Some Deadlock Properties of Computer Systems

RICHARD C. HOLT

*University of Toronto,**
Toronto, Ontario, Canada

Several examples of deadlock occurring in present day computer systems are given Next, there is a discussion of the strategies which can be used to deal with the deadlock problem A theory of computer systems is developed so that the terms "process" and "deadlock" can be defined. "Reusable resources" are introduced to model objects that are shared among processes, and "consumable resources" are introduced to model signals or messages passed among processes. Then a simple graph model of computer systems is developed, and its deadlock properties are investigated This graph model unifies a number of previous results, leads to efficient deadlock detection and prevention algorithms, and is useful for teaching purposes.

Key words and phrases: deadlock, deadly embrace, knot, interlock, pre-emption, resource allocation, operating system, graph reduction

CR categories: 4 31, 4.32

1. COMPUTER SYSTEMS AND DEADLOCK

In this paper we present a simple graph model of computer systems and investigate its deadlock properties. This model unifies a number of previous results, leads to efficient deadlock detection and prevention algorithms, and is useful for teaching purposes.

Deadlock is the situation in which one or more processes in a system are blocked forever because of requirements that can never be satisfied. That is, deadlocked processes will remain blocked until special action is taken by some "external force" such as the operator or the operating system.

Let us consider a simple example of deadlock which can occur when there are two processes, $P1$ and $P2$, and two resources, $R1$ and $R2$. Assume that a resource cannot be released by (or pre-empted from) a process waiting for a request. Suppose $R1$ has been assigned to $P1$, and $R2$ has been assigned to $P2$. Now suppose $P1$ requests $R2$ and, $P2$ requests $R1$. The result is that $P1$ and $P2$ are deadlocked.

* Department of Computer Science and Computer Systems Research Group.

To a large extent, studying deadlock means studying the logic of process interactions in computer systems. At present, our ability to build large, reliable computer systems is something less than satisfactory; this situation can be attributed to our lack of understanding of how the integral parts of such systems interact. Examples of interactions among processes leading to system failure from deadlock have been described by various authors [9, 11, 13, 21]. It is hoped that this paper will make the problem of deadlock easier to understand and analyze, thereby improving our ability to build reliable computer systems.

This paper is organized in the following fashion. Section 2 gives examples of deadlock in present day computer systems. This is followed by a listing of the methods available for handling the deadlock problem. In Section 4 we present a theory of computer systems so that terms such as "process" and "deadlock" can be defined. Then, "reusable resources" are introduced to model physical objects (or objects behaving like physical objects) which are shared by proc-

CONTENTS

esses, and "consumable resources" are introduced to model messages or signals passed among processes. Section 8 presents a simple graph model of processes interacting via reusable and consumable resources. This is followed by development of necessary and sufficient conditions for detection and prevention of deadlock for various special cases of the general model. Using these conditions, efficient algorithms are developed for the detection and prevention of deadlock.

2. EXAMPLES OF DEADLOCK IN CURRENT SYSTEMS

One of the major advantages provided by operating systems is the ability to share resources among processes. Whenever resources can be requested and held by processes, deadlock is a potential problem; thus, the operating system designer should be well acquainted with solutions to this problem.

One might argue that deadlock is of little interest in current operating systems, and that the problem is apparently solved because it is seldom observed. It is true that deadlock caused by competition for devices, such as tape and disk drives, has been prevented by effective ad hoc procedures. (See [9] for a discussion of such procedures used in OS/360.) However, it is not at all obvious that these ad hoc methods are optimal; further work is required to clarify exactly what methods are possible and which are the least costly.

In spooling systems, such as ASP/OS/360 [16], deadlock sometimes occurs because of competition for spooling space on the disk. The problem arises when the spooling space becomes completely filled with input records for jobs waiting to execute and with output records for jobs not finished executing. There is no way to recover the spooling space occupied by a partially executed job (at least not in ASP/OS/360); the only way to recover from such a deadlock is to restart the system. The somewhat crude ad hoc solution to this problem is to prohibit (manually) the spooling of new jobs once the utilization of spooling space becomes too high, say above 80% utilization. It is obvi-

ous that this solution is costly in terms of idle spooling space, and it is not obvious that this solution approaches being optimal

Deadlock caused by faulty synchronization of processes is still not well understood. (See Saltzer's discussion of the lost wake-up problem [23], and Rappaport's discussion of why the original design of Block/Wake-up in MULTICS caused deadlock [21].) Deadlocks caused by faulty synchronization of processes can and do occur in systems such as OS/360. The current solutions to these problems are typified by the following two examples:

1) If a job in OS/360 begins waiting (via a Wait macro [13]) for an event, the system has no way of knowing if the event will ever occur. While the job is waiting for the event to occur, all resources allocated to the job, including core memory, will remain idle. To prevent waiting forever, OS/360 allows an arbitrary absolute limit of 30 minutes waiting time before canceling the job [15]. Why 30 minutes and not 5 minutes or 50 minutes?
2) The Enq-Deq facility in OS/360 allows processes to gain (by Enq) exclusive control of a resource, and then to release it (by Deq) [13]. The simple example of deadlock presented in Section 1 can occur using Enq-Deq and goes undetected by the operating system, thereby allowing deadlocked processes, together with the resources they hold, to remain idle for an indefinite amount of time.

These two examples illustrate the fact that in OS/360 methods of deadlock detection are, at best, rudimentary.

With the introduction of parallel processes, and operations to synchronize these processes in high-level languages such as PL/I [14], a large community of users has at its disposal the means to deadlock the processes. A hostile user of an OS/360 system has only to submit the following three-line PL/I program to cause a deadlock:

```
REVENGE: PROCEDURE OPTIONS(MAIN,TASK),
  WAIT(EVENT);
END REVENGE;
```

The only executable statement in this program is "WAIT(EVENT);", which causes the program to begin waiting for an event which will never occur. The user will not be charged for CPU or input/output use because the program employs neither. However, all resources allocated to this program, such as the core it occupies, will remain idle until the deadlock is removed by the operating system or by a keen-witted operator.

3. DEADLOCK STRATEGIES

In any computer system where processes share resources or pass signals, there must be a strategy, stated or not stated, to handle the problem of deadlock. We can characterize these strategies as belonging to one of three classes.

Prevention. The system is designed so that deadlock is not possible. (In terms of the definitions given in Section 4, the system is designed so that it is "secure" from deadlock.) A system which is not secure from deadlock can sometimes be made secure by prohibiting operations which may lead to deadlock. Habermann's policy of using maximum claims (see Section 13) is an example of such a prevention strategy.

Detection. Deadlocks can occur, but they are detected when they happen. When a deadlock is detected, the system can recover by terminating the deadlocked processes or by pre-empting resources from processes. This strategy can allow higher resource utilization than is possible when deadlock is absolutely prevented, and it should be used when deadlock is not too frequent and recovery is not too expensive. The algorithms given below should make deadlock detection practical in many cases where deadlock goes undetected in current systems. This strategy has not yet been used widely, but it offers the distinct advantage of a "soft fail" over the next strategy.

Crash. Deadlocks are possible and are not automatically detected. It is the responsibility of the operator to decide that a deadlock has occurred and to take steps to remove the deadlock. One might name this the "no strategy" strategy. However, this name is a

bit too unkind because this strategy saves the time and space required by deadlock prevention and detection algorithms.

The methods of handling the deadlock problem which are discussed below are examples of prevention and detection strategies.

