IBM Research

LOCKING IN A DECENTRALIZED COMPUTER SYSTEM

Jim Gray

February 8, 1974

RJ 1346

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

Limited Distribution Notice

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

LOCKING IN A DECENTRALIZED COMPUTER SYSTEM

by

Jim Gray

IBM Research Laboratory

San Jose, California

ABSTRACT: Previous work on the deadlock problem in computer systems has focused on algorithms which require knowledge of the state of the entire system. These algorithms do not seem applicable to very large or to distributed (network) computer systems since each allocation decision requires that the entire system state be frozen temproarily. This paper defines the concept of a <u>local resource controller</u> which allocates resources solely on the basis of the state of the requesting process and the requested resource. A mathematical characterization of this model is presented and its relationship to previous work is explored. This model formalizes and generalizes the well known works of Havender and of Bensoussan.

RJ 1346 (#20934)
February 8, 1974
Computer Sciences

TABLE OF CONTENTS

MOTIVATION..1
THE MODEL (an informal presentation).............5
    DEFINITIONS.................................5
    ASSUMPTIONS.................................7
    PREVENTION vs DETECTION vs AVOIDANCE........8
    LOCAL CONTROLLERS (informal)................10
PROPERTIES OF LOCAL CONTROLLERS..................14
    THE ORDER RULE..............................14
        OS/360 EXAMPLE..........................14
        CHOOSING AN ORDERING....................15
        WHEN IS TOTAL ORDER REQUIRED............19
    UNORDERED LOCAL CONTROLLERS.................20
    APPLICATION TO SEMAPHORE SYSTEMS............24
SUMMARY AND EXTENSIONS...........................26
APPENDIX1: CRITIQUE OF ASSUMPTIONS...............30
APPENDIX2: FORMAL DEFINITION AND PRESENTATION OF RESULTS.......................................33
    DEFINITION OF A SYSTEM......................33
    DEFINITION OF A CONTROLLER..................35
    DEFINITON OF DEADLOCK.......................36
    DEFINITION OF LOCAL CONTROLLER..............38
    CONTROLLER INDUCES ORDERING.................40
    ORDER IMPLIES SAFE..........................42
    CHARACTERIZATION OF GENERAL CASE............44
    TOTAL ORDER IFF COMPLETE....................45
    TOTAL ORDER IS MAXIMAL......................47
REFERENCES.......................................59

MOTIVATION

The arrival of Large Scale Integration (LSI) promises to dramatically change the way computers and their software are constructed. The availability of inexpensive LSI solves many problems for which imperfect software solutions have been devised. For example, the problem of providing conventional interactive computing of the JOSS, Dartmoth-BASIC, APL, or even MULTICS variety may be solved by giving _each_ user a one MIP (million instructions per second) processor with a megabyte of one microsecond memory (all on one wafer) and a backing store of a billion bytes of bubble memory. The cost of such hardware is projected to be less than the current cost of a terminal by 1990 (if a high volume market can be found or generated).

A central problem that is not directly solved by LSI is the construction of systems that _require_ sharing of data among a large community of users. This problem is quite difficult even now, in a multi-processing system consisting of one or a few processors with a common storage hierarchy. The problem of hundreds or thousands of processors, each with its own storage hierarchy, sharing data over relatively slow and expensive communications paths is substantially more difficult.

A second problem is that (in the near term) LSI will be cheaper,but not much faster than existing devices. In fact, cheap-low power LSI may be slower than many existing

devices. Thus, if one wants high performance computers (for large scientific calculations, weather predictions, or even for large centralized data bases) and if one wants to benefit from cheap LSI then it seems that a highly parallel architecture utilizing many identical processing elements will have to be adopted.

At this point, the architecture of such distributed systems is a difficult and vague problem. There are several networks of computers in existence and several more are being constructed. Many of the existing networks are amazingly naive. One for example works by having each computer masquerade as a card reader plus card punch to the other computers in the net. Others, (e.g., the ARPA net) are more sophisticated in that the network accepts logical records and automatically does the blocking and routing of messages.

To my knowledge, no network provides a uniformly addressable file space which allows programs running on nodes of the net to transparently access data resident in other nodes of the net. To do this in existing systems, a process must be _explicitly_ constructed in each node and data access across nodes must be done by the user as an explicit send-respond protocol quite different from fetch or store protocols. Future networks will undoubtedly automate this process.

Systems which automate this process will have to choose mechanisms which decide:

How to schedule resource requests.

How to detect or avoid deadlock.

This paper investigates the deadlock problem for such a distributed system. Unlike the deadlock problem for a centralized system in which a single allocation program has complete knowledge of the system state, it is proposed here that the allocation decision must be made locally (i.e., must be independent of how many other processes there are, how many other resources there are, or what the general state of the system is). The motivation for this is that if the decision is made centrally, then the centralized controler will become a bottleneck through which all processes must pass. The power of such a system is limited by the capacity of the centralized controller to schedule resources. Our goal is to find an architecture in which the power of the system rises strictly monotonically with the circuit count of the system or to prove that no such architecture exists.

For example, searching the entire process-resource graph as IMS does (IBM 1973), a procedure which requires time proportional to the cube of the number of processes, or invoking the bankers algorithm (Habermann 1969), a procedure which requires time proportional to a non-linear function of the number of processes and resources, are not allowed since they require that the granting of resources be serialized and because significant computational overhead is required if the requested resource is busy.

The problem is to arrange things so that an anarchy of processes can share resources and yet not deadlock. This work has strong ties to the proposals by Hoare for monitors on data and by Hewitt for actors each of which has very local knowledge of the system.

The next section describes a model of locking in a distributed system in an informal way. The third section describes properties of the model (again in an informal way). The first appendix contains a critique of the model's assumptions and how almost all real systems violate these assumptions. The second appendix gives a formal presentation of the results of the paper.

THE MODEL

Suppose that there are a finite number of _processes_ P1,P2,... and _resources_ R1,R2,.... Processes are presumed to do many things but for the purposes of the model their only actions are to _request_ and _release_ exclusive access to various of the resources. When a process requests a resource, one of two things may occur:

(ACCEPT) The request may be accepted

(DENY) The request may be denied.

If the request is accepted then:

If the resource R is free, P is granted exclusive access to R immediately.

If the resource R is already granted to some process then P is queued (first-come first-served) and _suspended_ (i.e. prohibited from making further requests or releases until this request is granted). When P releases R, the next process in the queue for R is immediately made _active_ and granted access to R.

Only active processes may release resources, but releases are always accepted.

The _states_ of the system described by this model have a very simple representation as depicted in Figure 2.1. The advantage of this representation over the familiar biparitite graph with edges drawn between processes and resources (Holt 1971) is that it explicitly shows the queue of processes waiting for a resource so that scheduling the 'next' process is explicit.

```
STATE1:
     R1 <- P1  , P2
     R2 <- P1
     R3 <- P2
     R4 <-

RESOURCES(STATE1,P1)  =  {R1,R2}   REQUESTS(STATE1,P1)  =  ∅
RESOURCES(STATE1,P2)  =  {R3}      REQUESTS(STATE1,P2)  =  {R1}

              process P1 releases resource R1

STATE2:
     R1 <- P2
     R2 <- P1
     R3 <- P2
     R4 <-

RESOURCES(STATE2,P1)  =  {R2}      REQUESTS(STATE2,P1)  =  ∅
RESOURCES(STATE2,P2)  =  {R1,R3}   REQUESTS(STATE2,P2)  =  ∅

              process P2 requests resource R2

STATE3:
     R1 <- P2
     R2 <- P1  , P2
     R3 <- P2
     R4 <-
RESOURCES(STATE3,P1)  =  {R2}      REQUESTS(STATE3,P1)  =  ∅
RESOURCES(STATE3,P2)  =  {R1,R3}   REQUESTS(STATE3,P2)  =  {R2}

              process P1 requests resource R1

STATE4:
     R1 <- P2  , P1
     R2 <- P1  , P2
     R3 <- P2
     R4 <-
RESOURCES(STATE4,P1)  =  {R2}      REQUESTS(STATE4,P1)  =  {R1}
RESOURCES(STATE4,P2)  =  {R1,R3}   REQUESTS(STATE4,P1)  =  {R2}
```

Figure 2.1 The representation of states.

A state is said to be *deadlocked* if there is a sequence of processes P1,P2,...,PN such that P1=PN and P(i) has requested a resource held by P(i+1) for each i=1,...,N-1. Non-deadlock states are called *safe* states. In Figure 2.1, states STATE1, STATE2, and STATE3 are safe but STATE4 is a deadlock state. The decision to accept or to deny a particular request forms the crux of a system. The mechanism which makes this decision is called the system *controller*. A controller is *safe* if it never accepts a request which would transform a safe state into a deadlock state.

ASSUMPTIONS

This model and most other formal studies of the deadlock problem (more or less) implicitly assume that the processes and controller obey the following rules:

NO PRE-EMPTION

Processes explicitly release resources; neither the controller nor another process can unilaterally pre-empt a resource granted to a process.

