Distributed Deadlock Detection in Concurrent C

by

Scott William Hammond

B. S., Kansas State University, 1986

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Virgil E. Wallentine - Major Professor

Table of Contents

Table of Figures

1. Introduction

One often finds that extremely useful tools or applications can be abstracted into language constructs whose methods can be hidden from the user. Interprocess communication is a good example of this as shown in the Concurrent C transaction call [GeRo86], and the Ada rendezvous [DoD83]. Ideally we would like to see deadlock detection and resolution available at the language level as well. Since most concurrent languages involve a run time kernel which implements facilities like process management and interprocess communication, it is a natural place to put deadlock detection.

In this thesis we describe an implementation of a distributed deadlock detection algorithm in the distributed kernel of Concurrent C. In particular, we have considered the relationship between local and global deadlock detection and made some special improvements on published deadlock detection algorithms [CMH83] that can be made because we have special implementation (kernel) knowledge. Our specific implementation is in support of Ed Vopata's distributed discrete event simulator [Vop88] (for which we detect deadlock).

The distributed discrete event simulator [ReFu87] is an especially interesting application because in this particular implementation deadlock is a naturally occurring event. The deadlock must therefore be detected and resolved possibly many times during the course of a given simulation. More specifically, the simulator represents simulation elements, like queues and servers in a queuing model, as individual processes. The "jobs" which get passed through the simulation are represented by data objects and travel from process to process via

synchronous messages. Present in each job (object) is a simulation time. When jobs may arrive from several paths at a single point, the simulator must be careful which message to accept first. A message from each branch must be present, and then the message with the smallest time value in it must be received first. After a message is received, this node must wait again until there is a message for each incoming path before choosing another one to receive.

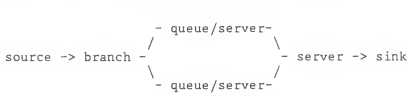


Figure 1.1: a simple simulation model

With this in mind, picture a simple model where a path branches in two, each branch leading to a queue-server pair, and then joining again at a server. The choice of paths to take at a branch is based on probability, and it may be the case that one of these paths has a very high probability. One can envision, then, most of the messages flowing through the one side. The first message passes through the queue, into the server, and then sits as a pending message at the join point while the join server process waits for a message from the other path so that it can choose, based on simulation time, which message to receive.

Because path choice at a branch is based on probability, we would expect occasional jobs to come along the other path and the simulation would continue. This is mostly true. However the queue along the heavily traveled path is not infinite in size. Therefore it may reach

a point at which it is full if there is a sufficient number of jobs in the system. In this case the branch point will get blocked trying to send a message to the queue (which will refuse to receive the message when it is full). We can now see that we have deadlock, because if the branch point is blocked, then there is now no way for an occasional job to make it along the lower probability branch to the join point. The node at the join point is waiting for a message which will never arrive.

A simulation may reasonably be designed so that the above situation is not an unlikely case. Nor is it the only way the simulation may deadlock. But if deadlock is natural, then so must be the detection and resolution thereof. Our intent with this project is to implement the deadlock detection independently of the application. That is, we tried not to take special advantage of the application in the deadlock detection. Because of the intended separation of detection and application, the resolution becomes a responsibility of the application, though necessary low level (kernel) facilities need to be provided to permit this. The resolution must be handled by the application because the detection software has no understanding of what the application is trying to do.

In the case of distributed simulation, resolution involves sending special NULL jobs (usually referred to as NULL messages) to the join processes. The resolver must determine the appropriate points to insert such messages when notified that deadlock exists. Kernel facilities have been developed which provide the resolver the ability to query the state of a process, and to intercept the current simulation time.

The rest of our paper will discuss the general deadlock problem, local deadlock detection and kernel knowledge, global deadlock detection, and finally our implementation of distributed deadlock detection in Concurrent C.

2. Deadlock in Concurrent Systems

The deadlock problem is a fundamental concurrent programming concept. Simply defined, a process is deadlocked if it is waiting for an event which will never occur. Obviously one process, if coded correctly, won't deadlock without the cooperation (or lack thereof) of some other process or external entity.

Deadlock is of particular concern in resource allocation. Many processes are vying for the same resources and it is possible that several processes will each get some but not all of the resources they want. If they are coded in such a way that they must have all requested resources to continue, and no process is able to complete its resource request, then each process will be blocked, waiting for another to free a resource. The group of processes will be deadlocked.

In a simple example there are two processes, a disk drive, and a tape drive. Each process needs one of each to run. It may be the case that one process runs, gets allocated the tape drive, but before it can request the disk the second process gets scheduled. The second process requests the disk drive and gets it. Now it tries to request the tape drive, but there are none to be allocated so it blocks, waiting for a free tape drive to show up. The first gets rescheduled and it makes its request for the disk. It also blocks, waiting on the availability of the disk drive. Now neither of them can run, each waiting on the other; they are deadlocked. This kind of circular dependency is typical of deadlock situations.

We can model the above example by representing each resource allocator

as a process. Resource allocators are passive agents, "accepting" messages from requesting processes only when they can grant the request. A requesting process is an active agent in the sense that it sends a message to a specific resource allocator when it wants something. Communication between processes is synchronous.

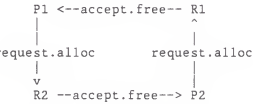
In the example above, we envision each application process as having completed a message exchange with different resource allocators, each being allocated one of the resources it requested. Having done that, they each now have requests pending outside of the other resource allocator.

```
P1 --request.allocate---> R2
P2 --request.allocate---> R1
```

There is another relationship, implicit, between the resource allocators and the processes which have received resources (because the allocators expect the eventual return of their resources).

```
R2 --accept.free--> P2
R1 --accept.free--> P1
```

We can therefore represent general resource allocation as a communication graph.



This permits the treatment of the deadlock in a generic fashion as

communication deadlock. Resource allocation deadlock detection is now a matter of detecting communication deadlock. Ideally the detection need know nothing about the application which is deadlocking, but can be abstracted from such details.

There are four necessary conditions for deadlock to exist:

1. Processes claim exclusive control of the resources they require (mutual exclusion)
2. Processes hold resources already allocated to them while waiting for additional resources (wait for)
3. Resources cannot be removed from the processes holding them until the resources are used to completion (no preemption)
4. A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain (circular wait)

In the above example, each of these conditions holds for the resource allocation graph, as well as the communication graph. Mutual exclusion is a stated property of the system. The wait for property and the no preemption property are true because of the synchronous communication mechanism. A process cannot tell that its request will get queued, nor can it undo the communication once it is initiated. Likewise, a resource can't be preempted because a process already involved in a communication can't be interrupted to talk to someone else (assuming that a resource would only be preempted from a blocked process which was waiting for more processes). Finally, the circular chain is evident when viewing who is waiting to communicate with whom.

There are many ways to deal with deadlock. Two basic approaches are

either to avoid deadlock in the first place, or to let it happen and then try to detect and resolve the deadlock. Either approach is aimed at affecting one of the necessary conditions for deadlock.

Deadlock avoidance usually changes the way a process is allowed to request resources. One such way is to require that each process allocate all of the resources it needs in one atomic command. This will deny the wait for property since no process will be holding a resource while waiting to be allocated another.

Deadlock resolution can take a variety of forms, depending on the condition being denied. If preemption were permitted, then an already allocated resource could be taken away from the process which had it, and be given to another process to fulfill that process's resource request. In distributed simulation, one would want to break the circular chain property by sending a null message to one of the blocked processes.

Our goal is to view deadlock in terms of communication. Though we will treat deadlock with respect to a specific communication model, it is useful to discuss first the different models of deadlock from a communication perspective.

2.1. Models of Deadlock

There are four basic models of deadlock in resource allocation [Kna87] corresponding to the complexity of resource requests. We will take a brief look at each of these.

2.1.1. One-resource Model

This is the simplest model, where a resource allocator can have at most one outstanding resource. The requesting process can likewise have at most one outstanding request. Hence any node on the wait-for graph (WFG) has a maximum outdegree of 1, and finding deadlock is a matter of finding a cycle.

Think of representing a resource allocator as a process, with that process holding exactly one resource. A user process may have at most one pending message. The process may request a resource by initiating a send (process call in Concurrent C), the completion of which denotes allocation of the resource. The allocator then waits (attempts to receive) for another message from the user process signaling the return of the resource, thereby establishing a dependency on the process presently holding the resource. The communication graph depicts the wait-for graph (WFG) and therefore detecting a cycle of communicants detects deadlock.

2.1.2. AND Model

In the AND model, a process may request a set of resources at one time. Each of these resources must be available to grant the request, otherwise the process is blocked. Deadlock detection is as the previous model a matter of finding a cycle.

In terms of communication the AND is represented by the ability to send a message to multiple destinations in one operation (the message must be received at all destinations to complete) and a process may receive messages from several sources at once (there must be a message

from each source requested). However, the receive does not play a role because resource allocations don't work that way. A user process may request several resources simultaneously from several allocators using an AND-send. Upon allocation, each allocator then initiates a single receive, waiting for the deallocate message. An allocator cannot allocate to anyone else because the receive would then have to be an OR'ed receive. Realize that this means there can't be more than one user process which can call this allocator. Hence a strict AND communication model isn't very useful, and does not fit Concurrent C, our target language. Finally, when all is said and done and we have again a communication graph which models the WFG. Due to its AND nature, detecting a single cycle detects deadlock.

2.1.3. OR Model

In the OR model, a process issues a request for a set of resources. This request is granted if any of the resources is available.