4. BASIC DEFINITIONS

In order to investigate the problem of deadlock, we need to understand exactly what is meant by the terms "system" and "process."

We define a *system* as a pair (Σ, Π) where Σ is a set of *states* $\{S, T, U, V, W, \ldots\}$, and Π is a set of *processes* $\{P1, P2, P3, \ldots\}$. (We do not insist that Σ and Π be finite sets. In the system introduced in Section 8, Σ is infinite and Π is finite.) Each *process* P_i is defined as a mapping from the system states into the subsets of system states.

Figure 1 illustrates a system whose states are $\{S, T, U, V\}$ and whose processes are $\{P1, P2\}$. In this system, process $P1$ maps state S into $\{T, U\}$. We may interpret this to mean that when the system is in state S, process $P1$ may change the state to either T or U.

If process P_i maps state S into a subset of Σ containing state T, then we write

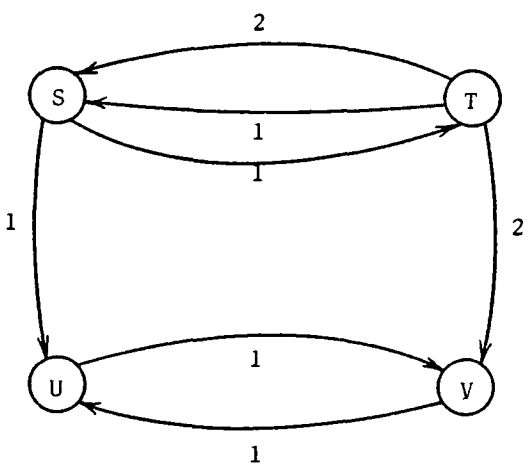

FIG. 1 Example of a system in which process $P2$ can deadlock. The set of states is $\Sigma = \{S,T,U,V\}$, and the set of processes is $\Pi = \{P1, P2\}$.

$$S \xrightarrow{i} T.$$

(Read "process P_i takes S to T.") We call the change of state from S to T an *operation* by process P_i. Operations in Figure 1 include the following: $S \xrightarrow{1} T$, $S \xrightarrow{1} U$, $T \xrightarrow{1} S$, $T \xrightarrow{2} S$, $T \xrightarrow{2} V$. If the system allows the following sequence of zero or more changes of state: $S \xrightarrow{i} T$, $T \xrightarrow{j} U, \ldots, V \xrightarrow{x} W$, then we write

$$S \xrightarrow{*} W.$$

For example, in Figure 1, $S \xrightarrow{1} T$ and $T \xrightarrow{2} V$; thus, $S \xrightarrow{*} V$.

The notion of "processes" in computer systems is well known [5, 12, 18, 23]. Our definition is noteworthy for two reasons. First, the definition is simple and precise, yet convenient for our purposes. Second, our definition allows processes to be nondeterministic; for example, in Figure 1 process $P1$ can change state S to either T or U. We may interpret this nondeterminism to mean that we may not be able to know (or may not even wish to know) exactly what each process will do next. For example, we may know that process $P2$ is presently capable of either requesting resource $R1$ or releasing resource $R2$, but we may not know which operation $P2$ will actually execute.

When process P_i can execute no operation, we say P_i is blocked; if P_i will never again be able to execute an operation, we say P_i is deadlocked. Formally:

> Process P_i is *blocked* in state S if there exists no state T such that $S \xrightarrow{i} T$. Process P_i is *deadlocked* in state S if for all T such that $S \xrightarrow{*} T$, P_i is blocked in T.

In Figure 1 process $P2$ is blocked (but not deadlocked) in state S, and process $P2$ is deadlocked in states U and V.

From these definitions it follows that process P_i is not deadlocked in state S if and only if there exists T such that $S \xrightarrow{*} T$ and process P_i is not blocked in T.

If one or more processes are deadlocked in

state S, then we say S is a *deadlock* state. If all processes are deadlocked in S, we say S is a *total deadlock* state. In Figure 1, U and V are deadlock states, but there are no total deadlock states.

We shall say that a state is secure if it cannot change (in any number of operations) to become a deadlock state. That is:

> State S is *secure* if for all T such that $S \xrightarrow{*} T$, T is not a deadlock state.

We shall say a system is *secure* if it contains one or more secure states.

The lemma given below follows immediately from the preceding definitions.

> Suppose $S \xrightarrow{*} T$. If S is a deadlock state, then T is a deadlock state. If S is a secure state, then T is a secure state.

Thus, deadlock and security are "permanent" conditions.

The definitions given in this section can be applied in various situations; for example, a process might be a production in a context-free grammar, or a set of transitions in a Petri net [10], or a chessman on a chessboard, or a vector in a vector addition system [17]. In the following sections, we shall apply these definitions to models of computer systems.

5. AN EXAMPLE OF A SIMPLE SYSTEM

We shall now show how these definitions can be used in a system consisting of two processes that share two identical units of a resource. Assume that each process is able to request only one unit of the resource at a time, and that a process holding units can release only one unit at a time. In this system, each process is in one of the following states:

0) The process holds no units and has requested no units.
1) The process holds no units and has requested one unit.
2) The process holds one unit and has requested no units.
3) The process holds one unit and has requested one unit.
4) The process holds two units and has requested no units.

(We can assume a process will not request another unit when it holds two units, because there is a total of two units.) A process can change states from 0 to 1 by requesting a unit, from 1 to 2 by being allocated a unit, from 2 to 3 by requesting a unit, and from 3 to 4 by being allocated a unit. A process can change states from 2 to 0 or from 4 to 2 by releasing a unit.

We represent each state of the system as S_{jk}, where j is the state of process $P1$ and k is the state of process $P2$. For example, if the state of $P1$ is 1 ($P1$ holds no units and has requested one unit) and the state of $P2$ is 4 ($P2$ holds both units), then the system state is $S14$. The possible changes of state in this system are illustrated in Figure 2.

In Figure 2 the vertical edges are operations by $P1$, and the horizontal edges are operations by $P2$. Edges directed to the right and edges directed down represent requests and acquisitions of units; edges directed to the left and edges directed up represent releases of units.

Process $P1$ is blocked whenever it has requested a unit but no more units are available. For example, in system state $S14$ process P1 has requested a unit, but both units have been allocated to process $P2$. In Figure 2 we can tell that $P1$ is blocked in $S14$ because the node $S14$ has no edges labeled "1" directed away from it.

Deadlock occurs in this system when one unit of the resource has been allocated to each process and each process requests one more unit. This situation occurs in system state $S33$. State $S33$ is a total deadlock state because both processes are deadlocked in $S33$. In Figure 2 we can tell that $S33$ is a total deadlock state because no edges are directed away from node $S33$.

Since deadlock state $S33$ can be reached from any other state there are no secure states in the system.