HARMONY

Each process, if granted its requests will ultimately release each of its granted resources.

INDEPENCENCE

Processes do not communicate with one another and do not delegate their access to granted resources to one another. All synchronization is done explicitly via the

controller.

SUSPENSION

Once a request has been accepted from a process, the process can neither cancel the request nor make a further request. From the point of view of the controller, the process is suspended until the request is granted.

FIRST-COME FIRST-SERVED

Resources are scheduled in a naive way (typically first-come first-served) subject to access, concurrency, and deadlock constraints.

GENERAL CLASSES OF CONTROLLERS

The literature contains three flavors of controllers: ones that *prevent deadlock*, ones that *detect* deadlock, and ones that heuristically *avoid* deadlock.

DEADLOCK PREVENTION

The simplest scheme to prevent deadlock is to outlaw concurrency. In such a scheme only one process is allocated any resources at any instant; all requests by other processes are queued for service. Requests can be accepted in any order so long as the resources exist. However this controller tends to make very poor use of resources. A second scheme is to have each process predeclare its (time varying) maximum future demand for resources (Habermann 1969, Hebalkar 1970). This declaration can only be changed

while the process is quiescent (has no resources granted). Whenever a request is made this information is used to decide whether a deadlock state might arise if the request were granted immediately. If deadlock could arise the request is scheduled to be granted at some future time. The bankers algorithm is an example of such a scheme. A third technique is to establish some protocol about resource requests. If all processes follow the protocol, deadlock will be avoided. The controller enforces the protocol. Linearly ordering the resources and limiting each process to requesting only those resources 'bigger' than its current holdings is an example of such a scheme (Havender 1968, Bensoussan 1968). A major point of this paper is a theortical argument that for a certain interesting class of systems this is the only such protocol.

DEADLOCK DETECTION

A second scheme is for the controller to grant all resource requests and releases which do not immediately result in a deadlock state. When a request is made which would result in a deadlock state, the request is rejected and the controller invokes some recovery mechanism to correct the state of the system. Whereas prevention schemes make worst-of-all-possible-worlds assumptions about the behavior of processes, deadlock detection schemes make rather optimistic assumptions about process behavior. That is, in a prevention scheme a request is granted only if it cannot possibly result in deadlock while in a detection

scheme a request is granted unless it immediately results in deadlock. In the event of a deadlock, a recovery procedure is initiated and some resources may have to be pre-empted. Detection schemes tend to have somewhat less computational overhead since they evaluate only the safeness of the next state rather than the safeness of all possible succeeding states. On the other hand, detection schemes allow deadlock and hence requirea pre-emption and a recovery mechanism.

DEADLOCK AVOIDANCE

Empirical ovservations by Needham and Hartley (Needham, Hartley 1969) and by others suggest that deadlock prevention schemes grossly undercommit resources and that detection schemes may give resources away so freely that deadlock or near-deadlock thrashing may occur. That is to say most requests for heavily used resources are queued rather than being granted immediately and hence incur the expense of a process switch. They recommend that the system be flushed out and recovered whenever it becomes congested and that the controller be made conservative enough so that this is an infrequent occurrence. Similar comments can be made in connection to real systems. The real bankers algorithm is to loan five times your assets. Airlines routinely overbook flights by ten percent. In the absence of laws to protect patrons, this figure might be substantially higher.

We are interested in characterizing a particularly simple kind of controller. We are interested in safe

controllers which have the following properties:

LOCALITY

The decision to accept or deny a request by process P for resource R is independent of the number or states of the other processes or resources in the system. It depends only on the state of the requesting process and on the resource R.

DETERMINISM

The acceptability of a sequence of resource requests by a process starting in a state in which it has no resources is independent of the state of other processes in the system and independent of the process name. That is a particular request sequence is either always acceptable for any process or it is always denied.

NO PRE-DECLARATION

There is no mechanism whereby a process can declare its future requests.

A controller having these properties is called a *local controller*. The accept predicate of a local controller is simply a function of the resources held by the requesting process and of the requested resource. More formally, in state STATE, a request by process P for resource R is accepted if and only if:

ACCEPT(RESOURCES(STATE,P),R) = TRUE.

The accept predicate for a particular local controller may be represented as a graph on subsets of the set of

resources. An example of this representation is shown in Figure 2.2.

RESOURCES: R1,R2

ACCEPT(Ø,R1) = TRUE   ACCEPT(Ø,R2) = TRUE

ACCEPT({R1},R2) = TRUE ACCEPT({R2},R1) = FALSE

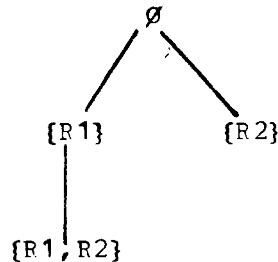


Figure 2.2 The representation of an ACCEPT predicate as a graph.

Our interest in this class of controllers stems from proposals for data base systems which envolve millions of resources (records) and thousands of processes (transactions). A further complication of these systems is that the data may reside in several different computers. Neither the one process-at-a-time nor the one resource-at-a-time controllers are adequate for these systems. Detecting cycles in such a large (and decentralized) graph would be a slow and costly process indeed which would have to be performed for every request for a busy resource. Algorithms for finding cycles have a

cost proportional to the number of edges in the graph. Further the cycle-detecting controller is not deterministic in the sense described above Determinism is desired because transactions are written to be run by naive users in many different environments. In such a context it may be preferable to have programs always fail rather than to have programs almost-always work.

Concerning the assumption of no-predeclaration, it is true that some transactions can pre-declare the resources they need. However it is common to find cases in which the identity of the next record to be locked is determined by the contents of the just-locked record. This data-dependent locking is even common in conventional operating systems where for example, the particular page table to be locked depends on the contents of the segment table (Bensoussan 1968 or figure 3.4). In this context, pre-declaration is not feasible.

PROPERTIES OF LOCAL CONTROLERS

A particularly simple kind of local controller may be constructed as follows. If all the resources of the system are ranked by some partial order '<' then a safe local controller can be built from the accept predicate described by:

THE ORDER RULE:

> Accept a request by process P for resource R only if R has rank higher than any resource currently held by P. (i.e. R>R' for all R' in RESOURCES(STATE,P)).

Both Havender (Havender 1968) and Bensoussan (Bensoussan 1968) have documented their use of this scheme in the nucleus of an operating system. To give a concrete example of the scheme, consider Figure 3.1 which describes the partial order used by OS/360. All resource requests must be made as follows: A process first requests all the data sets it will need, then it requests a partition of memory, then it requests the devices it needs, and then it requests the secondary storage it needs. The rational for this ordering is nicely explained by Goldstein (Goldstein 1973). If a process ever needs a new device it must first release all its secondary storage and then resubmit its device request. (Bensoussan calls this process <u>decomputing</u> the secondary storage). The process may then re-request the secondary storage it needs. Similarly, if a process needs another data set, it must decompute (release) all of its secondary

storage, devices and main store and then resubmit its requests for data sets. In fact the process must release all it's data sets because otherwise deadlock could arise (e.g. process P1 holds data set DS1 and wants DS2, process P2 holds DS2 and wants DS1). After the process gets the desired data set it may re-request the resources it held formerly.

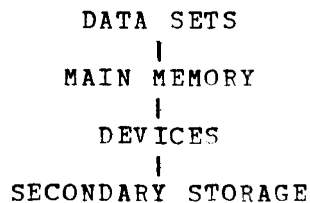


Figure 3.1 The OS/360 resource hierarchy.

Any partial order combined with the order rule will produce a safe controller. The principle differences among orderings will be:

(statistical) One ordering may require fewer decomputes in order to expand resource sets.

and (concurrency) One ordering may allow more possible resource sets than another.

The OS/360 hierarchy gives an example of statistical advantage. If the hierarchy were inverted, then the relatively common operation of asking for more secondary storage would become more costly (require more decomputes) while the relatively infrequent operation of asking for a new data set would become substantially cheaper. The advent

of TSO and other interactive systems running under OS/360 but which make dynamic requests for data sets and devices has complicated this decision.

The second way in which one ordering can be superior to another is that one may allow more resources to be granted at a time. Consider the orderings described in Figure 3.2. In 3.2.a the ordering is the empty ordering which allows the holding of only a single resource at a time. The ordering of 3.2.d is a total ordering which allows any subset of the set of resources to be requested. The other orderings are intermediary between these two extremes.

A safe local controller C2 is said to extend the local controller C1 if the accept predicate of C2 implies the accept predicate of C1 (i.e. if ACCEPT.C1(STATE,RESOURCE) = TRUE then ACCEPT.C2(STATE,RESOURCE) = TRUE). The extension is proper if C2 is not identical to C1. A safe local controller is said to be maximal if it has no proper extensions. A maximal local controller allows all the states and transitions allowed by the controllers that it extends.