This corresponds to multiple sending again, but where any receiver is sufficient. Likewise the receive becomes a selective receive where any message may be received. In practice this is most evident at the resource allocator for now it is possible to manage many resources of a given type; allocating them to more than one user process. The allocator can then initiate select receives to all processes which have been given resources and each may return its resource at any time. The communications graph once again is an image of the WFG, but the meaning is more complicated. Because of the OR nature, locating a single cycle may be insufficient to detect deadlock. Now, any dependent may be capable of freeing the deadlock since any one caller may

free a waiting process. In terms of the WFG, we must look for a "knot". If a process A is in such a knot, then for every process B reachable from A, A is reachable from B (by following the WFG). It is essentially a matter of checking all dependency paths.

2.1.4. AND-OR Model

As the name suggests, the AND-OR model is a combination of the previous two models where there can be any mixture of AND and OR resource requests. The resulting communication graph does not lend itself to any specific graph theoretic description of deadlock, but as the most general aspect is the OR model, detection of a knot will detect deadlock. However, a more efficient algorithm can be developed [HeCh83].

The OR model being less restrictive, it is possible to implement both the AND and the AND-OR models in it.

2.2. Deadlock in Concurrent C

To properly discuss communication deadlock detection, we must present a specific communication model. The language of interest is Concurrent C. Interprocess communication is via synchronous transaction calls, and is similar to an Ada rendezvous. The overall communication mechanism follows that of the OR model.

A transaction call in Concurrent C involves two elements, one active, one passive. There is the actual transaction call made by an initiating process, and there must be a matching accept by the transactee. A transaction call explicitly names the process to transact with. The

accept places no restriction on who may call (though this may be implemented in a suchthat).

Therefore a process can transact (rendezvous with another process) by naming a process and a transaction name within that process. A process can accept simply by saying it will accept one or more particular transaction types. An accepting process can select one of many transaction call types to accept. Each accept may further be qualified with a guard and a suchthat clause. The guard is a boolean expression using strictly local or global variables. The guard must be true before even considering accepting a transaction call type. The suchthat is also a boolean expression, but it can contain parameter values from the incoming transaction call as well as local/global variables of the accepting process. If the suchthat clause is true, then the transaction call will be accepted. Order of acceptance is first-in-first-out. Note that the suchthat will be evaluated for each transaction call in the incoming transaction queue until it evaluates to true, unless there are none, in which case the process will go back to sleep.

Here is a short Concurrent C example of a producer and consumer where the producer reads a character from standard input, sends it to the consumer, who writes it to standard output.

```
process body producer (process consumer)
{
    char c;
        while ((c = getchar()) != EOF)
            consumer.put(c);
}
process body consumer ()
{
    for (;;)
        accept put(c) {
            printf("%c");
        }
}
```

Figure 2.1: Simple Concurrent C producer consumer.

Given the Concurrent C communication model, lets try the simple deadlock example again. The accepts remain the same, though it is a transaction call instead of a message being accepted. The user processes make transaction calls to request resources. The transaction calls are queued outside of the accept.allocates because there are no resources left to allocate (a false guard). However the accept.free has a true guard because there are resources allocated. Figure 2.2 shows sample code for a simple resource allocator process.

```
#define TOTAL_RESOURCES 1
process body res_alloc()
{
    int number_resources = TOTAL_RESOURCES;

    for (;;) select {
        (number_resources > 0) : accept allocate () {
                number_resources--;
        }
        or
        (number_resources < TOTAL_RESOURCES) : accept free () {
                number_resources++;
        }
    }
}
```

Figure 2.2: A simple resource allocator process.

```
process user1 (process RA1, process RA2)
{
    /* .... do stuff .... */
    RA1.allocate (); /* get resource 1 */
    RA2.allocate (); /* get resource 2 */
    /* ...use resources... */
    RA1.free(); /* release resources */
    RA2.free();
}

process user2 (process RA1, process RA2)
{
    /* .... do stuff .... */
    RA2.allocate (); /* get resource 2 */
    RA1.allocate (); /* get resource 1 */
    /* ...use resources... */
    RA2.free(); /* release resources */
    RA1.free();
}
```

Figure 2.3: Two user processes programmed to request allocate in different orders.

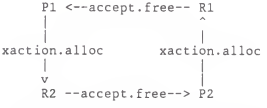
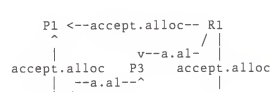


Figure 2.4: Communication graph from executing processes in figure 2.3.

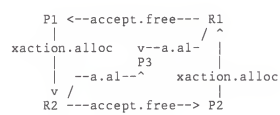
Figure 2.3 shows two user processes which will make their resource allocation requests in different orders. Execution of these processes could result in the communication graph shown in figure 2.4. A trivial method of deadlock detection in this situation is to observe that all processes are unable to run. Since it is highly unlikely that deadlock will involve all processes, this method is at best useful only when identifying programmer error (this method is used by Concurrent C).

Local deadlock detection (local, meaning the system on one virtual processor) is a matter of walking the communication graph, checking to see who is blocked on whom, and observing the state of the process being waited for.

This works fine for the trivial example above, so let's make it more complicated. Let us say there is an additional user process P3 which isn't doing anything in particular at the moment, but could ask for a resource. We would have the following:



Where R1 and R2 could also accept allocation requests from P3. Now we let events take their course where we deadlocked last. P3 takes no action:



Viewing the above picture, we see that if we did not know something about the distinction between "alloc" and "free" then we would be unable to detect deadlock. If we only knew that R2 could be called by P1,P2,P3, then we might think that it is possible that P3 has a resource which it could return to R2, enabling R2 to grant P1's

request. Why do we even know this much (whom can call whom)? Because an "accept" is an open ticket for anyone to try to transact. Hence any accept would create an implied relationship between every other process and itself for each "acceptable" transaction call.

To keep this under control, it is useful to know just exactly which processes can make transactions call to whom. Then when looking for deadlock, we don't have to concern ourselves with the state of every process in the system, only those which we know can call.

But we need to go one step further. What is needed are dynamic dependency relations, information about who can call whom at a given instant. For instance, in the example we know that if P3 has not done anything, then it can only call allocate. Since the resource allocator is out of resources, there can be no accept-allocate relationship with any process, hence P3 can be disregarded completely, and deadlock is still detected.

If we have dependency sets for each type of transaction call, then in the above example we won't need to know anything about allocate and free. As long as the resource allocator's guard on allocate goes to false (because there are no more resources to allocate), R1 and R2 will no longer be considered dependent on P3. If the dependent set of the transaction free is maintained dynamically such that only callers of allocate are in the free set, then in the above example the false guard on allocate will dispose of P3, which does not appear in the free set. In summary, when using transaction specific dependent sets, a kernel view of the guard in an accepting process can be used to eliminate certain members of dependent sets from consideration during

a deadlock computation. This can permit detection of some deadlocks which might not otherwise be possible.

There is one last situation to consider. In the above example, the deadlock is obvious because all resources have been allocated. Let's change the situation slightly such that each resource allocator starts with three resources. P1 and P2 will attempt to acquire two of each. As before each completes a request to an allocator, and gets blocked on a transaction call to the other (because the other only has one to give). Now we add in P3, and say that it may request a resource, or not (it takes no specific action this time). By observation, we know that P3 can request and release a resource from either or both allocators, but will have no effect on the deadlock of P1 and P2. This is a deadlock we cannot detect without knowing exactly how transaction calls affect each other. This requires more application-specific knowledge than we deem appropriate for kernel based deadlock detection.

It seems reasonable to view resource allocation in terms of communication, and work with deadlock at that level. We will now turn ourselves to a more specific discussion of deadlock discussion.

3. Kernel Knowledge and Local Deadlock Detection

A concurrent programming environment generally has three components: a compiler to support a concurrent programming language, a run-time library to implement the language features, and a software kernel to manage the concurrency aspect. This is the basic design of Concurrent C.

3.1. The Kernel of Concurrent C

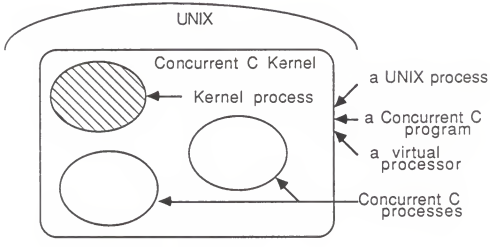


Figure 3.1: the design of Concurrent C

Implemented as a single UNIX* process, a Concurrent C program is composed of compiled user code, as well as the run-time library which contains the kernel and other support functions. As shown in figure 3.1, the Concurrent C kernel exists as a layer between the processes it implements and UNIX in much the same way as UNIX provides an

* UNIX is a trademark of AT&T

environment for its own processes. The boundary is somewhat less well defined, however, since a Concurrent C process may make direct use of UNIX system calls without passing through the Concurrent C kernel.

The kernel implements Concurrent C processes as lightweight processes---which have low context switch overhead and are most efficiently implemented in shared memory--all within a single UNIX process. The function call mechanism of the underlying hardware is employed for performing the context switch. The low cost encourages programmers to develop solutions which employ many smaller processes.

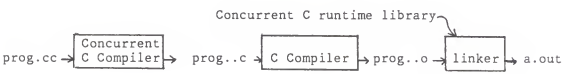


Figure 3.2: the making of a Concurrent C program

The compiler provides necessary language constructs for concurrent programming such as transaction calls and accept statements. The compiler produces pure C code which when compiled may be linked with the Distributed Concurrent C library. The library provides an implementation of functions in support of the code produced by the compiler. The kernel is actually comprised of functions which are also in the run-time library. The kernel routines are the first to take control when the program is started up. These routines handle basic process management needs such as process creation, termination, and scheduling. It resembles a small operating system.