6. RESOURCES IN COMPUTER SYSTEMS

Processes in computer systems can interact *explicitly*, for example, by exchanging mes-

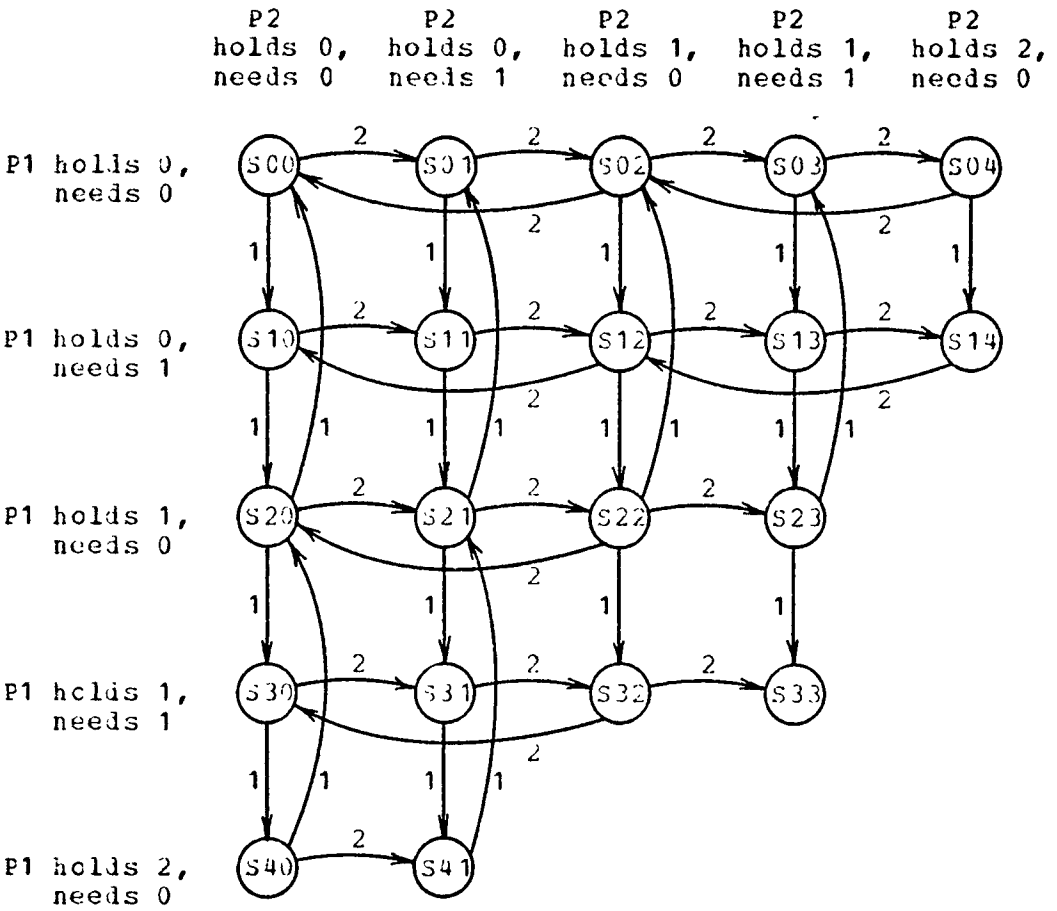

FIG 2 A system with two processes and a resource of two identical units.

sages, or *implicitly*, for example, by competing for physical objects such as tape drives. Either type of interaction may cause blocking of processes. We shall use the term "resource" in a special sense to mean any object which may cause a process to become blocked. "Reusable resources" will be used to model competition for objects, and "consumable resources" will be used to model exchange of signals or messages.

Both types of resources consist of a number of identical units which can be *requested* by processes. A process requesting units is blocked until enough units are available to satisfy its request; then the process can *acquire* the requested units. A process which is not waiting for a request can *release* units of resources, thereby making more units available.

Reusable resources have the following properties: There is a fixed total number of units of a reusable resource. Each unit of the resource either is available (not assigned) or has been acquired by (assigned to) a particular process. A particular unit of a resource can be assigned to, at most, one process at a time. A process can release any unit of a resource which the process has acquired but not yet released (assuming the process is not blocked waiting to acquire more units following a request). Units cannot be pre-empted; once a process has acquired a unit, the unit will not become available until released by the process.

(The term "reusable resource" has been borrowed from the term "serially reusable resource" used in IBM literature [9, 13]. Murphy [20] and Russell [22] investigate reusable resources whose units can be assigned to more than one process at a time.) Some examples of reusable resources follow.

EXAMPLE 1. The physical devices of the computer system, such as channels, core, tape drives, drums, and disks, can be reusable resources. The units of some of these resources will depend on the allocation strategies of the computer system; for example, disks may be assigned in units of tracks, or cylinders, or even entire disks.

EXAMPLE 2. Certain information structures shared by processes, such as records in a file and linkage pointers for buffer pools, are reusable resources. The processes must request, acquire, and release access to these information structures to guarantee that the structures can be inspected or updated without interference from other processes. (As Dijkstra [5] puts it, at most one process at a time should enter a "critical section" to inspect and update the information structure.)

Consumable resources have the following properties: There is no fixed total number of units of the resource. Every unit of the resource is available; if a unit is acquired by a process, the unit ceases to exist. Only a process which is a producer of the resource can release units of the resource; a producer is allowed to release any number of units of the resource at any time (assuming the producer is not blocked waiting to acquire units following a request). Any released units immediately become available.

Some examples of consumable resources are given below.

EXAMPLE 1. The card reader produces (releases) card images that are consumed (requested and acquired) by some process, probably the input spooling process. Thus, card images are a consumable resource.

EXAMPLE 2. In many systems, external interrupts are received by a special interrupt handling process that interprets the interrupt and passes a special type of message to another process which is waiting for that type of message. The interrupt handling process then cycles back to wait for the next external interrupt. The interrupt handling routine consumes the external interrupts and produces messages of various types. Thus, the external interrupts and the various types of messages are consumable resources.

The fundamental difference between reusable and consumable resources is that the units of a reusable resource are never created or destroyed, but only passed (requested and acquired) from a pool of available units to a process and then passed back (released) to the pool. By contrast, units of a consumable resource are created ("produced" or released) and destroyed ("consumed" or requested and acquired).

7. SOME DEFINITIONS FROM GRAPH THEORY

The following definitions will be used in our model of computer systems. A *directed graph* is a pair (N, E), where N is a nonempty set of *nodes* and E is a set of *edges*. Each edge in E is an ordered pair (a, b), where a and b are nodes in N. (For given nodes a and b, we will allow E to contain more than one edge of the form (a, b).) An edge (a, b) is said to be *directed from* node a and *directed to* node b. The graph is said to be *bipartite* if the set of nodes N can be partitioned into disjoint subsets Π and *RHO* such that each edge has one node in Π and the other node in *RHO*.

A *sink* is a node with no edges directed from it, and an *isolated node* is a node with no edges directed to or from it. If the graph contains edge (a, b), then node a is a *father* of b and node b is a *son* of a. A *path* is a sequence $(a, b, c, \cdots, r, s)$ containing at least two nodes, where (a, b), $(b, c), \cdots$, and (r, s) are edges. A *cycle* is a path whose first and last nodes are the same. (Definitions similar to the above are well known [2].)

It follows easily from these definitions that in a bipartite graph having no more than one edge directed from given node a to given node b, there are at most $2mn$ edges

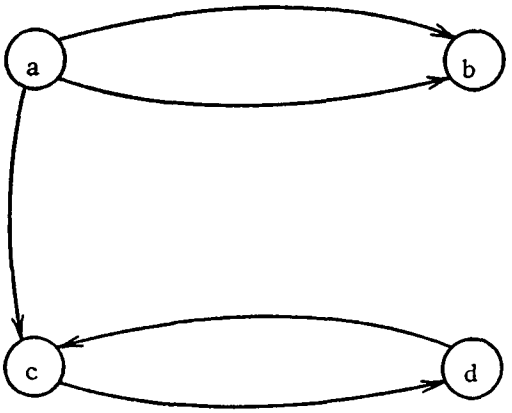

FIG 3. An example of a bipartite directed graph.

in E, where m and n are the numbers of nodes in RHO and Π.