I had earlier believed that a hierarchy of locks was more general than a total order because a hierarchy allows sets of locks to be requested in a natural way: requesting a single lock requests requests all locks below it in the lock tree (Gray 1969). Further, a linear order is simply a degenerate hierarchy. However, a hierarchical lock system is not maximal in the sense described above. I was suprised to find that a pre-order traversal of the lock tree (visit

root, visit sons in left to right order (see Knuth 1968) produces a total order which when combined with the order rule extends the hierarchical lock structure to a maximal ordered local controller. That is the hierarchy is less general than the total order. This realization caused me to want to characterize the properties of local controllers in a formal and complete way.

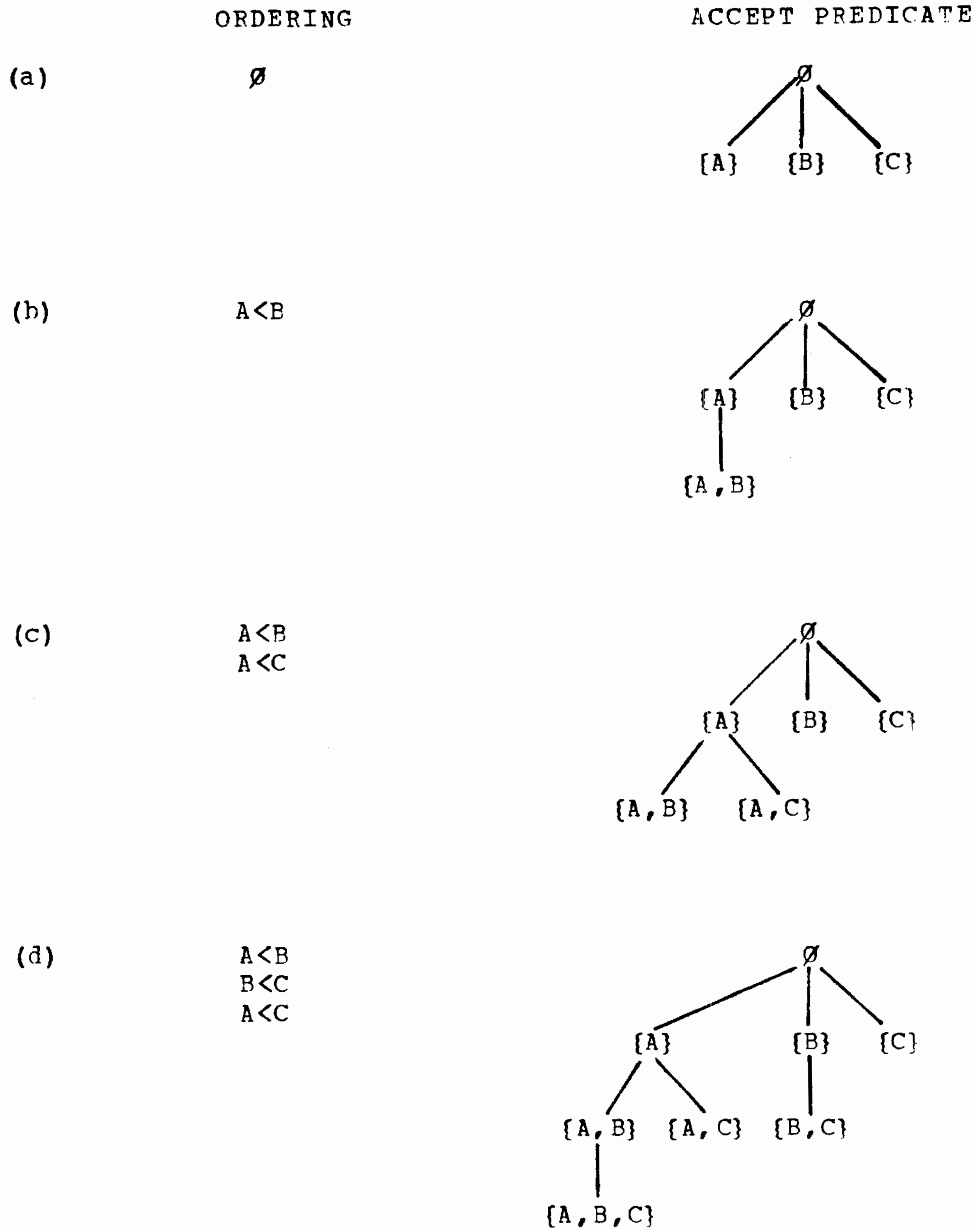


Figure 3.2 The possible orderings of three resources and their accept predicates.

Applying the criterion of maximality to the OS/360 hierarchy (figure 3.1), we see that if all the data sets were ordered, then it would not be necessary to release all data sets before requesting a new one. Rather it would only be necessary to decompute those data sets 'larger' then the requested one. In the example cited above, if DS2 were greater than DS1 then process P1 could request it directly while process P2 holding DS2 would have to release DS2 before requesting DS1.

Any partial order can be extended to a total order. A controller based on the total order has the property that it allows all the states allowed by the controller based on the partial order and that it may allow more states and require fewer decomputes (i.e. it is maximal).

To summarize then, if the order rule is used, it should be used with a total order because this produces a maximal controller.

This suggests a natural question: "Under what circumstances does the model require the use of the order rule with a total order?" The answer to this question turns out to be suprisingly simple. A system will be called complete if any process starting in a quiescent state can request any subset of the resources in some order. Then:

A local controller is safe and complete

if and only if

the controller is based on the order rule using a total order.

Again consider a few examples. In Figure 3.2, only

3.2.d is complete and it embodies a controller which is based on a total order. Figure 3.3.a displays another maximal safe local controller on three resources (all local controllers on three resources are homomorphs of 3.2.d, 3.3.a, or of 3.3.b). The controller with accept predicate depicted in Figure 3.3.a is not complete since it does not allow the set {B,C} to be requested directly. To request the set {B,C} a process must first request the set {A,B,C} and then release the resource A.

UNORDERED LOCAL CONTROLERS

So far, controlers based on the order rule have been characterized. Now consider local controlers not based on the order rule. Figure 3.3 depicts the three general forms in which an 'unordered' local controller can be constructed.

The accept predicate of Figure 3.3.a is the simplest example of a local controller not based on a total order. Note that resource B can be requested before or after resource C is requested. If the order rule were used this would require that $B<C$ and $C<B$. This means that the relation $<$ could not be antisymetric and hence cannot be called an order (partial or total). Thus the controller described by Figure 3.3.a does not derive from the order rule.

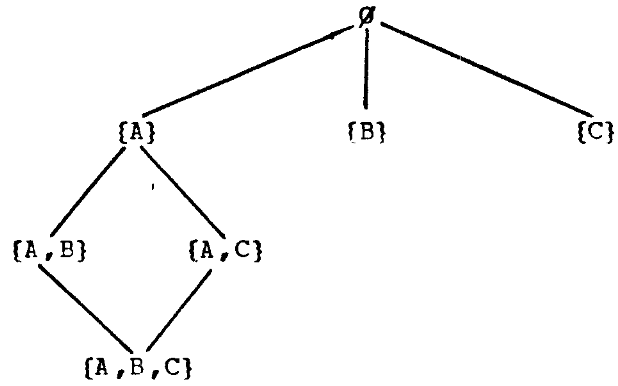
In figure 3.3.a a request for resource A in this system is essentially a declaration that resources B and C may be requested in any order. It is an implicit form of

pre-declaration. By the assumption of exclusive access, only one process may hold resource A at any instant. Hence the system is serialized on requests for the resource set {B,C} and the exact order in which resources B or C are requested is irrelevant. However, other processes may request resources B and C individually while a third process holds resource A. Thus the system is not completely serialized.

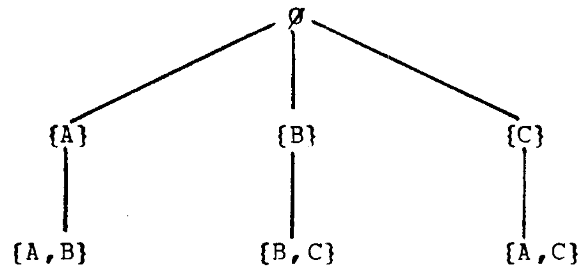
It is possible to use this serialization mechanism to an extreme. For example, if the system requires that all resource requests must be preceeded by a request for the resource A then the system becomes a single-process-at-a-time system. We have investigated the topic of using this trick of introducing new resources into the accept predicate of an unsafe controller so that the unsafe sub-graphs of the accept graph are serialized. The problem is to serialize the minimal sub-graph. The answer to this problem is difficult, combinatorial, and probably uninteresting. It is not included in the formal presentation.

The system described in figure 3.3.b is safe if there are two or fewer processes. However if the system contains at least three processes, then it is unsafe because the shortest cycle has length three: (ABC). The generalization of this construction to systems which are safe for at most 'n' processes should be obvious.