Since the kernel is doing process management, it maintains a process control block (PCB) for each Concurrent C process. The PCB holds

information about the process's address space, outgoing transaction data, incoming transaction queue, and references to parent and children. The process table is visible to the entire library and many of the library functions depend on this information.

Though dependent on implementation, the basic approach for process creation is to allocate an area of memory and designate it as stack space for a particular Concurrent C process. A stack frame is constructed as per the function call mechanism of the hardware, and is given the initial appearance of a function which has made a call somewhere, and whose return address is specified in the stack frame as the beginning of the Concurrent C process.

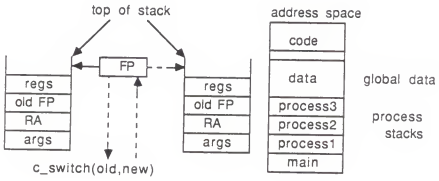


Figure 3.3: Context switch from P1 to P2 when P1 calls c_SWITCH (old, new)

The context switch mechanism involves calling a special assembly language routine, c_SWITCH(), with two parameters: the frame pointer of the process to switch to, and an address for the current frame pointer to be stored at (a location in the current processes PCB). C_SWITCH() simply takes the current frame pointer and places it in the location given in the parameter list, and takes the new frame pointer

and puts it directly into the frame pointer register. When C_SWITCH() returns (executes the return instruction) the subroutine return mechanism uses the current frame pointer* to determine the location of the stack frame to pop. Since it now has the frame pointer of a different process, context has effectively been switched.

The decision to perform a context switch is generally made by the scheduler, but there are other situations where c_SWITCH() can be called. One such time is during a transaction call. When a process makes a transaction call, the kernel function c_tc() is called to do the work. This function connects a transaction call structure to the destination process's incoming transaction queue. It then checks the state of the destination process. If it is waiting for this transaction, then c_tc() may switch context directly to the destination process.

Scheduling is likewise implementation dependent, though present implementations tend to be round robin with priority. Like real UNIX processes, Concurrent C processes are allocated a time slice, and will be preempted if they exceed their slice. The compiler also generates code to call the scheduler directly when a Concurrent C process can tell that it will block (ie. wait for a transaction when there are none to be accepted). Environment permitting, the implementation may choose to intercept certain slow UNIX system calls (ie. read()) until it knows the call can be fulfilled (ie. until the read won't block). This prevents the entire Concurrent C program from being blocked due

* Though hardware dependent, use of a single frame pointer to reference the previous frame and restore the stack seems to be typical.

to an action of a single Concurrent C process.

With the basic process control in place, it is possible to implement other features as kernel-knowledgeable processes, ie. as processes which have access to the kernel data structures. A good example of this is the Null process. This is a kernel process which is always ready to run. Its priority is adjusted such that it only runs when nothing else can. The Null process can check for system termination conditions and also can do crude deadlock detection (in the uniprocessor version). Termination can happen when all user processes are either at a select-terminate statement or have already completed. The Null process can scan the process table checking the process states to see if this is the case. Simple deadlock can be detected by checking the process states to see if each process is waiting for a Concurrent C event (like a transaction call, as opposed to a read on a slow device). Obviously all processes waiting on each other constitutes deadlock.

The process abstraction is a convenient way to add functionality to the kernel. Because of the inherent modularity of a separate process, one can readily see that new services and modification of old services might easily be accomplished. It was this observation which lead to our implementation of deadlock detection in kernel processes. Of particular interest is that different deadlock detection processes could be added, implementing different detection algorithms with little or no modification required to the kernel itself.

The distributed kernel is not much different from the regular kernel. A single UNIX process/Concurrent C program is now referred to as a

virtual processor. Interactions within a virtual processor are identical to those in the uniprocessor kernel. The difference is that multiple processors may be started, and Concurrent C processes may be created on them and may interact with all other processes in the distributed system. From a programming point of view, the primary difference is that processes may not share the same address space, so one must be careful to pass data by value rather than by reference.

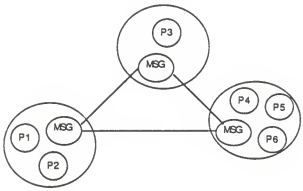


Figure 3.4: Kernel process Msg in a virtual processor maintains communication links to all other virtual processors.

The distributed kernel makes use of a kernel process on each virtual processor called Msg to manage incoming data from other processors. Msg can deliver, forward, and process incoming messages. It is this process which will see an incoming transaction call message, package the parameters into a transaction call structure, attach it to the destination process's incoming transaction queue, and determine if the process needs to be scheduled. Using a separate process for this purpose avoids the risk that the kernel itself might block trying to read a message from another processor. The present implementation is also clever enough to establish direct communication links with all other

virtual processors. The implementation is not dependent on a particular underlying transport mechanism

Because of the ease with which kernel processes could be added, modifications to the kernel to support deadlock detection were minimal. Two fields were added to the process control block. The first was p_tccallee which holds the process id of the process being called during a transaction call. This was needed because during a transaction call with a remote process, no information in the kernel on the calling side said who the callee was. When a transaction call is made, kernel function c_tc() is called, and it is in this function that p_tccallee is set (and cleared on completion of the transaction call). The other addition was the p_tcseq field. This sequence number is supposed to be updated whenever a process makes a transaction call, or waits on an accept. Updating on transaction calls is easy, and is done in c_tc(). Updating prior to waiting on an accept is more difficult and couldn't be done in the kernel. We only want the sequence number to change once before waiting. Whenever a transaction call comes in, the process gets awakened to evaluate the suchthat. Because of the way this is done, bumping the sequence number in a kernel function would occur every time the process was awakened, giving the illusion that the process was active when it really couldn't do anything. It is therefore necessary for the application programmer to place a call to special function bumpseq() prior to all select statements (and accept statements not enclosed in selects). Ideally the code to increment the sequence number should be directly added to the C source produced by the Concurrent C compiler, but modifying the compiler was beyond the scope of this project.

3.2. Local Deadlock Detection

Concurrent C has built into it a form of deadlock detection. This is really intended to catch programming mistakes (we presume) and is simply a matter of observing that there are no processes which can be run, but all are waiting for some event internal to Concurrent C (ie. not blocked on a read). This is trivial, and not useful for our purposes as it requires all processes to stop. We are interested in the deadlock of arbitrary groups of processes. Just because one segment of a simulation (for example) deadlocks, does not mean that they all should wait. Hence a more sophisticated deadlock mechanism is necessary.

The kernel data structures reflect all activity within the processor. Hence a deadlock detection process ought to be able to view the communication graph along with the process states to determine deadlock. Because of the communication model, (the fact that an accept creates an implicit relationship with all other processes), we must know which processes may choose to call which other processes. The best way to get this information is from the application itself. The present implementation is given a table of processes and the processes they may call. The deadlock detection software takes this table and inverts it into dependent sets of processes (from who P can call to who can call p). Knowing this, the deadlock process can pick a process, see if it is blocked, and scan the processes in its dependency list to determine their states. Deadlock exists if the starting process is blocked, and all of its dependents are deadlocked.

An interesting result from an implementation standpoint involves

information about non-local processes. One would think that local deadlock could only be detected if all processes and relevant dependents were local. However, in specific implementations it is possible to know what a remote process is doing. In Concurrent C an incoming transaction call is noted by attaching a transaction call structure to the destination process's incoming transaction queue on its process control block (PCB). If the transaction call cannot be accepted at the destination, then we know that the calling process is blocked, and since the identity of the caller is contained in the transaction call structure the kernel can tell what processes are blocked. Hence a remote process might call a local process, get blocked, and the kernel would know about it. If another local process is dependent on this remote process, then this information can be used when detecting the deadlock despite the fact that a participant is non-local. This provides a potential means of partitioning the processes among systems since intuitively less effort is required to do local deadlock detection when compared with distributed deadlock detection.

```
* build new dependent sets
for all processes A
    if process B is in the transaction queue of process A then
        Dependent_set (B) = { A }
        * and remember remote-blocked processes
        if not LOCAL (B) then
                Remote_blocked = Remote_blocked union B

choose a blocked process P
let set S = { P } where P is unmarked
while S contains unmarked processes do
    Let P be an unmarked process in S
    if not LOCAL (P)  and  P in Remote_blocked
       or  LOCAL (P)  and  BLOCKED (P) then
           mark S (P)
           S = S union Dependent_set (P)
    else abort
endwhile
if not abort then
    local deadlock detected
```

Figure 3.5: local deadlock detection algorithm

The algorithm is as follows. Knowing that the dependent set of a transacting process consists solely of the transactee, we build new dependent sets for all transacting processes. We do this by looking for transaction call structures in the incoming transaction queues of all processes. The transaction call structures identify the originator and when the originator is non-local, the process id is placed in a special list of remote-blocked processes. A blocked process P is chosen as the starting point. P is placed in set S and is "unmarked". P is first checked for locality. If it is local, and is running then the computation is aborted (of course this won't happen the first time since P was specially chosen). If it is local and blocked, then it is "marked" in set S, and the dependent set of P is added to set S. If P is not local, then if it is in the special remote-blocked list it is "marked" in set S and its dependent set is added to S. Finally a new P is chosen from the unmarked processes in S. This continues until the

computation either aborts or all processes in S are marked. We are really just computing closure on the dependent sets, tracking the states as we go so we can abort at the first opportunity if necessary (as opposed to computing complete closure first and then checking their states).