The *reachable set* of node a is the set of all nodes b such that a path is directed from a to b. A *knot* is a nonempty set K of nodes such that the reachable set of each node in K is *exactly* set K. (We shall show that in certain cases, the existence of a knot in a system state graph is a necessary and sufficient condition for deadlock.) It can be shown that a graph does not contain a knot iff each node is a sink or has a path directed from it to a sink.

Figure 3 illustrates a bipartite graph with nodes $\{a, b, c, d\}$ and edges $\{(a, b), (a, b), (a, c), (c, d), (d, c)\}$. In this example, node b is a sink, path (c, d, c) is a cycle, and set $\{c, d\}$ is a knot.

8. GENERAL RESOURCE SYSTEMS

In this section we shall define a formal model of a system of interacting processes. Each state of the system will be represented by a directed graph having a node corresponding to each process and resource. Interactions in the system will be represented by edges drawn from process nodes to resource nodes or vice versa. To define the system we say a *general resource system* is completely characterized by:

1) a nonempty set of processes $\Pi = \{P1, P2, \cdots, P_n\}$;
2) a nonempty set of resources $RHO = \{R1, R2, \cdots, R_m\}$;
3) a partition of RHO into two disjoint subsets, a set of reusable resources and a set of consumable resources;
4) for each reusable resource R_j, a strictly positive integer t_j, called the *total units* of R_j; and
5) for each consumable resource R_j, a nonempty subset of the processes which will be called the *producers* of R_j.

The set of states Σ of a general resource system is the set of all general resource graphs for the system, which we define as follows:

A *general resource graph* is a bipartite directed graph whose disjoint sets of nodes are $\Pi = \{P1, P2, \cdots, P_n\}$ and $RHO = \{R1, R2, \cdots, R_m\}$, together with a nonnegative integer vector $(r1, r2, \cdots, r_m)$, called the *available units vector*. Edges directed from process nodes will be called *request edges*, edges directed from reusable resource nodes will be called *assignment edges*, and edges directed from consumable resource nodes will be called *producer edges*. Each general resource graph must have the following properties:

1) For a given reusable resource node R_j:
   a) the number of assignment edges directed from R_j cannot exceed the total units t_j;
   b) r_j (the available units) is equal to the total units t_j minus the number of assignment edges directed from R_j;
   c) for a given process node P_i, the number of request edges (P_i, R_j) plus the number of assignment edges (R_j, P_i) cannot exceed the total units t_j.
2) For a given consumable resource node R_j:
   a) there is a producer edge directed from R_j to process node P_i iff P_i is one of the producers of R_j;
   b) r_j (the available units) is any nonnegative integer.

Part (1c) of this definition implies that a process cannot request more than the total units of a reusable resource. Part (2b) implies that a system having consumable resources

will have an (countably) infinite number of states.

Figure 4 shows a state (a general resource graph) of a general resource system. In the figure, we have illustrated the fact that reusable resource $R3$ has three total units ($t3 = 3$) by drawing three subnodes inside the node for $R3$. Thus, the available units $r3$ of $R3$ is the number of units which do not have assignment edges drawn from them. We have illustrated the fact that consumable resource $R2$ has two available units by drawing two subnodes inside the node for $R2$. The producer edge ($R2$, $P2$) indicates that $P2$ is the only process capable of releasing units of $R2$. (The subnodes are not part of the formal definitions and are drawn only as a convenient way of representing the total and available units.)

The edges in Figure 4 can be thought of as "waits for" relations. For example, the edges from $P2$ to $R1$ mean that process $P2$ is waiting for units of resource $R1$. We could have drawn the edges in the opposite direction; in that case the edges would have represented "flow of units" relations. For example, edges from $R1$ to $P2$ would have meant that units of resource $R2$ must flow (be assigned) to process $P2$.

The processes in a general resource system map the system states into subsets of the system states; we will specify these mappings by describing the operations the processes can execute. There are three types of operations: requests, acquisitions, and releases. Essentially, the general resource graph (the system state) is changed: 1) by a request operation to have more request edges; 2) by an assignment operation to have fewer request edges and fewer available units; and 3) by a release operation to have more available units. For system states S and T, we define these operations as follows:

Requests. In state S if no request edges are directed from node P_i, then

$$S \xrightarrow{i} T$$

where S and T are identical except that in T there are one or more request edges directed from node P_i.

Acquisitions. In state S if there are request edges directed from node P_i, and for each resource R_j, the available units r_j is as large as the number of request edges directed from P_i to R_j, then

$$S \xrightarrow{i} T$$

where S and T are identical except that for each request edge (P_i, R_j) in S: 1) r_j is decreased by one; 2) if R_j is a reusable resource, then each request edge (P_i, R_j) is replaced by an assignment edge (R_j, P_i); and 3) if R_j is a consumable resource, then request edge (P_i, R_j) is deleted.

Releases. In state S if no request edges are directed from node P_i and some edges (assignment or producer edges) are directed to P_i, then

$$S \xrightarrow{i} T$$

where S and T are identical except that at least one resource R_j having one or more edges (assignment or producer edges) directed from R_j to P_i has its available units (r_j) increased; if R_j is a reusable resource, then there are deleted a number of assignment edges (R_j, P_i) equal to the number by which r_j is increased.

From the definitions of these operations, it follows that a process P_i is blocked if and only if there is a resource node R_j such that

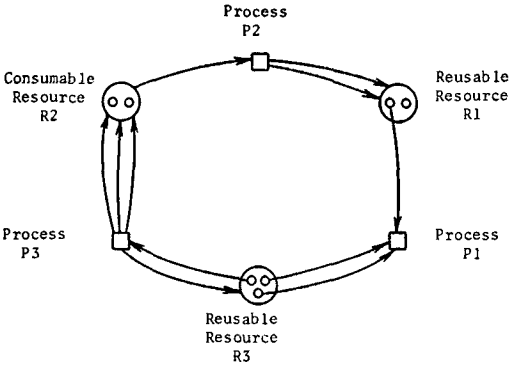

FIG 4 Example of a state in a general resource system. $R1$ and $R3$ are reusable resources whose total units are $t1 = 2$ and $t3 = 3$ $R2$ is a consumable resource whose only producer is $P2$. The available units are $r1 = 1$, $r2 = 2$, and $r3 = 0$ The state is not deadlocked because it can be reduced by $P1$, then by $P2$, and then by $P3$