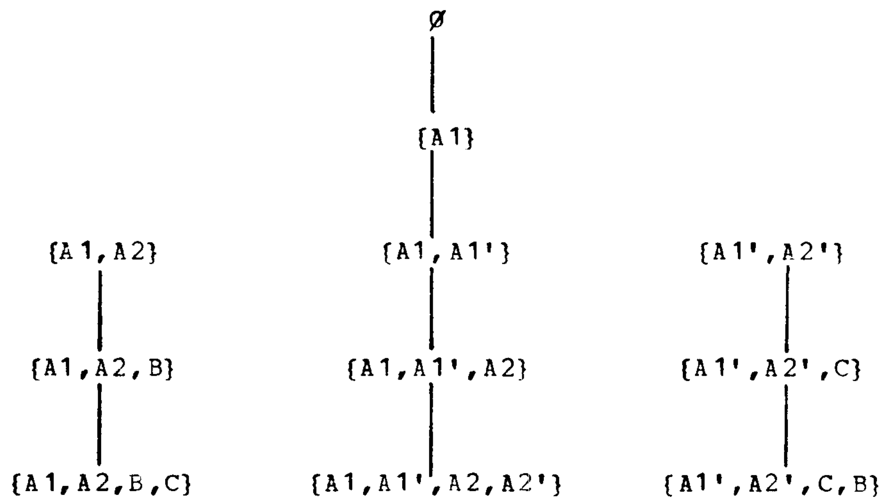
Figure 3.3.c is intended to convey the impression that very complex constraints can be represented implicitly by the resource sets held by a process. In the system described by Figure 3.3.c, it is impossible for the resource sets {A1,A2} and {A1',A2'} to co-exist in a state. That is, starting in a state in which no process has any resources, the controller does not allow transition to a state in which one process holds resource set {A1,A2} while a second process holds resource set {A1',A2'}. The argument to prove this is fairly simple: examination of the graph shows that a path to the resource set {A1,A2} must be indirect. First the set {A1,A1',A2} must be requested and then A1' is released. So one may assert that, by the assumption of exclusive access, the resource A1' must be free when the set {A1,A2} is first obtained. Similarly, to obtain the resource set {A1',A2'}, the resource set {A1,A1',A2,A2'} must first be requested and then the resources A1 and A2 can be released. This indirect path requires that the resources A1 and A2 be free when the resource set {A1',A2'} is obtained. So if one of the two sets exists, the second set cannot be obtained. This mutual exclusion is somewhat suprising since the sets are disjoint. However the effect is that the system is serialized on these two resource sets in much the same way that the single resource A in Figure 3.3.a serializes that system. Hence it is possible for the resources B and C to be requested in any order once one of these exclusion sets has been requested.



(a)



(b)



(c)

Figure 3.3. Unordered safe local controllers.

It is perhaps suprising that the results on local controllers are applicable to semaphores used for synchronization, as opposed to communication (Habermann 1969), or queues as used in OS/360 (IBM 1970). In particular, binary semaphores used for critical sections and mutual exclusion or exclusive access to queues (shared access is discussed briefly in the next section) fit the model of a local controller quite well. The access to the resources is exclusive and is scheduled first-come first-served. There is no global search for cycles in the resource-process graph and in fact requests are always accepted. Hence all the deadlock prevention of such systems is contained in the protocol implicit in the way resources are requested by the processes. This protocol can be extracted at 'compile time' by examining the control flow of the programs that request the resources. Associated with each line of code are the sets of locks which could be held at that point in the computation. The exact manner of computing these sets will be described in a separate paper; however, Figure 3.4 should be suggestive of how the algorithm works. It displays a pseudo-PL/1 program with multiple processes represented as procedures and with the resource sets possible at each step displayed next to each line which does a LOCK or UNLOCK. From this data one may construct the accept predicate required by the processes and use lemma 8 to test its safeness. The system of Figure 3.4 produces the accept predicate described in figure 3.3.a.

```
SYSTEM: PARALLEL PROCEDURE OPTIONS(MAIN);
   DECLARE (PAGE_TABLE, SEGMENT_TABLE, PERHAPS_BOTH) LOCK;

   PAGE: PROCEDURE();
      LOCK(PAGE_TABLE);                                {B}
      <update PAGE_TABLE>;
      UNLOCK(PAGE_TABLE);                              Ø
      END;

   SEGMENT: PROCEDURE;
      LOCK(SEGMENT_TABLE);                             {C}
      <update SEGMENT_TABLE>;
      UNLOCK(SEGMENT_TABLE);                           Ø
      END;

   PAGE_THEN_SEGMENT_PERHAPS: PROCEDURE();
      LOCK(PERHAPS_BOTH);                              {A}
      LOCK(PAGE_TABLE);                               {A,B}
      <update PAGE_TABLE>;
      IF <necessary based on PAGE_TABLE> THEN
         BEGIN;
         LOCK(SEGMENT_TABLE);                        {A,B,C}
         <read and update SEGMENT_TABLE>;
         <update PAGE_TABLE>;
         UNLOCK(SEGMENT_TABLE);                       {A,B}
         END;
      UNLOCK(PAGE_TABLE);                              {A}
      UNLOCK(PERHAPS_BOTH);                            Ø
      END;

   SEGMENT_THEN_PAGE_PERHAPS: PROCEDURE;
      LOCK(PERHAPS_BOTH);                              {A}
      LOCK(SEGMENT_TABLE);                            {A,C}
      <read and update SEGMENT_TABLE>;
      IF <necessary based on SEGMENT_TABLE> THEN
         BEGIN;
         LOCK(PAGE_TABLE);                           {A,B,C}
         <read and update PAGE_TABLE>;
         <read and update SEGMENT_TABLE>;
         UNLOCK(PAGE_TABLE);                          {A,C}
         END;
      UNLOCK(SEGMENT_TABLE);                           {A}
      UNLOCK(PERHAPS_BOTH);                            Ø
      END;

   END;
```

Figure 3.4. An example of a system which can be analyzed for safeness. The sets to the right of statements requesting or releasing locks denote what locks are held after the statement is executed. The code is:

A : PERHAPS_BOTH   B : PAGE_TABLE   C : SEGMENT_TABLE

SUMMARY

The central conclusion of this study is that:

IF one adopts the assumptions of a local controller (e.g. no pre-emption, harmony, locality,...)

THEN only very stylized locking strategies will prevent deadlock.

In fact the well known rule: "order the resources and always request them in order" *must* be used if any subset of the resources can be requested directly (i.e. if the system is complete).

One may have either of two reactions to this: either joy that life can be so simple or dispair that life must be so simple. Those who have the second response must then reject the idea of a local controller and decide upon which of the assumptions can be relaxed.

EXTENSIONS

There are several easy extensions of the model. The formal definition of these extensions and a formal exploration of their properties is not presented here in the interest of simplicity. However a brief mention of them is appropriate.

NON-EXCLUSIVE ACCESS:

Requests could be dichotomized into 'shared access' or 'exclusive access' requests. Exclusive access is the kind

described above, but shared access grants access to a resource whenever all the processes ahead of the requesting process have shared access. This model obeys all the results presented in Appendix 2 up to proposition 11. This model is directly applicable to systems where there are readers and writers. Readers request shared access to an object and may have concurrent access to it. Writers request exclusive access to the object and wait until all readers preceeding them have released the object before gaining access to it.

NON-TRIVIAL SCHEDULING:

It is quite a simple matter to generalize our results to a system in which requests are granted in a manner other than first-come first-served. The necessary criterion for such a scheduler is that it grant each request eventually. This allows for prioritizing processes in any way so long as the priority of each process grows with time and eventually exceeds the initial priority of all other processes. On the other hand it is interesting to note that the assumption that processes are not allowed to make further resource requests until their current request is granted is a critical assumption. If this assumption is relaxed, then deadlock occurs in the model we have presented. To see this consider the system of figure 2.2. Process P1 requests resource A, process P2 requests resource A and then B. Then P1 requests B. Now P1 is waiting for P2 and

P2 is waiting for P1.

REQUESTS FOR SETS OF RESOURCES:

Rather than requesting one resource at a time, processes might request or release a set of resources with each call to the controller. The controller must schedule these requests. If it is distributed, the controller is likely to be based on some form of local controller which 'sorts' the set and then requests one resource at a time and so degenerates to a local controller.

OPTIMIZATIONS

If one is willing to sacrifice determinacy, then several optimizations are possible. The first is to always accept requests for free resources. Assuming that resources are usually available this will save decomputing. However, algorithms which depend on this will work only sometimes. A second strategy is to have a time varying ordering of the resources. In this scheme, the linear ordering rule is used but the 'rank' of a resource varies with time. Free resources have 'infinite' rank. When a resource is granted, it is given rank greater than the rank of any resource held by the requestor. The resource keeps this rank until no requests for it are pending, at which time it gets infinite rank again. The correct sequence for requesting resources in such a system is time varying and non-deterministic; however, its other properties are similar to the

properties of local controllers.