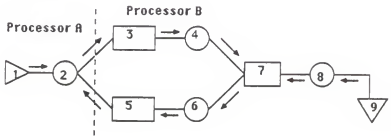


Figure 3.6: Communication graph of a simple simulation model.

Process	Dependent Set
2	3
3	2,4
4	7
5	2,6
6	5
7	4,6
8	7
9	8

Remote_blocked

2

Construction of S

7	(7)	(7)	(7)	(7)	(7)	(7)
	4	(4)	(4)	(4)	(4)	(4)
	6	6	(6)	(6)	(6)	(6)
			5	(5)	(5)	(5)
				2	(2)	(2)
					3	(3)

(p) denotes "mark"

Figure 3.7: Dependent sets for processor B and the construction of S

Figure 3.6 shows a sample communication graph from a simulation. At

server 2, the upper path has a high probability. Queue 3 has filled and is waiting for server 4 to call and get a job, while server 4 is trying to send a job to queue 7. Queue 7 is waiting for another job from server 6 so it can decide which job to accept. But, there are no jobs on the lower path, and queue 5 is waiting for one from server 2. The system has deadlocked. Now, local deadlock detection on processor B can detect the deadlock even though processes 3 and 5 depend on process 2 which is remote.

Figure 3.7 shows the dependent set on processor B and the successive states of set S. In this example process 7 is arbitrarily chosen to start with. It is placed into S. Next process 7's state is checked, found to be blocked, and its entry in S is marked. Then 7's dependent set is added to S. Process 4 is selected next (from S), found to be blocked, marked, but its dependent set (process 7) is already in S so no dependents are added. Process 6 is blocked, is marked, and its dependent (process 5) is added to S. When process 2 is handled, as per the algorithm it is identified as being non-local and is in the remote-blocked list so it is treated like any other blocked process. Finally process 3 is chosen, is marked, and its dependents are already in S resulting in an absence of unmarked processes in S. Therefore we have deadlock.

One last point is that to prevent inconsistent states from giving the appearance of deadlock when it isn't really there, this algorithm must run in a non-preemptable mode. This won't be a problem as long as local deadlock detection isn't run unnecessarily often. A possible heuristic might be used, based on the ratio of idle to active processes, for determining when detection ought to be tried.

Though the ability to detect deadlock locally even in the presence of non-local processes suggests a way of partitioning the processes, we can't expect to be able to condense all problems to this. In the next chapter we will treat the problems of distributed deadlock detection.

4. Global Deadlock Detection

Global deadlock detection in a distributed environment suffers from a lack of global state. It becomes necessary to flow information about the individual actions on the different processors amongst them until at some point deadlock, or the absence thereof is evident.

4.1. Classes of algorithms

A distributed environment complicates the deadlock problem by isolating the pieces of information necessary to determine deadlock. The various methods of reconstructing this information enable one to classify the algorithms involved into four basic groups [Kna87]: path-pushing, edge-chasing, diffusing computations, and global state detection.

4.1.1. Path-Pushing

The path-pushing method has its roots in the original construction of the WFG. The method involves construction of the local WFG at each node, and passing it to connecting nodes. Eventually one node will have enough of the graph assembled to tell if deadlock exists.

This method has drawbacks in that the underlying computation needs to be frozen during the deadlock computation in order to ensure a consistent WFG. Algorithms implementing this tend to fail to detect some deadlocks, and sometimes detect phantom deadlocks as well.

4.1.2. Edge-Chasing

Edge chasing involves sending special "probe" messages along the edges of the WFG. These probes are propagated from node to node until they reach a running process (and are discarded) or arrive back at the initiator of the probe (and thus detect a cycle). This method is only useful if finding a cycle is sufficient for the model of deadlock involved. Maekawa [1983] presents two algorithms which fit in this category, and takes special issue with avoiding the detection of phantom deadlocks.

4.1.3. Diffusing Computation

A diffusing computation is characterized by a manager process which suspects deadlock and initiates a computation following the structure of the existing application. This computation uses special "query" and "reply" messages. The queries propagate to dependent processes seeking information and the replies carry the results back. The computation actually grows and shrinks as queries and replies work their way through the system. The computation terminates when it shrinks back to its root. Chandy, Misra, and Haas [1983] and Natarajan [1986] present algorithms in this class which we will discuss in detail below.

4.1.4. Global State Detection

The main idea here is to see a consistent global state without having to suspend the underlying computation. This is related to the concept of snapshots where partial views of the system are assembled to create a picture of the whole. A partial ordering is developed based on a "happened before" relationship, which can ultimately be used to

describe a wait-for graph with a deadlock in it, as happening before a wait-for graph was built from snapshots. Hence the snapshot view is sufficiently accurate to detect deadlock. Of course there is much to consider here, but it is not relevant to our discussion.

4.2. A Distributed Deadlock Detection Algorithm

Chandy, Misra, and Haas present an algorithm for communication deadlock using an OR model of communication. We know from examining the OR model previously that the need is to detect a knot of idle waiting processes. In graph-theoretic terms, a vertex i of a directed graph is said to be in a knot if all vertices that can be reached from i can also reach i. Note that we can infer as well that an idle process waiting on a process or processes in a knot could be considered deadlocked though it does not participate directly in the knot.

The fundamental idea is that communicating processes know whom they communicate with and whom they are waiting for. All of the necessary wait-for information is present to detect deadlock, if the processes can be made to cooperate. Chandy, Misra, & Haas use a method classified by Knapp as a "diffusing computation".

Each process has a so-called "dependent set" which is the set of all processes which may send a message to this process. They further state that the process may continue upon receiving a message from any one of these. A process, upon entering the idle state, awaiting a message, may initiate a query computation to find out if it is deadlocked. From what we've already seen about diffusing computations, we can expect that the queries will propagate around, and

replies will be sent back describing what was found (and hence the computation grows and shrinks). The messages are of the form query (i,m,j,k) and reply(i,m,j,k) where m is the sequence number of the query computation initiated by process Pi and is being sent from Pj to Pk. Thus Pi is the initiator while Pj is the sender and Pk the receiver.

The authors state two properties which hold for the query computations:

> If process Pi is deadlocked then for every query (i,m,i,j) it sends, it will receive a corresponding reply (i,m,j,i).
>
> If initiator Pi has received reply (i,m,j,i) corresponding to every query (i,m,i,j) that it sent, then it is deadlocked.

In the algorithm, each process Pk has four tables of variables:

> latest(i) == the largest sequence number in any query (i,m,j,k) sent or received by Pk (initialized to 0)
>
> engager (i), for i <> k, is the identity, say j, of the process which caused latest (i) to be set to its current value m by sending Pk the message query (i,m,j,k) (initialize to arbitrary values)
>
> num(i) is the total number of messages of the form query (i,m,k,j) sent by Pk, minus the total number of messages of the form reply (i,m,j,k) received by Pk, where m = latest

(i) and j is arbitrary. Note that num (i) = 0 means that Pk has received replies to all queries of the form (i,m,k,r) that Pk sent, where m = latest (i).

wait (i) is true if and only if Pk has been idle continuously since latest (i) was last updated. Initially wait (i) is false, for all i.

From the above definitions, latest(i) is used to separate multiple queries from the same originator. Engager (i) remembers the identity of the process which sent the last query which could be acted upon. Num(i) tracks the matching of replies to queries and wait(i) remembers if a process has been idle since receiving the engaging query to which a response is being considered.

The reader can get kind of a feel for how this will work. All of the local variables are indexed by the initiator. The sequence number m is therefore used to dispose of old queries and replies from the same initiator. Num(i) tells the process when all responses to all queries to the dependent set have been received. Engager(i) then tells the process which process's query was being propagated so that it knows which process to respond to. And finally, wait(i) will catch situations where the process has not been idle.

In detail, an idle process Pi initiating a query bumps its deadlock computation sequence number (same as incrementing latest (i)), and sends query (i, latest (i), i, j) to all processes in its dependent set. It also sets num(i) to the number of processes in its dependent set and sets wait(i) = true (because Pi has been idle since latest(i)

was last changed). Whenever a process starts executing as the result of receiving a regular message (it is not processing a deadlock computation) is should set wait(i) = false for all i.

When an idle process Pk receives a query (i,m,j,k), if m < latest (i) this means a more recent query has been seen from Pi, so discard this query. if m > latest (i) this indicates a new query from Pi. Set latest (i) = m, set engager (i) = j, and wait (i) = true. Then send query(i,m,k,r) to all process in the dependent set, and set num(i) to the number of processes in the dependent set. If, on the other hand, wait (i) = true and m = latest (i) then this is a query which has been seen before and the process is still idle so send reply (i, m, k, j) to Pj. If m = latest (i) but wait (i) = false then discard the message because the process hasn't been idle.

When a process receives a reply (i, m, r, k), if m = latest (i) and wait (i) = true then the process has received a reply to a query, and it has remained idle. Therefore decrement num(i), and if num (i) = 0 and this process is the initiator then declare deadlock, otherwise send reply (i,m,k,j) to engager (i). Hence if m <> latest (i) or m = latest (i) but wait (i) = false then discard the query because the computation is old or this process has not remained idle.

On the one hand, Chandy, Misra, and Haas have a fundamentally sound algorithm. Messages for deadlock computation are small and are used in an orderly fashion. However it has a couple of drawbacks. In particular, it requires that the number of processes be statically known to size the deadlock variable arrays, and this storage is required for each process. The necessity for all this storage stems from the need

to remember what everyone has done. Natarajan [1986] proposed a similar algorithm which eliminates most of the storage requirements, and is independent of network size. He accomplishes this by using a periodic algorithm rather than trying to remember lots of state. Part of our preliminary investigation of distributed deadlock detection was to implement Natarajan's algorithm.