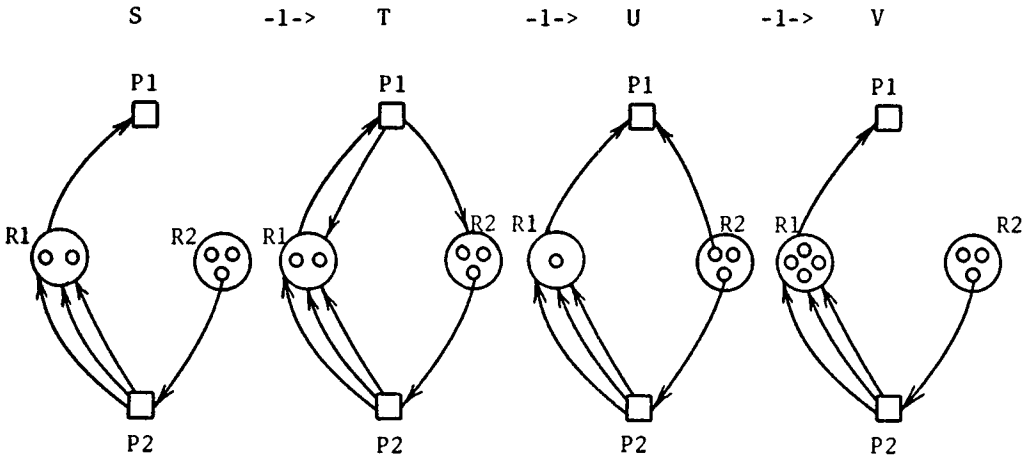

FIG 5 Examples of operations in a general resource system The system consists of consumable resource $R1$ (whose only producer is process $P1$), reusable resource $R2$ (which has three total units), and processes $P1$ and $P2$ Process $P1$ changes the state first by requesting one unit of each resource, next by acquiring the two units, and finally by releasing three units of $R1$ plus the acquired unit of $R2$ Process $P2$ is blocked in S because it has requested more units than are available, but $P2$ is no longer blocked in state V

the number of request edges directed from P_i to R_j exceeds r_j, i.e., when P_i has requested more units than are presently available.

Figure 5 contains examples of operations in a general resource system. When the system is restricted to having only reusable resources, these operations are equivalent to operations which Shoshani [24] developed previously and independently for a matrix-based model of computer systems.

9. NECESSARY AND SUFFICIENT CONDITIONS FOR DEADLOCK

In Section 4 we defined the terms "system" and "deadlock," and in Section 8 we gave an example of a system, namely, a general resource system. In this section we will use these definitions to develop necessary and sufficient conditions for deadlock in general resource systems.

A process is deadlocked when there is no way for the process to become not blocked. Thus, if we are able to find some sequence of operations which leaves a process not blocked, then we have shown that the process is not deadlocked.

We shall introduce sequences of "graph reductions" as a method of testing to see if processes are deadlocked. A graph reduction (by a particular process P_i) corresponds to the best set of operations which P_i can execute to help unblock other processes; this is equivalent to forcing P_i to release as many units as possible.

In order to define reductions we need a special symbol, *OMEGA*, which may be thought of as an "infinitely large" positive integer:

> *OMEGA* is a symbol such that for any integer i, $OMEGA > i$ and $OMEGA + i = OMEGA - i = OMEGA$.

During reductions, we will allow the available units r_j of a consumable resource to assume the value *OMEGA*. When a reduction assigns the value *OMEGA* to r_j, this can be interpreted to mean that enough units of the resource could be released to satisfy all subsequent requests. The definition of general resource graphs required that each consumable resource R_j have a nonempty set of producers; however, we shall adopt the convention that if $r_j = OMEGA$, then R_j may have no producers.

We define reductions as follows: A general resource graph can be *reduced* by any process

node which is not an isolated node and which is not blocked. The *reduction* by P_i

1) for each reusable resource node R_j, deletes all edges (P_i, R_j) and (R_j, P_i) (for each assignment edge (R_j, P_i) deleted, r_j is increased by one); and
2) for each consumable resource node R_j
   a) decrements r_j by the number of request edges directed from P_i to R_j,
   b) sets r_j to *OMEGA* if P_i is a producer of R_j, and
   c) deletes all edges (P_i, R_j) and (R_j, P_i).

We say the graph is *completely reducible* if a sequence of reductions deletes all edges in the graph. (See Figure 6 for examples of reductions.)

Under the convention that a consumable resource R_j need not have any producers when $r_j = OMEGA$, every reduction of a general resource graph leaves another general resource graph.

THEOREM 1. Process P_i is not deadlocked in a general resource graph S iff a sequence of reductions applied to S leaves a state in which P_i is not blocked.

ARGUMENT. First assume P_i is not deadlocked in S. Then it must be that $S \xrightarrow{*} T$ such that P_i is not blocked in T. Let *SEQ* be the sequence of processes whose operations change S to T. *SEQ* can be modified to describe a sequence of reductions which changes S to a state in which P_i is not blocked as follows: 1) delete each process in *SEQ* that has an isolated node in S; and 2) delete all but the first occurrence of each process in *SEQ*. It can be shown that, from state S, any sequence of operations by a given set of processes will result in, at most, as many available units as would a sequence of reductions by the processes in the same set. Thus, each reduction by a process in the modified *SEQ* sequence will result in at least as many available units as would the corresponding operation in the unmodified sequence. This implies that each reduction by a process in the modified sequence will result in enough available units so that the succeeding process will not be blocked and can be reduced. Hence, if P_i is not deadlocked in S, then S can be reduced to a state in which P_i is not blocked. Now assume a sequence *SEQ* of reductions changes S to state U in which P_i is not blocked. A reduction by any process P_j can be "simulated" by an acquisition and a release (or just a release) by P_j. Thus, the sequence *SEQ* of reductions can be "simulated" by a sequence of operations which changes state S to T in which P_i is not blocked; hence, P_i is not deadlocked in S.

Theorem 1 means simply that if there is a sequence of moves leaving P_i unblocked, then such a sequence can be found using graph reductions. From Theorem 1, the following corollary is easily proved:

COROLLARY 1. If a general resource graph is completely reducible, then it is not a deadlock state.

This follows from the observation that a complete reduction deletes all edges, including all request edges; thus, in the completely reduced state, no process is blocked.

Unfortunately, neither Theorem 1 nor its corollary suggests a fast method of testing for deadlock; the author knows of no better deadlock detection algorithm than a near exhaustive checking of the $n!$ different possible reduction sequences. The reason a fast deadlock detection algorithm has not been found is that reductions involving consumable resources may decrease the available units; consequently, the order of reductions is important. In Sections 10 and 12 this problem is avoided by imposing certain restrictions on the model, and fast detection algorithms are developed.

Now let us define an important type of state for which there is a simple sufficient condition for deadlock: An *expedient state* is a state in which all processes having requests are blocked. Any state which is not expedient will become expedient if all possible requests are granted.

Many computer systems use an "expedient" resource allocation strategy in which all requests for available units are granted. In such a system, units are assigned only immediately following requests and releases, and the system state is always expedient

(except for the brief intervals before assignments are made).

THEOREM 2. In a general resource graph:
1) a cycle is a necessary condition for deadlock; and
2) if the graph is expedient, then a knot is a sufficient condition for deadlock.

ARGUMENT. If the graph contains no cycle, then there must exist a linear ordering of the processes which has the following property: If a path in the graph is directed to P_i from P_j, then P_i appears before P_j in the linear ordering.

It can be shown that if processes having isolated nodes are deleted from this ordering, then the ordering gives a sequence of reductions that will completely reduce the graph. Thus, if there is no cycle, then the graph is completely reducible; hence, the graph is not a deadlock state.

If an expedient graph contains a knot, then the processes in the knot are all blocked waiting for units of resources in the knot, and the resources in the knot can have their available units increased only by (blocked) processes in the knot. Hence, all processes in the knot are deadlocked and the state is deadlocked.

If all possible requests have not been granted, i.e., if the state is not expedient, the general resource graph may contain a knot and still not be a deadlock state. For example, state T in Figure 6 contains knot $\{P1, R2, P2, R1\}$ and still is not a deadlock state.