APPENDIX 1

ASSUMPTIONS IMPLICIT IN THE MODEL OF A LOCAL CONTROLLER

The model adopted for a local controller is quite simple. Implicit in that model are several assumptions which were described briefly in the section introducing the model. These assumptions are somewhat outrageous since almost all 'real' systems violate almost all of them. The following paragraphs expand on this statement.

HARMONY

No-one intends to author an in-harmonious program. In practice, design errors are quite common. Further, programs run on real hardware and hence are subject to hardware errors. This means that programs cannot be assumed to be harmonious.

PRE-EMPTION

Clearly deadlock detection schemes and deadlock avoidance schemes allow deadlock and so require recovery mechanisms. As argued above, even deadlock prevention schemes can not assume complete harmony so they too must have a recovery mechanism. A recovery mechanism necessarily implies aborting or 'backing-up' certain processes and pre-empting their resources.

INDEPENDENCE

One of the main reasons for multiprogramming is to allow computations to be broken into control structures that are conceptually simple and which reflect the structure of the computation. So for example, each transaction of

an airlines-reservation system is given its own process rather than having a single server process which handles all transactions. Interprocess communication plays a crucial role in any such system. Processes often delegate their access to various other processes that they spawn or invoke. These activities violate the assumption that processes are independent and complicate the deadlock problem enormously.

NAIVE SCHEDULING

As mentioned above, a sure way to prevent deadlock is to grant the requests of only one process at a time and to keep all other processes suspended until the active process releases all its resources. The major drawback to this scheme is the lack of concurrency. Any more sophisticated controller than this is an attempt to get greater concurrency. The philosophy of granting a free resource whenever it is requested (subject to the constraint that the request will not result in deadlock), is obviously sound. However, if several processes are queued waiting for a resource, the use of a first-come first-served discipline may not be desirable. For example the FCFS algorithm for processor allocation (round robin) can be significantly inferior to other algorithms in terms of throughput, overhead, and response. In general, if the access constraints on resources are complex, then an elaborate scheduler may be needed to regulate and schedule this access. The behavior of such schedulers is

difficult to abstract or to analyze.

SUSPENSION

Suspending a process or dis-allowing multiple requests is a common restriction. Although many real systems contain this constraint, it is often there more for the convenience of the implementor than for the protection of the system or for the correctness of the deadlock avoidance mechanism. The strategy of overlapping input and output operations with computation is a very common example of the general case in which a request for a resource (I/O facilities in this case) should not cause suspension of the requesting process. After a request is made the process has the option to continue or to wait for the requests to be granted. As explained in the extensions section above, our model cannot relax this assumption.

REAL TIME

Very few (none to my knowledge) of the models have any concept of real time. There is no attention paid to the problem of evaluating the response time to requests or the throughput of the system. This appears to be a fruitful area for study.

APPENDIX 2

A FORMAL PRESENTATION OF LOCAL CONTROLLERS AND THEIR PROPERTIES

Definition 0: Let SET be a set. Then SET* denotes the set of all sequences of elements of SET including, 'e', the empty sequence. Let RELATION be any binary relation on the set SET (i.e. RELATION is a subset of the cartesian product of SET x SET). A relation RELATION' is said to *extend* RELATION if RELATION' contains RELATION. RELATION is *transitive* if for every (A,B) and (B,C) in RELATION, (A,C) is in RELATION. RELATION is *antisymmetric* if for every (A,B) in RELATION, (B,A) is not in RELATION. RELATION is said to satisfy *trichotomy* if for any pair of elements A, B in SET either: A=B or (A,B) is in RELATION or (B,A) is in RELATION. RELATION is a *partial order* if it is transitive and antisymmetric. RELATION is a *total order* if it is a partial order and it satisfies trichotomy. RELATION+ denotes the *transitive closure* of RELATION: the smallest transitive relation extending RELATION. ■

Definition 1: Let RESOURCES be a finite set of *resources* and let PROCESSES be a finite set of *processes*. A *system state* is any function:

S: RESOURCES -> PROCESSES*

where S(R) = P(1),...,P(n) is interpreted to mean that process P(1) holds resource R and processes P(2),...,P(n)

have requested resource R. Let STATES be the set of all states. There is a distinguished (system) state denoted So and called the <u>quiescent state</u>:

So: RESOURCES -> {e}

in which no process has any resources or requests.

Fix state S and process P and define:

<u>RESOURCES</u>(P,S) = { R in RESOURCES | for some z in PROCESSES*
S(R) = P z }

<u>REQUESTS</u>(P,S) = { R in RESOURCES | for some y, z in PROCESSES*
y ≠ e and S(R) = y P z }

The process P is said to be <u>active</u> in state S if REQUESTS(S,P) = Ø otherwise it is said to be <u>suspended</u>. Define the <u>graph</u> of state S, GRAPH(S), as a binary relation on PROCESSES U RESOURCES by:

for each process P and resource R,

(R,P) is in GRAPH(S) if R is in RESOURCES(S,P)

(P,R) is in GRAPH(S) if R is in REQUESTS(S,P).

Further, for each R in RESOURCES define:

<u>REQUEST</u>(S,P,R) to be the state S' such that:

S'(R') = S(R') for each resource R' ≠ R,

S'(R) = S(R) P.

and if S(R) = P z for some sequence z of processes then define:

<u>RELEASE</u>(S,P,R) to be the state S' where:

S'(R') = S(R') for each resource R' ≠ R,

S'(R) = z ■

Definition 2: A binary relation CONTROL on STATES is said to be a (resource) controller if it satisfies the constraint that:

(i) for each state S and for each process P active in S either RESOURCES(S,P) = Ø or for some resource R in RESOURCES(S,P) it is the case that (S,RELEASE(S,P,R)) is in CONTROL.

(ii) If (S,S') is in CONTROL then for some process P active in S and for some resource R, either:

(ii.a) S' = REQUEST(S,P,R), R not in RESOURCES(S,P)

or (ii.b) S' = RELEASE(S,P,R) where R is in RESOURCES(S,P)

For a given controller, CONTROL, and state S define:

ORBIT(CONTROL,S) = { S' | (S,S') in CONTROL+ }.

A state S is said to be safe (with respect to CONTROL) if So is in ORBIT(CONTROL,S), otherwise the state is said to be deadlocked. The controller, CONTROL, is said to be safe if all states in the ORBIT(CONTROL,So) are safe.

The safe controller CONTROL is maximal if there does not exist a safe controller CONTROL' which extends CONTROL and properly extends CONTROL on ORBIT(CONTROL',So). ■

Remark: System states, operators on states, and the concept of a controller which always accepts releases but regulates requests have been defined. Note that the decision of the controller to accept or deny a request may depend on the entire system state. It is only required that it depends

only on the system state. The next step is to establish the classical result that deadlock is equivalent to a cycle in the state graph.

Lemma 3: Let CONTROL be a controller, then S is a deadlock state with respect to CONTROL if and only if GRAPH(S) has a cycle (i.e. there is a sequence P(0), R(1), P(1),..., R(n), P(n) such that (R(i),P(i)) and (P(i-1),R(i)) are in GRAPH(S) for i = 1,...,n and such that P(0) = P(n)).

Proof: (deadlock=>cycle) Suppose S is a deadlock state. Consider any active process P such that:

$$RESOURCES(S,P) = RS \neq \emptyset.$$

Definition 2.i implies that for some resource R in RS, the pair (S,RELEASE(S,P,R)) is in CONTROL. Since P continues to be active in the new state and since RESOURCES is finite it follows that a finite number of releases produce a new state, S', such that for any active process P RESOURCES(S',P) $\neq \emptyset$. On the other hand, ORBIT(CONTROL,S') is a subset of ORBIT(CONTROL,S) because S' is in ORBIT(CONTROL,S). So if So is not in ORBIT(CONTROL,S) then So is not in ORBIT(CONTROL,S'). Thus S' is a deadlock state if S is. This proves the general result that any (deadlock) state can be transformed into a new (deadlocked) state which has the property that:

(i) for any process P either P is suspended,

or RESOURCES(S,P) = $\emptyset$.

Since S'$\neq$So there is a suspended process P(0) and a

resource R(1) such that R(1) is in REQUESTS(S',P(0)). Since P(0) is suspended there is a process P(1) such that R(1) is in RESOURCES(S',P(1)). Since P(1) has some resources assumption (i) implies that P(1) is suspended. This construction can be continued to produce an infinite sequence: P(0), R(1), P(1),.... which satisfies the property that (P(i-1),R(i)) and (R(i),P(i)) are in GRAPH(S') for each i = 1,.... Since there are only a finite number of processes it must be the case that for two distinct integers i and j: P(i) = P(j) and hence GRAPH(S') has the cycle P(i), R(i), P(i+1),..., P(j). Now observe that GRAPH(S') is a subset of GRAPH(S) and so GRAPH(S) also contains a cycle. This shows that the graph of a deadlock state must contain a cycle.