4.3. Natarajan's Algorithm

Briefly, Natarajan views a distributed program as a network of computing agents which cooperate by exchanging messages. Each agent has a set of output ports through which it sends messages, and a set of input ports for receiving messages. For each communication event (transaction) initiated by an agent, a communication identifier (CommId) is created as a pair <transaction number, node number>. The agent then uses this identifier in querying the states of its output ports. The CommId becomes a deadlock reference number, and each agent keeps this deadlock reference (Dref) in loose synchronization [Lam78]. As periodic port query information travels around the network, Dref will tend to the highest CommId of processes involved in transactions. This is the election aspect of the algorithm, for the agent whose CommId==Dref will be the agent to detect the deadlock.

Agents, in the course of querying their output ports and answering queries on their input ports, will suspect that deadlock has occurred when each port has taken on an "inactive" state and will initiate deadlock computations. More than one agent may initiate a deadlock computation (which means the agent involved uses a different kind of port query message), but due to the election algorithm, only one will

end up reporting the deadlock. When detecting, the agents use a different type of query to send out over their output ports which will cause queries to be propagated and eventual responses elicited. Deadlock is detected when the detector can determine that all of its output ports are "Quiet", that is, when all of the agents with whom this agent is trying to communicate with are also blocked trying to communicate with some agent, and each of their output ports is quiet, and so on.

Though Natarajan describes this as an algorithm suitable for a node kernel, it is somewhat difficult to picture that way. It appears that each process corresponds to any "agent" of his, and visualizing a kernel routine running through all this on behalf of each process is confusing. For our experimental implementation we elected to go ahead and implement it on a per-process basis in part because the method of kernel implementation was not obvious and in part because we did not have access to the Concurrent C kernel source at that time.

To gain the desired level of control over the interprocess communication we had to implement a virtual network with a separate group of processes. That is, we implemented interprocess communication through other processes of our own creation. These processes implemented the synchronous communication requirement of the application layer above, while remaining active to exchange the query and detect messages asynchronously with other processes of the same layer. Because we did not have access to the kernel at this time, we did not have a global view of the actions of local processes. Hence we could do the implementation within a single Concurrent C virtual processor and it would be the same as if we were truly distributing, but easier to debug.

This experiment became something of a study of problems in distributed implementations. The code was complicated and difficult to debug because of the concurrency. The shared memory environment within a virtual processor posed problems where multiple instances of processes wanted to invoke the same set of functions. Each function had to be reentrant and all process state information local to a given process and necessary to the function had to be passed as arguments to the function. This wasn't hard to do, but it reduced the readability and increased the confusion. The shared memory was also a problem because it is trivial for one process to destroy another's data. The lessons learned with this experiment were further reason for us to make ease of implementation a realistic criterion for our final project.

Though Natarajan's algorithm does seem to be an improvement over Chandy, Misra, and Haas, they both suffer in the area of complexity. The algorithms are hard to understand and indeed we sacrifice readability for functionality (or is it quality) when moving from CHM's algorithm to Natarajan's algorithm. In fact our first attempt at implementing Natarajan's algorithm was wrong due to a misunderstanding of the author's intent.

Both algorithms require a certain amount of per process state and both are best viewed from the perspective of the individual processes themselves. Though Natarajan states that his algorithm is suitable for implementation in the node kernel of a distributed language, how best to achieve this is not obvious. The results of our experience led us to the following qualities we wanted to see: ease of implementation, easily debugged, and easily understood.

Our target environment is Distributed Concurrent C which employs a node kernel on each virtual processor. Rather than embed the deadlock detection into each process, we would rather see the algorithm implemented in the kernel itself. This we have done, using a hybrid algorithm of our own which more suitably fit the kernel environment, and was easer to implement.

4.4. Implementing Deadlock Detection in Concurrent C

We will now describe the algorithm we implemented.

Our communication model is the OR model. Hence we are concerned with identifying the states of dependents as are Chandy, Misra, and Haas. Rather than using the diffusing computation approach, we use something similar to path-pushing.

Recall that the essence of deadlock detection in the OR model is detection of a knot. This means that for a process to be deadlocked, it must be waiting for a process or processes from its dependent set, and each of these must similarly be blocked, waiting on their dependent set, and so on. Our approach is to flow messages over each of these paths until they either loop, or reach running processes. Their status is reported back to a centralized agent responsible for the particular deadlock computation, and it is this agent which will report the existence of distributed deadlock when it has enough information.

The kernel knows which processes may call which other processes. This is the way the dependent set is handled. The information is already provided for local deadlock detection. Recall that Concurrent C

processes interact with transaction calls. For more complicated applications it may be necessary to handle separate dependent sets for each possible incoming transaction call. We do not make any distinction at this time for distributed simulation. A Concurrent C process may be "idle" because it has made an uncompleted transaction call, or it is trying to accept a transaction. The dependent set mentioned above is only relevant when accepting transaction calls. When making a transaction call, there is exactly one dependent, the destination of the transaction call being made.

The algorithm works in the following way. A kernel process called "deadlock" periodically wakes up and performs local deadlock detection. If local deadlock is not found, then it looks for a process which is either transacting with or trying to accept from a non-local process. If such a process exists, then a deadlock detection agent is dynamically created to monitor the deadlock computation. The deadlock process creates a message containing the process id of the agent, as well as listing the current process and the current process's sequence number. This message is then flowed to the member(s) of the dependent set. If the process is transacting then the message is sent to the deadlock process on the processor where the destination process is located. If the process has multiple dependents, then the message must split and flow to each dependent. The agent must be notified of the split so that it can account for the findings of each message. To keep this orderly, each message has an identifying sequence number. The deadlock process makes a split transaction call to the agent, which returns with a new sequence number. The new message, a copy of the first except for the sequence number is then flowed to a depen-

dent. Many successive splits may occur if there are many dependents.

When a message being flowed to process P arrives at a deadlock process, the deadlock process examines the state of P to see what to do next. If P is active, then deadlock cannot exist, an abort message is sent back to the agent, and the deadlock message is discarded. If P is not active it must be idle. Now the deadlock process needs to determine if this message has passed through P before (by scanning the list in the message). If the message has been to P before, the process's sequence number is compared with the number stored in the message to see if the process has been active since receiving the last message. If they match then a deadlock message is sent back to the agent identifying the message and indicating that this message found conditions for deadlock. If the sequence numbers don't match, an abort message is sent to the agent. If the message has not visited P, then it is propagated to the members of the dependent set as before.

There are some special cases to consider. In our implementation, the application specifies which processes are to be "watched" for deadlock. This is to avoid service processes, statistics collectors, and other processes which should never deadlock and usually are just support for the main activity of the application. The situation arises where a process may be transacting with one of these when a message is flowed to it. In keeping with the algorithm, the message will be dutifully flowed to the destination process. To handle this case, the deadlock process which receives the message for P will verify that P is in the set of "processes to watch". If it isn't, then an abort message will be sent back to the agent since, regardless of

```
if ACTIVE (dest_proc)
    send abort message to msg.agent
else /* blocked */

    /* if this process has been visited before */
    if dest_proc in msg.list
        /*
         * if current process seqno is the same as that
         * process's seqno stored in the message
         */
        if dest_proc.seqno = msg.list[proc].seqno
            send complete message to msg.agent

        else /* something changed since last visit */
            send abort message to msg.agent

    else
        /*
         * if making transaction call, propagate to the
         * process being called
         */
        if TRANSACTING (dest_proc)
            append (msg.list, dest_proc, dest_proc_seqno)
            dest_proc = TRANSACTEE (dest_proc)
            send msg to deadlock (PROCESSOR(dest_proc))

        else /* ACCEPTING */
            append (msg.list, dest_proc, dest_proc_seqno)
            for all P in Dependent_set (dest_proc)
                dest_proc = P
                /*
                 * the message only splits if it has to
                 * take more than one path
                 */
                if not first_time
                    msg.seqno = msg.agent.split()
                else
                    first_time = TRUE
                send msg to deadlock (PROCESSOR(dest_proc))
```

Figure 4.1: Distributed deadlock detection algorithm. A deadlock process receives a detect message "msg", which is being propagated to "dest_proc".

the state of P, deadlock cannot exist with a dependence on a process out of the set of relevant processes.

Another special case is really an optimization. It involves the

dependent set of an accepting process. If a process in the dependent set has already called, then it is unnecessary to flow a message to it since it is obvious that it is already transacting with a blocked process (the one the message is at). This can happen when a process calls, but the transaction cannot be accepted because of a false guard or suchthat.

One additional optimization is important. The problem can arise where after several splits, the paths join up again. The basic algorithm does not have a join facility, and so many messages may duplicate the same path unnecessarily. The algorithm will still work, but may tend to feed on itself, allowing more messages than necessary which when traversing the same path to a series of splits (ie. split join split) causes them to multiply even faster. The solution to this is for each process to maintain a list of deadlock detection agent ids which have passed through that process. Since we believe at most one agent should exist on each processor, the table need be only as big as the number of processors. The idea is that the agent id uniquely identifies a deadlock computation. When a message arrives at a process, the last agent id from the same processor is checked. If they are different, the new one is saved and the message is treated as usual. If they are the same, this means a message from the same computation has already been here, so there is no need for this message to continue and a completion message is sent to the agent.