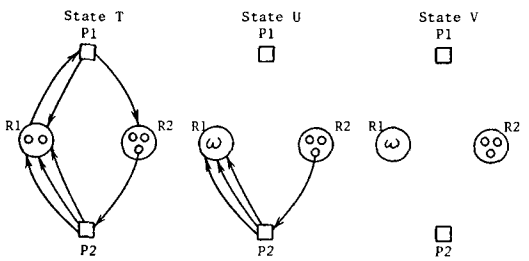

FIG 6 Reductions of a general resource graph. (State T is identical to state T in Figure 5.) $R1$ is a consumable resource, and $R2$ is a reusable resource. State T is reduced by $P1$ to obtain U, and U is reduced by $P2$ to obtain V. T is completely reducible because the reductions delete all edges, therefore, T is not a deadlock state

The following corollary of Theorem 2 can be proved using the graph properties of knots.

COROLLARY 2. Suppose the general resource graph is expedient. If P_i is not a sink and no path is directed from P_i to a sink, then process P_i is deadlocked.

In the next three sections we shall show that for important special cases of general resource systems, there are simple necessary and sufficient conditions for deadlock, and for security from deadlock.

10. GENERAL RESOURCE SYSTEMS WITH SINGLE UNIT REQUESTS

We shall impose the following restriction on the model.

Single Unit Requests. A process may request only one unit at a time. In the general resource graph this means that at most one request edge may be directed from any process node.

The restriction has the practical advantage of simplifying the algorithms used to implement request and acquire operations.

THEOREM 3. An expedient general resource graph with single unit requests is a deadlock state iff it contains a knot.

ARGUMENT. Since Theorem 2 states that a knot is a sufficient condition for deadlock, we have only to show that when there are only single unit requests, a knot is a necessary condition for deadlock. If the graph does not contain a knot, then from any blocked process's node there exists a path directed to a sink, and such a sink is necessarily a process node. Any such blocked process can be shown to be not deadlocked by showing that the graph can successively be reduced by each process on the path, starting from the sink and working backward. Hence, when there is no knot, no process is deadlocked and the state is not deadlocked.

The following results are closely related to Theorem 3. Let S be an expedient general resource graph with single unit requests:

1) S is not a deadlock state iff S is completely reducible.
2) Suppose different sequences of reductions applied to S result in states which cannot be reduced. Then all these resulting states are identical.
3) Process P_i is not deadlocked in S iff node P_i is a sink or has a path directed from it to a sink.

(See reference [11] for discussion and proofs of these results.)

The requirement for single unit requests in Theorem 3 is essential, as is shown by the following example. Consider a system having two consumable resources $R1$ and $R2$ whose only producers are $P1$ and $P2$, respectively. Let us suppose that no requests are pending and no units are available. Now suppose that there is a multiple unit request by process $P1$ for one unit of $R1$ and one unit of $R2$. As a result, $P1$ will be deadlocked, but the general resource graph will not contain a knot.

Theorem 3 and its related results are important because they imply that efficient deadlock detection algorithms are available for this special case of general resource systems.

Algorithm 1 can be used to detect if a graph is a deadlock state by testing to see if the graph contains a knot. The algorithm works by successively making all fathers of sinks into sinks; the graph will not have contained a knot iff all nodes become sinks. Algorithm 1 can be thought of as a simplified mechanism for successively reducing the graph.

ALGORITHM 1. Determination of whether a directed graph contains a knot. This can be used to detect if an expedient general resource graph with single unit requests is deadlocked.

```
1.  Do for each node Q on list of sinks;
2.      Do for each father F of Q;
3.          If F is not already on list of
                sinks
4.          Then add F to list of sinks;
5.      End,
6.  End;
7.  Knots = (Not all nodes are now on
            list of sinks);
```

Algorithm 2 can be used to determine if a particular blocked process is deadlocked. It works by systematically tracing out all paths leading from the process's node. The process will not have been deadlocked iff some path leads to a sink.

ALGORITHM 2. Determination of whether blocked process P is deadlocked in an expedient general resource graph with single unit requests.

```
        /*Switch D will tell if P is
              deadlocked.*/
1.  Set switch D to say P is deadlocked;
2.  Initialize a list to contain only P;
3.  Do for each node Q on list while D
          says P is deadlocked;
4.      Do for each son S of Q;
5.          If S is a sink
6.          Then set switch D to say P
                  is not deadlocked;
7.          If S has not yet been added
                  to list
8.          Then add S to end of list;
9.      End;
10. End;
```

The maximum execution times of both Algorithms 1 and 2 are proportional to the total number of edges in the graph. This can be shown by observing that the inner Do-group of each algorithm is executed at most once for each edge in the graph.

If more than one edge is directed from a given node to another given node, then these edges can be represented by a single edge together with an integer giving the number of edges represented. We will say this alternate representation uses *weighted edges*. Given that the graph uses weighted edges, then the number of edges is at most $2mn$ (see Section 7), and the maximum execution times of Algorithms 1 and 2 are proportional to mn. (m and n are the numbers of resources and processes.) This fast execution time means the algorithms can be used in practical systems.

It may be desirable in some systems to test for deadlock *continually*, i.e., after each operation which can cause a deadlock. It can be shown that if the system has only single unit requests and if requested available units are immediately acquired, then

the only operation that can cause deadlock is a request for an unavailable unit [11]. Such a deadlock will necessarily involve the requesting process; hence, continual deadlock detection can be accomplished by applying Algorithm 2 to any process requesting an unavailable unit. This detection method has been incorporated into a simple language which is being used for teaching concurrent programming [7].

Some of the overhead required by continual deadlock detection can be avoided by testing for deadlock only *occasionally*, say every 20 minutes, or when some process has been blocked for a suspiciously long time. This occasional test can be accomplished efficiently by Algorithm 1.

11. CONSUMABLE RESOURCE SYSTEMS

We shall now discuss another special case of general resource systems, which we call *consumable resource systems*, in which there are only consumable resources. All interactions in such systems are explicit; we may consider that processes can interact only by producer-consumer relationships.

We shall associate with each consumable resource R_j a set of processes, called the *consumers* of R_j. We will assume that every process is a producer or consumer of at least one resource. We define a *claim limited consumable resource system* as a consumable resource system from which has been eliminated each request by process P_i for resource R_j such that P_i is not one of the consumers of R_j.

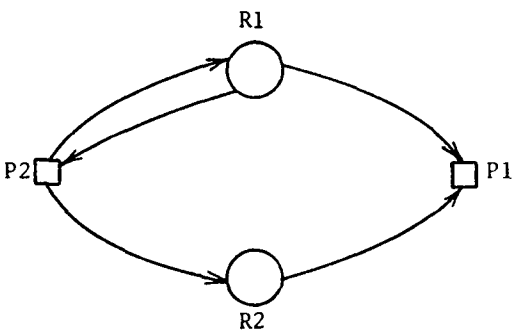

FIG 7. Example of a claim limited graph for a claim limited consumable resource system

We can now show how to characterize a claim limited consumable resource system by one particular graph, and how to determine if the system is secure from deadlock. For a given claim limited consumable resource system, the *claim limited graph* is the state of the system having: 1) zero available units, and 2) a request edge (P_i, R_j) iff P_i is a consumer of R_j.

(Part (2a) of the definition of general resource graphs in Section 8 describes the producer edges in the claim limited graph.) Figure 7 is an example of a claim limited graph.