(cycle=>deadlock) To prove the converse, induct on 'n', the number of processes in the system. For n=0 the result is vacuous since So is the only state. Suppose that the result holds for any system of at most n-1 processes and consider a system of n processes. Let S be a state containing a cycle. If there is a process P not in the cycle then consider the state with P removed. The new state has a cycle and so by hypothesis is a deadlock state. Hence S is a deadlock state. Suppose that each process is in a cycle. Then for each process P there is a resource R such that (P,R) is in GRAPH(S). Thus each process has a non-null requests set and is suspended. Inspection of definition 2.i,ii shows that if no process is active in S then there is no state S' such that (S,S') is in CONTROL. Hence So is not in the

ORBIT(CONTROL,S). This shows S is a deadlock state if it contains a cycle. ∎

<u>Definition 4</u>: Let RESOURCE_SET be the set of all sets of resources. The controller CONTROL is said to be a <u>local controller</u> if there is a function:

ACCEPT: RESOURCE_SET X RESOURCES -> { TRUE , FALSE }

such that for any resource set RS and any resource R it is the case that:

ACCEPT(RS,R) = TRUE

if and only if

for any state S in ORBIT(CONTROL,So) and any process P active in S:

if RESOURCES(S,P)=RS then (S,REQUEST(S,P,R)) is in CONTROL.

Extend ACCEPT to be a predicate on sequences of resource requests as follows: for any resource set RS and any sequence R(1),...,R(n) of resources;

ACCEPT(RS,R(1)...R(n))=TRUE if and only if

ACCEPT(RS U {R(i)|i=1,...,j-1},R(j))=TRUE for j=1,...,n.

In the latter case, the sequence R(1),...,R(n) is said to be a valid <u>request sequence</u> with respect to the local controller CONTROL.

If CONTROL is a local controller we say it <u>induces the order</u> < on RESOURCES, where < is the binary relation on RESOURCES defined by:

R(1) < R(2) if and only if for some resource set RS containing R(1), ACCEPT(RS,R(2)) = TRUE.

Conversely, we say that any binary relation, <, on RESOURCES *induces the local controller* CONTROL defined by:

for any state S, and any process P active in S, and any resource R,

(i) (S,RELEASE(S,P,R)) is in CONTROL if R is in RESOURCES(S,P)

(ii) (S,REQUEST(S,P,R)) is in CONTROL if R(1) < R for each R(1) in RESOURCES(S,P).

(i.e. ACCEPT(RS,R) = TRUE if and only if R' < R for each R' in RS).

The set of all resource sets which may be requested from CONTROL is defined by:

REQUEST_SETS(CONTROL) =

{ RESOURCES(S,P) | S in ORBIT(CONTROL,So) and P any process}

The local controller CONTROL is *complete* if

(a) for every set RS in REQUEST_SETS(CONTROL) the elements of RS may be ordered into a sequence R(1),...,R(N) which is a request sequence for CONTROL.

and (b) REQUEST_SETS(CONTROL) is the set of all subsets of RESOURCES. ■

Remarks: Notice that a local controller always allows releases of resources and that it has a very simple algorithm for deciding the acceptability of a resource request: all processes are anynomous, the decision is based solely on the current holdings of the requestor and on the

requested resource. The next few results develop the properties of local controllers based on the order rule (see page 14).

Lemma_5: Let CONTROL be a local controller which induces the ordering < on RESOURCES. Let S be any state in the ORBIT(CONTROL,So). Then for any process P and resources R(1) and R(2), if R(1) is in RESOURCES(S,P) and R(2) is in REQUESTS(S,P) then R(1) < R(2).

Proof: First, observe that So vacuously satisfies the lemma. The proof will show that for any (S,S') in CONTROL, if S satisfies the lemma then S' does. Invoking induction will prove the lemma is valid.

Suppose that S satisfies the lemma and that (S,S') is in CONTROL. By definition 2.ii there is an active process P and a resource R such that either:

(i) S' = REQUEST(S,P,R),

or (ii) S' = RELEASE(S,P,R) and R is in RESOURCES(S,P).

In case (i) the resources and requests of any process except P are the same in both S and S'. Further, since CONTROL is assumed to be a local controller, (i) implies that ACCEPT(RESOURCES(S,P),R) = TRUE. Since < is induced by CONTROL, this in turn implies that:

(iii) R(2) < R for each R(2) in RESOURCES(S,P).

There are two subcases to consider: If REQUESTS(S',P) = Ø then P satisfies the lemma vacuously in S'; otherwise, REQUESTS(S',P) ={R} and by (iii) the lemma is satisfied. So

in case (i) the lemma is satisfied.

Now consider case (ii). Examine state S. Since R is in RESOURCES(S,P) either:

(iv) S(R) = P

or (v) for some non-null sequence of processes

P(1),...,P(n); S(R) = P,P(1),...,P(n).

In any case P continues to be active in S' (since it has been granted resource R) and so P vacuously satisfies the lemma in S'. In case (iv) the resources and requests of other processes are identical in S and S' and so the lemma continues to hold for them. In case (v) this is not true for the distinguished process P(1) which had been suspended in S but is active in S'. But since P(1) is active in S' it vacuously satisfies the lemma also. Hence S' satisfies the lemma in any case.

Invoking induction establishes the lemma. ∎

Remark: Next follows the classical result that a partial order combined with the order rule yields a safe local conroler.

Proposition 6: Let CONTROL be a local controller which induces the relation < on RESOURCES. If the transitive closure of <, <+, is a partial order then CONTROL is safe.

Proof: Suppose for the sake of contradiction that <+ is a partial order but that CONTROL is not safe. Then there is a state in ORBIT(CONTROL,So) which is a deadlock state. Let S be such a state. By lemma 3, GRAPH(S) must contain a cycle R(0), P(0), R(1), P(1),..., R(n), P(n) such that R(0) = R(n) and the pairs (P(i),R(i+1)) and (R(i),P(i)) are in GRAPH(S) for i = 0,...,n-1. By the definition of GRAPH(S):

R(i) is in RESOURCES(S,P(i)) for i = 0,...,n

and R(i) is in REQUESTS(S,P(i-1)) for i = 1,...,n.

By lemma 5 this implies:

R(i) < R(i+1) for i = 0,...,n-1

By the transitivity of <+ we conclude that:

R(0) <+ R(n) = R(0).

The latter property contradicts the antisymmetry property of a partial order. Hence, if S is a deadlock state then <+ cannot be a partial order. This proves that if <+ is a partial order then CONTROL is safe. ∎

Remark: The next result shows that controllers based on partial orders can be extended to controllers based on total orders.

Proposition 7: Let CONTROL be a local controller which induces the relation < on RESOURCES. If <+ is a partial order and if <' is any extension of <+ to a total order then the local controller CONTROL' induced by <' has the properties:

(i) CONTROL' is safe,

and (ii) CONTROL is a subset of CONTROL',

and (iii) ORBIT(CONTROL,So) is a subset of ORBIT(CONTROL',So).

Proof: Let <' and CONTROL' be as specified above. Then by proposition 6, CONTROL' is safe since a total order is a partial order. This establishes (i).

To prove (ii) consider any (S,S1) in CONTROL. By definition 2.i,ii for some active process P and some resource R either:

(iv) S1 = RELEASE(S,P,R) and R is in RESOURCES(S,P),

or (v) S1 = REQUEST(S,P,R).

In case (iv), (S,S1) is in CONTROL' by virtue of definition 4.i. Case (v) implies ACCEPT(RESOURCES(S,P),R) = TRUE with respect to CONTROL and so by the definition of < for each R(1) in RESOURCES(S,P); R(1) < R. Since <' extends < it follows that for each R(1) in RESOURCES(S,P); R(1) <' R. Thus by the definition of induced controller, ACCEPT'(RESOURCES(S,P),R) = TRUE for CONTROL' and so CONTROL' contains (S,REQUEST(S,P,R)). This establishes (ii).

To prove (iii) note that (ii) implies that CONTROL+ is a subset of (CONTROL')+ and so ORBIT(CONTROL,So) is a subset of ORBIT(CONTROL',So).

Lemma 8: Let CONTROL be a local controller which induces the relation > on RESOURCES. Let R(1),...,R(N) and R'(1),...,R'(M) be two request sequences such that for some integers I less than N and J less than M; R(I) = R'(M) and R(N) = R'(J). Then the sets {R(1),...,R(I)} and {R'(1),...,R'(J)} can not be disjoint if the system is safe and contains at least two processes.