The agent responsible for a deadlock computation must account for each message regardless of the state of the computation. If each message reports back that deadlock appears to exist, then the agent will notify the resolver process that indeed distributed deadlock has been

detected. If any one message triggers an abort, then deadlock cannot exist. To aid in the termination of an aborted deadlock computation, the agent will return a special invalid sequence number on subsequent split transaction calls which will be recognized by the calling deadlock process as a signal to abort the computation on its end.

Finally, to make the job of the resolver somewhat easier, each "complete" message returned to the agent will contain the entire deadlock message. The agent can then create a list which is the union of all processes in "complete" messages. This will result in a single list of all processes involved in the knot, which can then be handed to the resolver process when (and if) deadlock is reported. This can simplify the resolver's task because it doesn't have to guess or determine for itself the processes involved [Vop88].

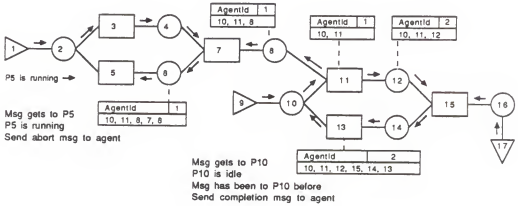


Figure 4.2: Example of distributed deadlock detection

In figure 4.2 we have an example of distributed deadlock detection. For simplicity, we do not show or keep track of the individual process sequence numbers. All processes are idle except for P5 which is

running. The figure shows deadlock detection as having started at process 10 and assume a detecting agent has already been created with a pid of AgentId. The arrows show the dependencies (A -> B means A depends on B). The deadlock detection messages are shown at different points in the computation as they flow the graph.

The message starts at process 10 and flows to its only dependent, P11. P11 is found to be idle, and having two members in its dependent set. The message splits (previously we had just message 1, now we have 1 and 2) and message 1 is sent to P8, while the other (same message but different message sequence number) is sent to P12. Each message then flows through several nodes, following simple dependencies. Note that in this example P5 is running. When message 1 gets to it, P5 is found running and immediately an abort message is sent to the agent. Message 2, on the other hand, arrives at P10 which is found to be idle. The message is checked to see if it has been to P10 before. It has, and the process's transaction sequence number is compared with the one stored in the deadlock detection message for that process to determine if it has been active since the message was last at this process. It finds the sequence number has not changed, causing a completion message to be sent to the agent.

The agent receives both the abort and the completion message, realizes that it has heard from all outstanding messages and quietly terminates (because of the abort). Now consider the same example, but this time P5 is idle, waiting on P2, and nothing else is changed. In this case the deadlock message would not stop at P6, but would continue until it reached P7. It would stop at P7 because it had visited P7 before. Since P7 is idle and has remained so, a completion message is sent to

the detecting agent. This time the agent would report deadlock to the resolver.

4.5. Analysis

Before we compare the three algorithms we first need to discuss criterion for judging them. Our experience has led us to suggest three criterion: the clarity of the algorithm, the ease of implementation from a kernel standpoint, and benefits or assistance provided to the resolver.

We say clarity, thinking specifically of how easy it is to understand. Our experience with Natarajan's algorithm showed that difficulty in understanding led to misinterpretation of the author's intent and an incorrect implementation. Because the level of complexity sharply increases in a concurrent environment, we chose the simpler, cleaner algorithm.

Ease of implementation follows directly from clarity, but we further qualify that it must be a kernel-oriented implementation. This qualification is a direct reflection of our goal in placing deadlock detection within a language kernel. We also consider ease of implementation equivalent to how easy the algorithm is to debug and how difficult it is to verify correct behavior. Again we learned from Natarajan's algorithm; when it was producing results there was the nagging question whether those results were appearing for the right reasons or not. We would like to eliminate that feeling of uncertainty.

Finally we consider assistance to the resolver. Detecting deadlock is

only part of the job. It must still be resolved. To this end we merit algorithms which can assist the resolution, generally by trying to provide more information than a simple "I am deadlocked".

Natarajan's algorithm is the most complex and the least clear. This stems from the periodic nature of the algorithm and the minimal amount of per-process state being maintained. The combinations of local variables cause different actions when messages arrive, and make it difficult to follow. Implementation is similarly difficult and confusing in part because his terms and description don't seem to match our environment. He claims his algorithm is suitable for a node kernel implementation, but just how to do this is not obvious to us. Another problem with this algorithm is debugging it in a concurrent environment. There seems to be too much to keep track of. Regarding resolution, Natarajan's algorithm will report deadlock at exactly one node. The only particular benefit this algorithm has for the resolver is that only one node will report this deadlock.

Chandy, Misra, and Haas' algorithm is not as complicated as Natarajan's. They use more per-process state information which reduces the complexity of the algorithm itself, but incurs the cost of the extra storage. In terms of implementation, this algorithm is also difficult to view in a node kernel approach. Resolution benefits are minimal, and the same deadlock may be detected by many processes.

We believe our algorithm is easy to understand. Almost all necessary state information is carried in the deadlock detection messages, and the actions performed when a message arrives are straight-forward and direct. Implementation is easier than the other algorithms, espe-

cially in terms of debugging. This is because the deadlock detection message we use holds information telling where it has been, and from examining the contents of the messages one can see how the algorithm determined the presence or absence of deadlock. We believe this to be significant since we can more easily verify that the algorithm is behaving the way we expect it to. Finally, we consider assistance to the resolver which our algorithm does best. We can provide the resolver with a list of deadlocked processes, much more than just the single process of Natarajan or Chandy, Misra, and Haas (though in the latter case this could probably be added). Further note that our local deadlock detection algorithm described in chapter 3 can also provide this information, meaning that a single resolver may be used to deal with both kinds of deadlock.

This is not to say that our algorithm is perfect. We pay a price for having larger messages than either of the other algorithms, and like Chandy, Misra, and Haas, the same deadlock may be detected by several agents.

Often the performance of distributed algorithms is characterized by the number of messages which get sent. Chandy, Misra, and Haas show that their algorithm requires at most 2nk messages where n is the number of processes, each of which has a dependent set of size k or less. Natarajan claims his algorithm is comparable with a limit of 2nk + n - 1 for the detection part of the algorithm. He does not state the cost of the election part, so it is unclear just how meaningful his figure is. In our case, the limit is 3nk - 2n + 1. We arrive at this as follows: there are nk messages propagated among processes, n(k-1) split messages sent to back to the agent, and n(k - 1) + 1

completion messages sent to the agent. This is really worst case since all processes are not likely to have the same size dependent set. Furthermore, any process making a transaction call will have only one member in its dependent set. In terms of minimum messages, it is not clear whether anything meaningful can be determined. The simplest case is where k = 1, giving a minimum of 2n for Chandy, Misra, and Haas, 3n - 1 for Natarajan, and n + 1 for our algorithm.

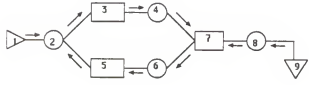


Figure 4.3: Familiar picture of deadlock

For example, in our familiar picture of deadlock, starting at process 2 Chandy, Misra, and Haas would flow queries 2-3-4-7-6-5-2 and replies would flow back 2-5-6-7-4-3-2, sending 12 messages. Our algorithm would flow 2-3-4-7-6-5-2, thus sending 7 messages (counting the completion message). When the deadlock is reported, our algorithm will list the processes involved.

In summary, we have looked at two distributed deadlock detection algorithms suitable for our model of communication. We have proposed a third which we believe to improve on the others in terms of understandability, kernel implementation, and resolution assistance. Though we pay a small price for larger messages, and have a potential for a worse worst-case in terms of sending messages, we feel that our algorithm is a reasonable alternative to those we have discussed.

5. Recovery Mechanisms

Recovery involves denying one of the basic conditions of deadlock. Resource allocation frequently makes use of preemption to free deadlock. Communication deadlock is a bit different since the reasons for why a group of processes are waiting are not evident to the kernel. Though they would be to the application writer. Our particular application environment is distributed simulation. Deadlock occurs because processes are doing accepts on processes which will never call and other processes are making calls which cannot be accepted. The basic idea to free this is to make a transaction call to the process which needs one so that it may accept and get on its way.

Distributed simulation passes timing information in the transaction calls it makes. In order to perform a proper resolution the resolving transaction call must have the correct time values in its parameters. The resolver must get the proper time from the process which would ordinarily be making the call that the resolver will have to make. If the process with the needed time is accepting, then the resolver can just ask it what the time is, and the process can be coded to accept special transaction calls of that nature even though it is deadlocked. However, if the process is making a transaction call, then the needed time information is located in the parameters of that transaction call. But that process is blocked making a transaction call (we presume) elsewhere. Hence the deadlock detection software must provide a means for intercepting pending transaction calls to get a copy of the parameters back to the resolver.

The resolver is an application layer process because the details of

the application must be known to know how to approach resolution. This limits the amount of information available to it about other processes and the deadlock. Our implementation can presently provide the resolver with a list of processes involved in a knot, but it may need to know more. To this end we provide two services to the resolver.

The first is a simple query-state(process) call which will return the state (running, accepting, or transacting) of any process in the system. If the process is transacting, query-state() will also return the process id of the transactee. With this the resolver can see the direction of the dependencies. The second service is an intercept-transaction (process a, process b). This returns the parameters with in the transaction call going from process a to process b. This only works if the call is as yet unaccepted at b. In this way we allow the resolver to intercept a message. It is up to the resolver to interpret the information and act upon it further. These two primitives are sufficient for the current needs of the resolver. Further details regarding resolution in our environment may be found in [Vop88].