A claim limited graph completely characterizes a claim limited consumable resource system since from it we can determine the processes, the resources, and the sets of producers and consumers. For example, the claim limited graph in Figure 7 shows that the processes are $\{P1, P2\}$, the resources are $\{R1, R2\}$, the producers and consumers of $R1$ are $\{P1, P2\}$ and $\{P2\}$, respectively, and the producers and consumers of $R2$ are $\{P1\}$ and $\{P2\}$, respectively.

We can use the following theorem to determine if deadlock is possible in a claim limited consumable resource system.

THEOREM 4. A claim limited consumable resource system is secure iff its claim limited graph is completely reducible.

ARGUMENT. Let the claim limited graph be called V. First, it can be shown that any sequence of reductions of V leads to a unique graph which cannot be reduced. It can then be shown that V (now considered to be a state in the system) is not deadlocked iff it is completely reducible. The proof is completed as follows. It is assumed that V is not completely reducible (thus, V is a deadlock state), and it is shown that for any (supposedly) secure state S, $S \xrightarrow{*} V$. Then it is assumed that a sequence of reductions completely reduces V, and it is shown that a similar sequence completely reduces any state in the system.

The claim limited graph in Figure 7 is completely reducible by the sequence $(P1, P2)$. Hence, the system is secure; i.e., neither $P1$ nor $P2$ can deadlock.

In theory, Theorem 4 could be used to determine whether a simple computer system could deadlock. In practice, the condition for security (complete reducibility of the claim limited graph) is too strong and could probably not be met. This means that the processes in a practical system can avoid deadlock by regulating their use of resources according to some knowledge of the system state; however, the theory makes no assumptions about such "intelligent" behavior by processes.

12. DEADLOCK DETECTION IN REUSABLE RESOURCE SYSTEMS

We shall now consider the special case of general resource systems, which we call *reusable resource systems*, in which there are only reusable resources.

THEOREM 5. Let S be any state of a reusable resource system.

1) S is not a deadlock state iff S is completely reducible.
2) Suppose different sequences of reductions applied to S result in states which cannot be reduced. Then all these resulting states are identical.

ARGUMENT. Part (2) follows from the observations: 1) that a reduction can never decrease the available units; and 2) that a reduction by given process P_i will delete the same edges regardless of which reductions were done previously. Since Corollary 1 (see Section 9) states that complete reducibility is a sufficient condition for a state not to be deadlocked, we need only show that for this case complete reducibility becomes a necessary condition. The required proof follows from the fact that when the graph is not completely reducible, blocked processes which cannot be reduced are deadlocked.

If each reusable resource has exactly one total unit, it can be shown that the conditions of deadlock, "not complete reducibility," and "existence of a cycle in the graph" become equivalent [11].

Part (2) of Theorem 5 implies that a "reasonably fast" deadlock algorithm exists for reusable resource systems. The algorithm successively reduces the graph as long as possible; the original graph will not have been deadlocked iff these reductions delete all edges. Since every sequence of reductions will lead to the same final graph, the algorithm will not need to backtrack.

Such an algorithm will be slowed since following each reduction a search must be made to determine which process (if any) to reduce by next. Algorithm 3 avoids this searching by assuming that the representation of the system state uses "wait counts" and "ordered requests" in the following manner: 1) for each process there is a *wait count* which gives the number of resources whose available units are less than those requested by the process; and 2) for each resource there is a list of processes requesting the resource, the list being in order by the number of units requested.

Algorithm 3 works by successively reducing (in Do-group 1) by processes which have zero wait counts and nonzero allocations. Each reduction increases the available units of at least one resource (in statement 3); this in turn may result in decreasing the wait counts (in statement 5) of other processes requesting the resource. Notice that no searching is required in statement 4 to locate "process Q" because "process Q" will be the next process on the ordered list of processes requesting resource R. Notice also that the total number of executions of Do-group 4 is limited by the number (m) of ordered lists of requests multiplied by the maximum number (n) of processes on each list. Thus, maximum execution time of Algorithm 3 is proportional to mn because Do-group 1 is executed at most n times (once for each process), Do-group 2 is executed at most mn times (once for each weighted assignment edge), and Do-group 4 is executed at most mn times (once for each weighted request edge). (Russell [22] has concurrently and independently developed an equivalent algorithm with this same maximum execution time. Shoshani [4, 24] has developed an equivalent algorithm which does not use wait counts or ordered requests and which requires maximum time mn^2.)

ALGORITHM 3. Testing to see if a state in a

reusable resource system is completely reducible. The "list to be reduced" is initialized to contain those processes with zero wait counts and some allocations. Processes with zero wait counts and no allocations are considered already reduced.

```
1.  Do for each process node P on list
        to be reduced;
2.      Do for each resource R assigned
            to P;
3.          Increase available units of
                R by units assigned
                to P;
4.          Do for each process Q whose
                request for R can
                now be granted;
5.              Decrease wait count of
                    Q by 1;
6.              If the wait count of Q
                    is zero
7.              Then add Q to list to
                    be reduced;
8.          End;
9.      End;
10. End;
11. Completely reducible = (All process
        nodes are now reduced);
```

While it might appear to be costly to maintain the wait counts and ordered lists of requests during system operation, there is a significant advantage in doing so. The advantage being that following a release, no searching is necessary to find processes which have become able to acquire their requested units; these processes will be exactly the ones whose wait counts become zero as a result of the release.

It can be shown that only requests for unavailable units can cause deadlock in reusable resource systems. Therefore, to maintain continual deadlock detection, one need apply Algorithm 3 only when unavailable units are requested. The request will have caused a deadlock only if the requesting process has become deadlocked; thus, Algorithm 3 can be stopped (with the conclusion that deadlock has not occurred) as soon as the requesting process's wait count reaches zero.

Processes interact only via reusable resources generally when the processes represent nominally independent user jobs in a batch-processing system. In such a system, Algorithm 3 can be used to test for deadlock; if deadlock has occurred, it will be necessary to terminate jobs or to pre-empt resources from jobs.

13. DEADLOCK PREVENTION IN REUSABLE RESOURCE SYSTEMS

Habermann [8] has shown how to prevent deadlock in reusable resource systems in which a maximum limit (a claim) is placed on each process's need for resources. We will briefly discuss Habermann's prevention method, showing how Algorithm 3 can be used to improve his original algorithm.

We define a *claim matrix* C as an n by m matrix where C_{ij} gives the maximum number of units of resource R_j which will be required by process P_i. We require that $0 \leq C_{ij} \leq t_j$. For given process P_i, we require that $C_{ij} > 0$ for at least one resource R_j; this means every process can request at least one unit of one resource. We define a *claim limited reusable resource system* as a reusable resource system from which has been eliminated each request by process P_i which (for some R_j) causes the number of request edges (P_i, R_j) plus the number of assignment edges (R_j, P_i) to exceed C_{ij}. That is, we assume processes never request more than they claim.

For a given state in a claim limited reusable resource system, the *claim limited graph* is constructed from the original graph (state) by adding (for all i and j) request edges (P_i, R_j) until the number of request edges (P_i, R_j) plus the number of existing assignment edges (R_j, P_i) is equal to C_{ij}.

Intuitively, the claim limited graph for a given state is formed by having all processes request as many units as allowed by their claims.

To prevent deadlock, one must guarantee that even though all processes request as many resources as allowed by their claims deadlock will not occur. Thus, one might expect that deadlock will be prevented if the system is never allowed to enter a state whose claim limited graph represents a dead-

lock state. Essentially, this is what Theorem 6 states.