Proof: Suppose not, let I and J be as specified and let P and P' be two distinct processes. Let K and L be the least integers greater than I and J respectively such that R'(L) is in R(1),...,R(I) and R(K) is in R'(1),...,R'(L). K and L exist since, by hypothesis, N and M are (potentially non-minimal) cantidates for K and L. Now consider:

S(1) = REQUEST(So,P,R(1),...,R(I))

S(2) = REQUEST(S(1),P',R'(1),...,R'(L))

S(3) = REQUEST(S(2),P,R(I+1),...,R(K))

(The interpretation of a sequence of resource requests to the function REQUEST should be obvious). Note that P and P' are both active in state S(1) because R(1),...,R(I) and R'(1),...,R'(J) are disjoint. Further, P' is suspended in S(2) and S(3) because R'(L) is in R(1),...,R(I) but P continues to be active in S(2). But P is suspended in S(3) because R(K) is in R'(1),...,R'(L). Inspection of S(3) shows:

R(K) ∈ RESOURCES(S(3),P') and R(k) ∈ REQUESTS(S(3),P)

R'(L) ∈ RESOURCES(S(3),P) and P'(L) ∈ REQUESTS(S(3),P').

This implies that GRAPH(S(3)) contains the cycle P,R(K),P',R'(L). By lemma 3 S(3) is a deadlock state. This contradicts the assumption that CONTROL is safe and so proves that the sequences R(1),...,R(I) and R'(1),...,R'(J) cannot be disjoint. ∎

Remark: The next two results characterize local controllers which are based on the order rule and a total order of the resources. The first shows the equivalence between total orders and complete safe local controllers. The second states that such controllers are maximal.

Proposition 9: Suppose the system contains at least two processes.

(i) If CONTROL is a complete and safe local controller it induces a total order < on RESOURCES.

(ii) If < is a total order, it induces a complete and safe local controller.

Proof: (safe,complete => total) Let < be the order induced on RESOURCES by the complete and safe controller CONTROL. If the system has only one resource a safe controller will not allow ACCEPT({R},R) = TRUE because that produces the state REQUEST(REQUEST(So,P,R),P,R) for each process P. Such a state has the cycle (R,P) (P,R) and so by lemma 3 is not safe. Hence R $\not<$ R and < is a (vacuous) total order of the singleton resource set. Hence assume the system has at least two resources. Consider any two distinct resources R(1) and

R(2). By completeness the set {R(1),R(2)} may be requested and either R(1),R(2) or R(2),R(1) is a valid request sequence. So either ACCEPT({R(1)},R(2)) = TRUE or ACCEPT({R(2)},R(1)) = TRUE. This shows that either: $R(1) < R(2)$ or $R(2) < R(1)$ which establishes the trichotomy $R(1) < R(2)$ or $R(1) = R(2)$ or $R(1) > R(2)$.

Let R(1) and R(2) be any two distinct resources. By completeness, the set {R(1),R(2)} may be requested. By the symmetry of the problem, assume without loss of generality that:

(iii) R(1),R(2) is a valid request sequence.

and hence that $R(1) < R(2)$. For the sake of contradiction suppose that $R(2) < R(1)$ also. Then by the definition of $<$ there is a set of resources RS containing R(2) such that:

(iv) ACCEPT(RS,R(1)) = TRUE.

By completeness, RS is in REQUEST_SETS(CONTROL) and there is an ordering $\underline{R}' = R'(1),\ldots,R'(N)$ of RS which is a request sequence for RS. Combining these two results implies that:

(v) $\underline{R}'$,R(1) is a valid request sequence for RS U {R(1)}.

Since R(2) is in RS, R(2) occurs in $\underline{R}'$. Thus the conjunction of (iii) and (v) satisfy the hypothesis of lemma 8. Hence R(1) must occur in the sequence $\underline{R}'$. But by definition 2.ii.a and by (iv) above, R(1) is not in RS and so R(1) is not in $R'(1),\ldots,R'(n)$. This contradiction establishes that $R(2) \not< R(1)$. This establishes the antisymmetry of $<$.

To show transitivity, suppose that $R(1) < R(2) < R(3)$.

By completeness the set {R(1),R(2),R(3)} can be requested. Let R,R',R'' be a valid request sequence for the set. Note that R < R' and that R < R'' and by hypothesis R(1) < R(2) and R(2) < R(3) so that by antisymmetry, R(1) ≠ R' and R(1) ≠ R''. Thus R = R(1) and R(1) < R(3). This establishes the transitivity of <.

Hence if CONTROL is complete and safe it induces an order on RESOURCES which is total.

(total order induced => complete and safe) Let CONTROL be the controller induced by the total order <. By lemma 6, CONTROL is safe. Let RS be any resource set. Let R(1), R(2),...,R(n) be the ordering of RS implied by <. Since < is total the sequence exists. By definition 4.ii:

for each i = 1...n; ACCEPT({R(1),...,R(i-1)},R(i)) = TRUE.

Hence R(1),...,R(n) is a valid request sequence for RS and so CONTROL is complete. ∎

<u>Proposition 10</u>: A controller induced by a total order is maximal.

<u>Proof:</u> Let < be a total order on RESOURCES which induces the controller CONTROL. Suppose that CONTROL' is a safe local controller extending CONTROL. Let <' be the ordering on resources induced by CONTROL'. First observe that by definition 4, < is a subset of <'. By proposition 9.ii CONTROL is complete. But CONTROL' contains CONTROL so CONTROL' is also complete. Applying proposition 9.i to CONTROL' shows that <' is a total order. Since a total order

has no proper extension to a new total order and since < is a subset of <' it follows that < = <'. Hence < extends <' also. Applying this result to proposition 7 shows that CONTROL' is a subset of CONTROL. Thus CONTROL' = CONTROL and CONTROL has no proper safe extensions. Hence CONTROL is maximal. ∎

Proposition 11: Let CONTROL be a local controller. If CONTROL can not be extended to a total order induced controller then: (i) There exists an integer N $\geq$ 2,

There exist resources R(1),...,R(N),

There exist resource sets RS(1),...,RS(N),

such that:

ACCEPT(RS(i),R(i)) = TRUE for i = 1,...,N

and R(i-1) $\in$ RS(i) for i = 2,...,N; and R(N) $\in$ RS(1).

Further, CONTROL is safe iff:

(ii) There does not exist a state S $\in$ ORBIT(CONTROL,So),

such that:

There exist distinct processes P(1),...P(N):

RESOURCES(S,P(i)) = RS(i) for i = 1,...,N,

where N, R and RS are as in (i).

Proof: If the order induced by CONTROL, <, has the property that the transitive closure of <, <+, is a partial order then by Proposition 7, CONTROL may be extended to a total order induced controller. Hence if CONTROL has no such extension, then <+ cannot be a partial order. <+ is by definition transitive so if <+ is not a partial order then

it must fail to be antisymmetric or to be irreflexive. Hence for some resources R(1) and R(2), either (iii): R(1)<+R(1) or (iv): R(1)<+R(2)<+R(1). In the second case, transitivity of <+ implies P(1)<+R(1) and so (iii) holds in either case. If <+ satisfies (iii) then for some sequence of resources R(1),...,R(N), R(1)<R(2)<...<R(N)<R(1). Since < is induced by CONTROL, Definition 4 requires that there be resource sets RS(1),...,RS(N) such that ACCEPT(RS(i),R(i)) = TRUE for each i and where R(i-1) ∈ RS(i) for each i = 2,...,N, and P(N) ∈ RS(1). Hence, if < cannot be extended to a total order, condition (i) holds. This establishes the first clause of the proposition.

To establish clause (ii), first prove that the state S described implies a deadlock state. In state S, each process P(i) may request (could have requested) resource R(i) since that is allowed by the accept predicate. The resulting state has the cycle: P(1), R(1), P(2), ... , P(N), R(N). By Lemma 3, this resulting state is a deadlock state. Thus if control is safe, S ∉ ORBIT(CONTROL,So).

Conversely, suppose CONTROL is not safe. Then we will show that a state S satisfying (ii) must be in ORBIT(CONTROL,So). Let S be a deadlock state reachable from So under CONTROL. By Lemma 3, GRAPH(S) has a cycle P(1), P(1), ...P(N), R(N). Let RS(i) = RESOURCES(S,P(i)) for each i. Then since S is reachable from So, ACCEPT(RS(i),R(i)) = TRUE for each i = 1,...,N. And by the definition of GRAPH(S), R(i-1) ∈ RS(i) for i= 2,...,N and R(N) ∈ RS(1).

This is exactly the criterion stated in (ii). Hence if CONTROL is not safe its orbit contains some state satisfying (ii). This proves the second clause of the proposition. ▪

Remark: The above results have implications for the design of safe protocols for local controllers. One might also ask whether they can be used to automatically evaluate the safeness of designed protocols. In particular, could they be used to mechanize the logic found in most correctness proofs given for lock and semaphore systems? It appears that this can be done to some extent as was indicated by the example of figure 3.4. In general, if the locks and semaphores are used for synchronization rather than communication, then these results seem to apply. The supporting definitions and results are presented here and will be exploited in a later paper.

To test the safeness of an arbitrary controller would require the enumeration of all states in the orbit of the controller and then the checking of each such state. There are at most |PROCESSES|**(2**|RESOURCES|) such states. One would apply lemma 3 to each such state to check its safeness. This is a nontrivial computation if there are more than three processes or resources.