6. Summary and Future Work

In conclusion, we have looked at two distributed deadlock detection algorithms suitable for our model of communication. We have proposed our own which we believe to improve on the others in terms of understandability, kernel implementation, and resolution assistance. We also developed a local deadlock detection algorithm which takes advantage of special implementation (kernel) knowledge and which enables it to detect deadlock in certain cases where local processes are dependent on remote processes. We actually did the implementation, a distributed deadlock detection package, which works with Ed Vopata's distributed discrete event simulator [Vop88].

An especially interesting area for future work involves a close look at the amount of local work which can be accomplished in the node kernel. For example, the present distributed deadlock implementation uses message passing between deadlock detection processes when flowing to any application process, even if the process is local (meaning it will send itself a message). We did this because our first priority was to get the algorithm running, and because we lacked the time to do a more complicated implementation correctly. The approach which first comes to mind is a recursive one, and since recursion is elegant but often not efficient, and because of internal stack limitations, a proper solution will require careful consideration. However one can readily see that if much of the local flow can be done internally, then the amount of message passing would drop dramatically.

There are several other ideas worth looking at. Of special interest is the relationship between local and global deadlock detection, and

the impact of the presence or absence of local deadlock detection (our global algorithm will detect any deadlock, global or local). Another topic is the question of partitioning the communication graph over processor boundaries to take advantage of the local algorithm's ability to detect deadlock in certain situations where dependencies exist with non-local processes.

Because our deadlock detection software was added to Concurrent C with little modification to Concurrent C itself, and because of the ability to implement the algorithm in independent processes, it should be possible to implement other detection algorithms as well with minimal modification to the kernel. A comparative analysis of different algorithms and different kinds of applications might prove interesting.

References

Chandy, K. M., Misra, J., and Haas, L. M. "Distributed Deadlock Detection", ACM Transactions on Computer Systems, Vol 1, No. 2, May 1983, pp. 144-156

DoD, Reference Manual for the Ada Programming Language. United States Department of Defense, 1983

Gehani, N. H. and Roome, W. D. "Concurrent C: Implementation Details--I", AT&T Bell Laboratories, 1984

Gehani, N. H. and Roome, W. D. "Concurrent C", AT&T Bell Laboratories, 1986

Knapp, E. "Deadlock Detection in Distributed Databases", ACM Computing Surveys, Vol 19, No 4, Dec 1987, pp. 303-328

Lamport, L. "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM, vol. 21, July 1978, pp. 558-565

Maekawa, M. "Distributed Deadlock Detection Algorithms Without Phantom Deadlocks", Technical Report (1983). 83-11, Department of Information Science, University of Tokyo, Tokyo, Japan

Misra, J. and Chandy, K. M. "A Distributed Graph Algorithm: Knot Detection", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, October 1982, pp. 678-686

Natarajan, N. "A Distributed Scheme for Detecting Communication Deadlocks", IEEE Transactions on Software Engineering, Vol SE-12, No. 4, pp. 531-537, April 1986

Reed, Daniel A., Fujimoto, Richard M., Multicomputer Networks: Message-Based Parallel Processing, The MIT Press, Cambridge, Massachusetts, 1987

Vopata, Edward W. "Distributed Discrete Event Simulation", Masters Thesis, Sept 1988

Appendix A: Implementation hurdles

The implementation involved much more than just coding the algorithms. The first task was to port Distributed Concurrent C to the AT&T 3b2/400's. This was necessary because Concurrent C will only distribute between homogeneous systems. Sound simple? Though uniprocessor Concurrent C is already capable of running on a 3b2, the distributed version as released by AT&T Bell Labs only distributes on systems based on 4.2/4.3 BSD UNIX (using sockets for interprocess communication), and the AT&T 3b4000 multiprocessor (using System V message queues). With the arrival of WIN/3b networking software (which includes the Berkeley sockets) for the 3b2s we were able to successfully port Concurrent C to the 3b2s.

Since the distributed code was meant for a BSD environment, we had to be familiar with some of the finer distinctions between System V and BSD. One example is interrupting system calls. In BSD UNIX, when a system call (such as read()) is interrupted and the signal handler returns control to the interrupted routine, the system call will automatically be restarted. System V on the other hand does not restart; the system call fails and errno==EINTR. Hence wrapper code needs to be inserted around some of the system calls in the multiprocessor code to disable the alarm() signal prior to making the call (alarm() is used to implement time slices) so as to avoid interrupting the call.

To get here from there we discovered a bug in AT&T's C Compiler which AT&T was previously unaware of (and was sufficiently nasty to find that we had to go to assembly language to catch it). In the course of

this port we learned that when a Concurrent C process's stack is clobbered, you only find out about it when context returns to the process whose stack was trashed. When this happens, the core dump is useless in figuring out who crashed whom and where because the stack is messed up. This is one of the toughest problems to debug. There is no mechanism to detect when a process has exceeded its designated stack area (though I think for development purposes such a thing could be implemented to catch some violations). Translation: porting took nearly as much time as writing the project itself. We also had to get C++ running so we could recompile the Concurrent C compiler to understand the keyword for asynchronous message passing.

