does not necessarily allocate resources in accordance with the requirements of these sequences. It can easily be shown that such situations cannot exist in closed systems as defined here, and that in the more general system the "effective" deadlock is identical with a lockout.

**Definition:** A closed system is one in which only a finite number of new jobs are created.

**Theorem 1:** In closed systems, a deadlock situation implies that more than one job is in permanent wait.

**Proof:** 

If \( J_1 \) is the only job in permanent wait, \( h_i(t, P) > 0 \) for all \( t > t_0 \). By definition, all other jobs in \( S \) will terminate eventually, and since the system is closed, there are only a finite number of these, so that by time \( t_p \), only job \( J_1 \) will remain in the system. By the definition of termination, all resource elements other than those allocated to \( J_1 \) will be free at time \( t_p \). Therefore, if the current demand of \( J_1 \), \( h_i(t_p, P) \), cannot be satisfied at time \( t_p \), \( J_1 \) must be demanding allocation of more resources than are assigned to node \( P \), which by assumption 2 is an illegal request. Therefore, such a job \( J_1 \) cannot exist.

**Theorem 2:** In closed systems, more than one job in permanent wait implies the existence of at least one circular wait.

**Proof:** 

Let \( J(t) \) be the set of all jobs in permanent wait at time \( t \). Since the system is closed, only a finite number of jobs will eventually be created, some of which may enter \( J \). Eventually however, at some time \( t_0 \), all jobs not in \( J \) will terminate, by definition 1, so that only jobs in \( J \) remain in \( S \). There must be at least two such jobs, \( J_1 \) and \( J_k \), by Theorem 3, so we can assume there are exactly two without loss of generality. Assume there is no circular wait between \( J_1 \) and \( J_k \). Since \( h_i(t, P_i) > 0 \) cannot be satisfied, there must be enough free resource elements in node \( P_i \) to satisfy it. By assumption 2, the total
number of resources required by this request, plus any already allocated
to J_k from P_k, cannot exceed the number assigned to P_i. Therefore, if
this request is legal, the resource elements needed to satisfy D_i(t_p,P)
must be allocated to some other job, which can be only J_k. However, J_k
is in a similar situation at node P_k, since its request D_k(t_p,P_k)
cannot be satisfied either. and the only source of this can be allocation of the nec-
essary resource elements to another job, namely J_1. Therefore, J_1→J_k
and J_k→J_1, so we have a circular wait J_1→J_k→J_1. This is a contra-
diction to our assumption of no circular wait. Hence, there must be
at least one circular wait in these systems.

**Theorem 5:** In closed systems, deadlock implies the existence of at least
one circular wait.

**Proof:**
Obvious from Theorems 3 and 4

**Theorem 6:** Any closed system represented by this graph model is free of
deadlock.

**Proof:**
Obvious from Theorems 2 and 5.

Finally, we can prove a theorem about systems that are not closed, but are
restricted in their choice of queuing disciplines.

**Theorem 7:** If all queues are ordered by a FIFO discipline, then systems
represented by this graph model are deadlock free.

**Proof:**
The idea is to show that the demand D(J,P) of the top element J in the
input queue for arbitrary node P will always be satisfied in a finite
time. Since the queue is FIFO, the relative position of jobs in the
queue does not change, so that as each top element is removed, each
job in the queue gets progressively closer to the top until each
eventually reaches the top and is also satisfied. Then no job at
node P will ever be in permanent wait. If this is true for all nodes
P (and hence all jobs) then no deadlock can exist, by definition 3.
Suppose P is a terminal node, and J is the top element in the input queue. Suppose J\textsubscript{k} controls resources needed by J, J→J\textsubscript{k}. Then by Lemma 1, P = P\textsubscript{k}, since there is no node following P in the graph. Therefore, J\textsubscript{k} is active, since if it were not, it would simultaneously have P\textsubscript{k} on its path history already (due to J→J\textsubscript{k}) and yet be in the input queue for P=P\textsubscript{k}, implying P→P which does not exist. Eventually J\textsubscript{k} will (1) issue a terminate or (2) issue a retrace which will carry it above P, both of which release all resources in node P allocated to J\textsubscript{k}, or (3) issue a release that frees those resource elements needed by J. Note that J\textsubscript{k} cannot issue a request without preceding it with at least one retrace, since P is a terminal node. This is true for any J\textsubscript{k} such that J→J\textsubscript{k}, so that by some time t'> t, enough resources in P will be freed to satisfy D(t,P), thereby removing J from the input queue.