Not suprisingly, the problem of deciding the safeness of a particular local controller is much simpler than the more general problem of analyzing an arbitrary controller.

The computation becomes an analysis of the ACCEPT predicate of the local controller. In the worst case the accept predicate may be enormous also (about (2**|RESOURCES|)**2 elements). In practice it seems to be rather smaller (proportional to the length of the programs which embody it).

There are two steps to the process of deciding the safeness of a local controller:

FIND CYCLES:

Find a cycle as described in proposition 11.i.

TEST COEXISTENCE:

Establish that the RS(i) described in proposition 11.i may coexist in some state S in the ORBIT(CONTROL,So) as per proposition 11.ii.

If the second step can be satisfied, the controller is not safe. Otherwise it is safe.

The first problem (cycles) is easily solved by constructing the following graph and enumerating its elementary cycles (using the algorithm of Tarjan 1973).

<u>Definition 12:</u> Let CONTROL be a local controller with accept predicate ACCEPT. Define CYCLES(CONTROL) to be the directed graph defined on nodes chosen from the set:

{ (RS,R) | ACCEPT(RS,R)=TRUE; and R∈ RESOURCES and
RS a subset of REQUEST_SETS(CONTROL)}.

where:

((RS,R),(RS',R')) ∈ CYCLES(CONTROL)

if and only if

ACCEPT(RS,R)=TRUE & ACCEPT(RS',R')=TRUE & R ∈ RS'. ▪

<u>Remark</u>: The set of elementary cycles of CYCLES(CONTROL) is exactly the set described by proposition 11.1 given that the cycle has length N and that there are at least N processes in the system.

The second question (coexistence) is much more difficult to decide. For each sequence of resource sets generated by the first step, it must be shown that there is no state S in ORBIT(CONTROL,So) such that: RS(i)=RESOURCES(S,P(i)) for some sequence P(1),...,P(N) of distinct processes. Clearly, if for some i≠j, RS(i) and RS(j) have an element in common, then by the assumption that processes have exclusive access to the resources, it is impossible that the sets RS(i) and RS(j) be the resource sets of two distinct processes in some state. This simple test for pairwise disjointness of the RS(i) is not <u>sufficient</u> to imply deadlock but it is <u>necessary</u>. Figure 3.4.c gives an example in which the sets {A1,A2,B} and {A1',A2',C} are disjoint but in which no state in ORBIT(CONTROL,So) can have both of these two resouece sets assigned to two processes simultaneously. In fact the system is safe. The logic of the latter assertion was described in section 3.

We have only a trivial characterization of the general test required to determine the coexistence of a set of

resource sets. However it is possible to state an important sub-case in which the disjointness test is necessary and sufficient for deadlock detection.

Definition 13: A controller CONTROL is direct if for each RS in REQUEST_SETS(CONTROL) there is some ordering of the elements of RS, R(1),...,R(n) such that ACCEPT(Ø,R(1)...R(n))=TRUE. ∎

Remark: Consider the protocol implicit in critical sections (Brinch Hansen 1971). Resources are requested and released in the same order that blocks of the procedures are entered and exited. Thus they form direct controllers since any request set may be expressed the prefix of some stack of resource requests. The example of figure 3.4 also is direct.

Lemma 14: If each process of a local controller always releases its most-recently-requested-resource first then the system is direct.

Proof: Induct on the number of transitions, N, from state So to state S to show that for any process P, the set RS = RESOURCES(S,P) may be ordered R(1),...,R(m) so that ACCEPT(Ø,R(1)...R(m))=TRUE. Clearly, for N=0 this is vacuously true since each such RS=Ø. Suppose that S(o),...,S(N) are such that each S(i) satisfies the hypothesis for i<N, and suppose that (S(i-1),S(i)) ∈ CONTROL for i=1,...,N. Then consider the process P such that for some resource R:

(i) S(N) = REQUEST(S(N-1),P,R),

or (ii) S(N) = RELEASE(S(N-1).P,R),

By definition 2, P and R exist if S(N) exists.

Let

RS = RESOURCES(S(N-1),P),

and RS' = RESOURCES(S(N),P).

By hypothesis, RS may be ordered R(1),...,R(m) so that ACCEPT(Ø,R(1)...R(m))=TRUE.

In case (i) definition 4 implies that ACCEPT(RS,R)=TRUE so ACCEPT(Ø,R(1)...R(m)R)=TRUE also. Hence in case (i) S(N) satisfies the induction hypothesis.

In case (ii), we conclude that R=R(m) since process P must release it's most recently requested resource first. Hence, RS' = RS-{R} and by hypothesis ACCEPT(Ø,R(1)...R(m-1))=TRUE so RS' satisfies the induction hypothesis in this case.

In both cases the induction is extended and we conclude that the hypothesis is true for each state S in ORBIT(CONTROL,So). Hence, CONTROL must be a direct controller. ∎

Remark: The next result shows that pairwise disjointness of the request sets is sufficient to guarantee their coexistence in some state in ORBIT(CONTROL,So) if CONTROL is direct. This then gives a simple test for the safeness of a collection of procedures built on critical sections.

Proposition 15: Let CONTROL be a direct local controller. Let RS(1),...,RS(n) be any sequence of sets chosen from REQUEST_SETS(CONTROL).

There is a state S ∈ ORBIT(CONTROL,So) such that for some distinct processes P(1),...P(n): RESOURCES(S,P(i)) = RS(i) for i=1,...,n, if and only if

(a) the system has at least n distinct processes.

and (b) the RS(i) are pairwise disjoint sets.

Proof: (=>) Suppose that a state S exists which satisfies the forward hypothesis. Then there are at least n processes in the system because the processes P(1),...,P(n) are distinct. Further, By the definition of RESOURCES(S,P(i)) (see definition 2) the resource sets RS(i) must be disjoint for at most one process can be at the front of a resource queue.

(<=) Let P(1),...,P(n) be distinct processes. Since CONTROL is direct, each RS(i) may be ordered R(i,1),...,R(i,m(i)) so that ACCEPT(Ø,R(i,1)...R(i,m(i))) = TRUE. Consider the state S(n) constructed by:

S(1) = REQUEST(So,P(1),R(1,1)...R(1,m(1)))

.
.
.

S(n) = REQUEST(S(n-1),P(n),R(n,1)...R(n,m(n)))

By the disjointness of the RS(i), each resource queue has at most one process in it. Thus none of the P(i) become suspended and hence S(n) is in ORBIT(CONTROL,So). Further, RESOURCES(S(n),P(i)) = RS(i) for each i. Thus S(n) satisfies the first clause of the proposition and the proposition is

established in both directions of implication. ∎

ACKNOWLEDGMENTS

This paper benefited from many stimulating discussions with Irving Traiger and with Don Slutz.

REFERENCES

Bensoussan, A., "Overview of the Locking Strategy in the File System.", MULTICS System-Programmers' Manual Section BG.19.00, pp. 1-6. Project MAC, Cambridge, Mass. (1968).

Brinch Hansen, P., "Short Term Scheduling in Multiprogramming Systems.", Proceedings of Third Symposium on Operating Systems Principles, pp. 101-105. (1971).

Goldstein, B.C., "On the Resolution of Deadlocks.", IBM Technical Report TR 00.2176-1, Poughkeepsie Laboratory. pp. 1-17. (1973).

Gray, J.N., "Locking.", Concurrent Systems and Parallel Computation, Conference Record (J.B. Dennis ed).. ACM (1969).

Needham R.M., and Hartley D.F., "Theory and Practice in Operating System Design.", Proceedings of the Second ACM Symposium on Operating Systems Principles, pp. 8-12. Brandon Systems Press. Princton, N.J. (1969).

Habermann, A.N., "Prevention of System Deadlocks.", Communications of the ACM. Vol. 12, No. 7. pp. 373-377. (July 1969).

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

Hebalka , P.G., "Deadlock-Free Sharing of Resources in Asynchronous Systems." Project MAC Technical Report TR-75. Project MAC. Cambridge, Mass. (1970).

Holt, R.C., "On Deadlock in Computer Systems.", Technical Report CSRG-6, University of Toronto, Toronto Canada. (April 1971).

IBM, "Information Management System / 360, Version 2. System Manual" Vol. II, Form No. LY20-0630-2, pp. 18-22. IBM Corporation, Armonk N.Y. (1972).

IBM, "IBM System 360 Operating System, MVT Supervisor, Program Logic.", Form No. GY28-6659-6. pp. 49-58. IBM Corporation, Armonk N.Y.. (1970).

Knuth, D.E., The Art of Computer Programming Vol. 1, pp. 334-336. Addison Wesley Publishers, New York., (1968).

Tarjan, R., "Enumeration of the Elementary Circuits of a Directed Graph.", SIAM Journal of Computers, Vol. 2, No. 3. pp. 211-216. (September 1973).