Habermann's method employs the operating system's ability to determine when to grant which requests. In terms of our formal definitions, this means that some of the state transitions defined by acquire operations can be eliminated by the operating system. We define an *acquisition policy* as a rule which eliminates some of the acquisition operations of the system, thereby creating a new system which is secure. ("Secure" was defined in Section 4.) We shall say an acquisition policy is *optimum* if there exists no other acquisition policy which eliminates fewer acquisitions. Finally, an acquisition operation is said to be *safe* if it results in a state whose claim limited graph is completely reducible.

THEOREM 6. The optimum acquisition policy for a claim limited reusable resource system is the one that eliminates acquisitions which are not safe.

ARGUMENT. The theorem is proved by showing: 1) that if the acquisition policy allows an acquisition which is not safe, then necessarily a sequence of operations exists which leads to a deadlock; and 2) that if only safe acquisitions are allowed, then any state whose claim limited graph is completely reducible is not a deadlock state and can be changed only to similar states. (The proof is discussed in detail elsewhere [8, 11].)

Theorem 6 means that deadlock can be prevented (while granting as many requests as possible) by refusing to grant requests when the resulting state "may lead to deadlock." We determine if a state "may lead to deadlock" by seeing if its claim limited graph is completely reducible; this can be accomplished efficiently by Algorithm 3. Thus, Algorithm 3 is useful both for detecting and for preventing deadlock.

Habermann's algorithm to determine if an acquisition is safe is equivalent to Algorithm 3, but requires maximum execution time proportional to mn^2 instead of mn required by Algorithm 3. His algorithm is used to prevent deadlock caused by competition for plotter and paper tape punches in the "THE" multiprogramming system [19].

14. CONCLUSION

We introduced reusable and consumable resources to model interactions among processes in computer systems. It was shown that the technique of graph reductions can be used 1) to determine if a state in a system having reusable and consumable resources is deadlocked; 2) to determine if a system with only consumable resources is secure from deadlock; and 3) to implement deadlock prevention in a system with only reusable resources. Graph reductions are easy to understand, and this wide range of uses in investigating the deadlock problem attests to their importance.

The fast execution times of Algorithms 1, 2, and 3 indicate they can be used in practical systems for detecting and preventing deadlock.

In general, the results presented here, and the methods used to obtain them, do not depend greatly upon the exact definitions of "resources" and "operations." The message-passing operations of the SUE operating system [1] (being developed at University of Toronto) have been shown to have deadlock properties analogous to those described in this paper [25]. Similar results can be obtained for systems in which interactions are a result of P and V operations [6], Wake-up and Block operations [18], Wait, Post, Enq, and Deq operations [13], or Send Message, Wait Message, Send Answer, and Wait Answer operations [3].

This paper has been based on material in the author's PhD thesis; those interested in a more leisurely and complete treatment of the material should refer to the thesis [11].

REFERENCES

1. ATWOOD, J. W.; CLARK, B. L.; GRUSHCOW, M. S., HOLT, R. C.; HORNING, J. J.; SEVCIK, K. C.; AND TSICHRITZIS, D. "Project SUE status report." Computer Systems Research Group, Technical Report CSRG-11, Univ. Toronto, Toronto, Ontario, Canada, April 1972.
2. BERGE, CLAUDE. *The theory of graphs* John Wiley & Sons, New York, 1962 (originally published in French in 1958).
3. HANSEN, PER BRINCH "The nucleus of a multiprogramming system" *Comm. ACM* **13**, 4 (April 1970), 238–241, 250.

4. COFFMAN, E. G.; ELPHICK, M. J.; AND SHOSHANI, A. "System deadlocks." *Computing Surveys* **3,** 2 (June 1971), 67–78.
5. DIJKSTRA, E. W. "Cooperating sequential processes." Technological Univ., Eindhoven, The Netherlands, Sept. 1965.
6. DIJKSTRA, E. W. "The structure of the "THE" multiprogramming system." *Comm. ACM* **11,** 5 (May 1968), 341–346.
7. DRYER, MATTHEW. "User's manual for TOPPS." Computer Systems Research Group, Univ. Toronto, Toronto, Ontario, Canada, 1972.
8. HABERMANN, A. N. "Prevention of system deadlocks." *Comm. ACM* **12,** 7 (July 1969), 373–377, 385.
9. HAVENDER, J. W. "Avoiding deadlock in multitasking systems." *IBM Systems J.* **7,** 2 (1968), 74–84.
10. HOLT, A. W., AND COMMONER, F. "Events and conditions." *Record of the project MAC conference on concurrent systems and parallel computation*, ACM, June 1970, 3–52.
11. HOLT, RICHARD C. "On deadlock in computer systems." PhD Thesis, Cornell Univ., Ithaca, N.Y., Jan. 1971 (Reproduced as CSRG Technical Report 6, Dept. Computer Science, Univ. Toronto, Toronto, Ontario, Canada.)
12. HORNING, J. J., AND RANDELL, B. "Structuring complex processes." Report RC2459, IBM Research Lab., Yorktown Heights, N.Y., May 1969.
13. IBM System/360 operating system: supervisor and data management services. IBM Form No. C28-6646-2, IBM Corp., Nov. 1968.
14. IBM System/360 PL/I reference manual. IBM Form No. C28-8201-1, IBM Corp., March 1968.
15. IBM System/360 operating system: job control language. IBM Form No. C28-6359-8, IBM Corp., Nov. 1968.
16. System/360 attached support processor system (ASP) (360S-CX-15X) version 2· system manual. IBM Form No. Y20-0305-0, IBM Corp., Dec. 1968.
17. KARP, RICHARD M.; AND MILLER, RAYMOND E. "Parallel program schemata." *J. Computer & System Sciences* **3,** 2 (May 1969), 147–195.
18. LAMPSON, B. W. "A scheduling philosophy for multiprocessing systems." *Comm. ACM* **11,** 5 (May 1968), 347–360.
19. MCKEAG, R. M. "The multiprogramming system." Queen's Univ. Belfast, Dept. Computer Science, Belfast, N. Ireland, 1972.
20. MURPHY, J. E. "Resource allocation with interlock detection in a multitask system." In *Proc. AFIPS 1968 Fall Joint Computer Conf.*, Vol. 33, Pt. 2, AFIPS Press, Montvale, N.J., 1169–1176.
21. RAPPAPORT, ROBERT L. "Implementing multiprocess primitives in a multiplexed computer system." Masters Thesis, Dept. Electrical Engineering (Project MAC), Massachusetts Institute Technology, Cambridge, Mass., Nov. 1968.
22. RUSSELL, R. D. "A model of deadlock-free resource allocation." PhD Thesis, Dept. Computer Science, Stanford Univ., Stanford, Calif., July 1971.
23. SALTZER, JEROME HOWARD. "Traffic control in a multiplexed computer system." PhD Thesis, Project MAC, Massachusetts Institute Technology, Cambridge, Mass., July 1966.
24. SHOSHANI, A.; AND COFFMAN, E. G., JR. "Detection, prevention, and recovery from deadlock in multiprocess, multiple resource systems." Technical Report 80, Computer Sciences Lab., Dept. Electrical Engineering, Princeton Univ., Princeton, N.J., Oct. 1969.
25. VERNER, YVES. "On process communication and process synchronization." Masters Thesis, Dept. Computer Science, Univ. Toronto, Toronto, Ontario, Canada, Oct. 1971.