Now suppose P is a non-terminal node, with J the top element in the input queue, and further suppose that all nodes P\textsubscript{k} below P in the graph satisfy the condition that no job is ever in permanent wait at that node. Let J\textsubscript{k} be some job, such that J→J\textsubscript{k}. By lemma 2, either P=P\textsubscript{k} or P→P\textsubscript{k}. In the first case, J\textsubscript{k} must be active (to avoid cycles in the graph), but in the second case J\textsubscript{k} may be waiting in the input queue of node P\textsubscript{k}. By assumption, this wait is not infinite since P→P\textsubscript{k}, so that eventually J\textsubscript{k} becomes active. This is true of any subsequent requests by J\textsubscript{k} as well. Therefore, eventually J\textsubscript{k} will (1) issue a terminate, or (2) issue a retrace above P, both of which release all resources in P allocated to J\textsubscript{k}, or (3) issue a release that frees those elements needed by J. Since this is true for all J\textsubscript{k} such that J→J\textsubscript{k}, and since no other job can be allocated these resources ahead of J, due to the FIFO discipline, eventually D(t,P) can be satisfied and J is removed from the input queue.
V. Representing Existing Systems in the Model

A. One Job at a Time

Most second-generation machines processed each job from start to
finish and never worried about deadlock problems. Obviously, a circular wait
could not occur, by definition, but lockout could occur if the job scheduler
chose a job in a LIFO or priority manner. However, once chosen a job essentially
had access to all the resources of the machine. This is represented trivially
by a graph consisting only of the initial node connected to a single allocator
node to which all resources are assigned. Each job is then allocated all the
resources at that node.

B. Job Class Scheduling

A simple way to multi-program is to assign resources to fixed job
classes, and then allocate all the resources in a class to each job in the class.
In the graph, this is represented by n nodes, one for each of the n classes,
each connected by an arc to the initial node. By specifying a class, each job
effectively selects the arc designated for its class and when it is chosen
from the input queue of the class node, is allocated all the resources assigned
to the node. Only one job per class is allowed to progress, but there is no
interference between classes.

C. Release Before Request

Another way to multi-program is to allow each job to request any re-
sources in any order, but before each new request it must release any resources
already allocated to it. This is simply a variation of the single-node graph
already discussed. Each job is allocated only those resources it requests when
they are available, according to some queuing scheme. Since all resource ele-
ments are assigned to the single allocator node, any subset is legal. In order
to change its allocation, the job must first issue a retrace, releasing all its
resources, and followed by the new request that returns it to the input queue
for the allocator node.
C. **Multics**

As described by Saltzer and Reppaport, the resource allocation aspects (but not the intertask communication) of the Multics time sharing system can be represented in the graph model (see Figure 2). "Eligibility" is considered a resource in order to control the number of jobs able to compete for storage, the next lower node in the graph. This minimizes the danger of thrashing, as discussed by Denning. As part of a paging fault, a job simply "retraces" back to the storage allocator to request a new page in addition to its current pages. The allocator is designed to try to keep the pages already allocated to the job reserved for it while attempting to locate a new page, but it is possible for a job to loose one of the pages allocated to it at the time of the fault, since, in essence, it has released them as part of the retrace. The "reservation" phenomenon is obviously a necessary adjunct for efficiency, since it is doubtful if the system could ever work without it.

Similarly, at the end of a time slice a job "retraces" back above the "eligibility node", releasing its CPU cycles, its storage pages/its eligibility, and returning to the input queue to await reallocation of "eligibility".

E. **OS 360/MVT**

As described by Havender, resource allocation under OS 360/MVT occurs in levels, each level containing all the resource elements of a particular class. As implemented, the first level consists of the job initiator/terminator, the second level consists of all the data sets known to the system, the third level is the storage region, the fourth is the I/O devices, and the fifth the CPU cycles. In the graph model, each level is represented as a node, with all resource elements of the appropriate type assigned to that node. There is a single input edge into the node from the level above it, and a single output edge to the next lower level. The functions defined on each node constitute the "resource class manager" in the OS system. The queuing rules vary from level to level,
It is summarized in Table 1. Initiators and data sets are allocated for the duration of a job, core storage and I/O devices are allocated by job step, and CPU cycles are allocated as needed within a job step execution. The completion of a job step results in a "retrace" back to the core allocation level (node), resulting in the often annoying loss of I/O devices, but not data sets, between job steps. The rationale for this ordering is explained by Havender.

TABLE 1

<table>
<thead>
<tr>
<th>Level</th>
<th>Resource Class</th>
<th>Queue Discipline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initiator/terminator</td>
<td>(Job Class, Priority,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FIFO)</td>
</tr>
<tr>
<td>2</td>
<td>Data Sets</td>
<td>FIFO</td>
</tr>
<tr>
<td>3</td>
<td>Core Storage Regions</td>
<td>(Priority, FIFO)</td>
</tr>
<tr>
<td>4</td>
<td>I/O Devices</td>
<td>FIFO</td>
</tr>
<tr>
<td>5</td>
<td>CPU Cycles</td>
<td>(Priority, FIFO)</td>
</tr>
</tbody>
</table>

FIGURE 2

[Diagram showing the process of eligibility, storage pages, and CPU cycles with retraces on an end-of-time slice and a paging fault.]