Appendix B: Source code for local and distributed deadlock detection

```
# $Header: Makefile,v 1.1 88/06/20 13:49:15 scott Locked $

CSRC=
COBJ=
CCSRC= mkgraph.cc deadlock.cc dead2.cc dead3.cc
CCOBJ= mkgraph..o deadlock..o dead2..o dead3..o
DEFS= defs.h spec.h
CCFLAGS= -DDEADLOCK

    cc -Dc_MPCC -g -c $*.c

    CCC ${CCFLAGS} +M -g -c $*.cc

dd.a: ${CCOBJ}
    ar r dd.a ${CCOBJ}

a.out: ${COBJ} ${CCOBJ} ${DEFS}
    cc -g ${COBJ} ${CCOBJ} /usrb/scott/ccc/lib/libmpcc50g.a -lnet

dead2..o: dead2.cc ${DEFS}
    CCC ${CCFLAGS} -DPASS1 +M -g -c $*.cc

dead3..o: dead3.cc ${DEFS}
    CCC ${CCFLAGS} -DPASS1 +M -g -c $*.cc

mkgraph..o: mkgraph.cc ${DEFS}
deadlock..o: deadlock.cc ${DEFS}
```

```
static char
rcsid[]=
    "$Header: deadlock.cc,v 1.6 88/09/14 02:24:12 scott Locked $";

typedef long    c_pid, c_tp; /* defined in ccproc.h, need for pid */
/* #include "Pid.h" */
/*
#undef PASS1
#include "../../ccc/src/include/ccproc.h"
*/
#include <stdio.h>
#include <concurrentc.h>

#include "Pid.h"

#include "defs.h"
#include "spec.h"

    LIST    table[D_NPROCS];
    /*
    extern  long    startmem;
    */

    TPTR_report_deadlock d_report_deadlock; /* for use by gdetect */
    long    d_debug;
    extern process anytype dda_id;

/*
 * Deadlock is the process whose primary responsibility is deadlock
 * detection.  One such process is supposed to be created on each
 * virtual processor, and the process id of each should be stored in
 * a table called "deadmen" where the index is the processor number
 * of the process.  (ie.  process deadlock on processor 5 will
 * appear in deadmen [5]).
 */

process body deadlock (report_deadlock, Query_tab, D_debug)
{
    c_pidu proc, spid, dpid;
    c_pid   p;
    CALLERS callers;
    OUTBUF_S    outbuf;
    register LIST   *ptr;
    register int    i;
    int     status, done=0;
    int     mypid;
    int     count;
    int     timeout;
    int     detect_enabled = TRUE;
    char    *b;
    DDMSG   ddmsg;
    c_pidu  newpid;
    extern  char *getxaction();
    int     dorpt = 1;
```

```
char    flowtcall;

/* So I can identify myself */
mypid = c_mypid();
dda_id = c_nullpid;
d_report_deadlock = report_deadlock; /* place in global var */
d_debug = D_debug; /* place in global variable */
for (; !done ;)
    select {
        /*
         * putcallers (c_pidu Proc, CALLERS_S Callers)
         *
         * This transaction is called by mkgraph whose job it
         * is to inform each deadlock process in the system
         * of the list of processes it is expected to watch
         * out for.  Each deadlock process will only be asked
         * to watch out for processes local to it.
         *
         * Proc is the process id of a process to watch, and
         * Callers is a list of processes which may call this
         * one.  Note that 0 is an invalid pid, hence the
         * list may be terminated with a 0.
         */
        accept putcallers (Proc, Callers) {
            /*
             * if Proc is not local then we've got bogus data
             */
            if (!ISLOCAL(Proc)) {
                c_printf("dead: Proc %P Not local\n", Proc);
                printf("dead: Proc %#x not local\n", Proc);
                fflush (stdout);
                treturn();
            } else {
                for (i = 0; i < CALLERSIZE; i++) callers [i] = 0;
                /*
                 * copy parms so we can free the transaction right
                 * away
                 */
                for (i = 0; (Callers.callers [i] != 0) &&
                            (i < CALLERSIZE); i++)
                    callers[i] = Callers.callers[i];
                proc = Proc;
            }
            } /* free transaction call */
            /*
             * table is indexed by the matching process table
             * index
             */
            table [proc.u_px].pid.u_pid = proc.u_pid;
            /*
             * If we do NOT already have a list of callers for
             * this process
             */
            if ((ptr = table [proc.u_px].next) == 0) {
                /*
```

```
                 * if there are any callers
                 */
                if (callers[0] != 0) {
                    /*
                     * allocate space for the first entry; fill it
                     */
                    ptr = table [proc.u_px].next =
                        (LIST *) malloc (sizeof (LIST));
                    ptr -> pid.u_pid = callers [0];
                    ptr -> next = 0;
                    i = 1;
                }
                else continue; /* ??? */
            }
            else i = 0;
            /*
             * skip to end of linked list
             */
            ptr = table [proc.u_px].next;
            while (ptr -> next != 0) ptr = ptr -> next;
            /*
             * loop through callers, adding LIST structs to the
             * list as we go
             */
            for (; (callers [i] != 0) && (i < CALLERSIZE); i++) {
                ptr -> next = (LIST *) malloc (sizeof (LIST));
                ptr = ptr -> next;
                ptr -> pid.u_pid = callers [i];
                ptr -> next = 0;
            }
        or
        /*
         * Mkgraph calls done when there are no more putcallers
         * transactions to be made (we have all the info now)
         */
        accept done () {}
            done = TRUE;
    }
if DBG(20)
    for (i = 0; i < D_NPROCS; i++)
        if (table [i].pid.u_pid != 0) {
            printf(" deadman P%d: table[%d].pid = %#x\n", c_por,
                    i, table [i].pid.u_pid);
            ptr = table [i].next;
            while (ptr != 0) {
                printf("deadman: P%d:     caller= %#x\n",
                    c_por,ptr->pid.u_pid);
                ptr = ptr -> next;
            }
        }
count = 0; done = 0;
for (;!done;) {
    select {
        /*
         * ASYNC intercept (c_pidu src_pid, c_pidu dst_pid,
```

```
 *                    TPTR_intercept_response response_tptr)
 *
 * This transaction call is provided for the benefit of
 * the deadlock resolver (Ed's project).  Its purpose is
 * to intercept the parameters of a transaction call which
 * has been made but cannot complete at the callee side
 * (an unaccepted transaction call).  Ed needs this so
 * that he may determine the simulation time in the
 * caller.
 *
 * This call is implemented as an asynchronous transaction
 * call and includes as a parameter a transaction pointer
 * to call with the results when they become available.
 * This call-back mechanism is necessary because there is
 * no way in Distributed Concurrent C to return parameters
 * other than a single simple type in a transaction
 * call--the only way being via "treturn".  Regular
 * Concurrent C doesn't present this problem because the
 * shared memory environment permits passing pointers
 * (call-by-reference).
 * *
 * The local resolver process will ask its local deadhead
 * for the intercept.  If the intercept destination is
 * local, then it will snag the parameters and call back
 * the resolver.  If, however, the destination is
 * non-local, then it will forward the request to a
 * queryserv process on the appropriate virtual processor
 * for handling (the queryserv process will do the
 * call-back directly).
 */

accept intercept (src_pid, dst_pid, response_tptr) {
    if DBG(21)
        printf(
"deadman %#x: accepted intercept (src= %#x, dst= %#x)\n",
            mypid, src_pid, dst_pid);
    if (ISLOCAL (dst_pid)) {
        spid = src_pid; dpid = dst_pid;
        b = getxaction (spid, dpid);
        /*
         * if there was no transaction to intercept, a -1
         * status is returned
         */
        if (b == 0) status = -1;
        else {
            status = 0;
            /* copy the buffer */
            for (i = 0; i < OUTBUFSIZE; i++)
                outbuf.outbuf [i] = b[i];
            /* this was dynamically allocated; free it */
            free (b);
        }
        (*response_tptr) (outbuf, status);
    }
    else
```

```
        /* can I use the original parms? */
        Query_tab.query_tab [dst_pid.u_por].intercept
            (src_pid, dst_pid, response_tptr);
}
/*
 * getstate (c_pidu pid)
 *
 * This is yet another service provided for the benefit of
 * the resolver process.  It allows the resolver to
 * determine the state of an arbitrary process in the
 * system.  Just as with intercept, a request about a
 * local process will be handled directly, whereas the
 * query will be forwarded if it is non-local.
 *
 * Note that this attempts to return the result via
 * "treturn" because the necessary info can be fit into a
 * long, which is simple enough that Distributed
 * Concurrent C can deal with it.  Hence the transaction
 * is synchronous, and indeed if the query is forwarded
 * elsewhere, it must wait for that call to return.
 */

or accept getstate (pid) {
    if DBG(21)
        printf("deadman %#x: accepted getstate\n",
        mypid);
    if (ISLOCAL (pid))
        treturn (getstate (pid));
    else
        treturn (Query_tab.query_tab [pid.u_por].getstate
                (pid));
}
/*
 * This transaction is used to enable deadlock detection
 *
 * This feature is provided because in the amount of time
 * it takes the resolver process to perform the
 * resolution, the detector is liable to detect the same
 * deadlock again before the resolver has had a chance to
 * do anything about it.  Running deadlock at a low
 * priority (to reduce frequency of scheduling) didn't
 * have much of an effect.
 *
 * Detection will be disabled when it successfully reports
 * deadlock to the resolver.
 */

or accept enable_detect() {} detect_enabled = TRUE;
/*
 * This is where deadlock detection is initiated.  Note
 * that the deadlock detection doesn't run on processor 0
 * (Ed doesn't put any regular simulation processes there,
 * just collector and other support processes)
 */
or (detect_enabled && c_por != 0): delay 10.0; {
```

```
    if (/*count++ > 500*/ 1) {
        count = 0;
        c_printf(
    "deadlock[%x]: thinking of detection, ddaid= %x\n",
            mypid, dda_id);
        if (dorpt) { dorpt--; c_prpt(); }
        if (!DBG(30)) {
            if DBG(21) {
                printf("deadman %#x: calling detect\n",
                    mypid);
                fflush (stdout);
            }
            if (p = detect()) {
                if DBG(22) {
                    printf(
                "deadman %#x found deadlock at %#x\n",
                        mypid, p);
                    fflush (stdout);
                }
                /*
                 * If we can't make the report within 2
                 * seconds, forget it.
                 */
                timeout = within 2 ?
                    (*report_deadlock) (p) : 1;
                if (!timeout) detect_enabled = 0;
            }
            else
                if (!DBG(29) && dda_id == c_nullpid) {
                    ddstart1 ();
                }
        }
        else
            if (!DBG(29) && dda_id == c_nullpid) {
                ddstart1 ();
            }
    }
    c_sch(); /* good idea? */
}
/*
 * Global detection messages come from deadlock processes
 * on other virtual processors.
 */
or accept global_detect (Ddmsg, Newpid, Flowtcall) {
    ddmsg = Ddmsg;
    newpid = Newpid;
    flowtcall = Flowtcall;
    }
    c_printf("deadman %x: got glob_detect\n", mypid);
    printddmsg (mypid, ddmsg, newpid);
    ddetect (ddmsg, newpid, flowtcall);
/*
 * This allows the application layer to explicitly
 * terminate us
 */
```

```
            or accept term() {done = TRUE;}
        }
    }
}

    DEADMEN deadmen;
/*
 * process queryserv()
 *
 * This purpose of this process is to field certain queries from
 * deadlock processes on other virtual processors.  It is done as a
 * separate process (instead of calling another deadlock process)
 * because with independent activity on different processors the
 * deadlock processes might synchronously request something from
 * each other which would deadlock them.  And it simply wouldn't do
 * to have the deadlock detector deadlocked.
 *
 * One queryserv process exists for each deadlock process, one per
 * processor.
 */

process body queryserv ()
{
    char    *b;
    register    int     i;
    int mypid, status;
    OUTBUF_S    outbuf;
    extern  char *getxaction();

    /* note my pid */
    mypid = c_mypid();

    /*
     * Each queryserv process receives a putdeadguys call from the
     * buildgraph process.  It is simply a table of pids for deadlock
     * processes in the system.  The table is indexed by processor
     * number.  Note that queryserv places this into a global variable
     * so that others may use it.
     */
    accept putdeadguys (Deadmen) {
        c_printf("queryserv [%x]: got putdeadguys transaction\n",
            mypid);
        for (i = 0; i < D_NPORS; i++) {
            deadmen [i] = Deadmen.deadmen[i];
            if (deadmen [i] != 0)
                c_printf("queryserv [%x]: deadmen [%d]= %x\n",
                    mypid, i, deadmen[i]);
}
}

    for (;;)
        select {
            /*
             * For a proper treatment of this transaction call, see
             * the identical transaction call in the deadlock process.
```

```
            * This * is just the implementation of the intercept when
            * the request gets forwarded to a remote processor.
            */
           accept intercept (src_pid, dst_pid, response_tptr) {
               if (ISLOCAL (dst_pid)) {
                   b = getxaction (src_pid, dst_pid);
                   if (b == 0) status = -1;
                   else {
                       status = 0;
                       for (i = 0; i < OUTBUFSIZE; i++)
                           outbuf.outbuf [i] = b[i];
                       free (b);
                   }
                   (*response_tptr) (outbuf, status);
               }
               else {
                   /*
                    * since only deadlock processes should call, if
                    * this was a non-local process then someone
                    * screwed up
                    */
                   printf(
                   "queryserv: intercept -- non-local xptr\n");
                   fflush (stdout);
               }
           }
           /*
            * For a complete description of the getstate transaction,
            * see the identical transaction call in the deadlock
            * process.  This is just the queryserv implementation
            * which deadlock calls when it gets a non-local request.
            */
           or accept getstate (pid) {
               if (ISLOCAL (pid))
                   treturn (getstate (pid));
               else
                   printf(
                   "queryserv: getstate -- non local pid= %#x\n",
                   pid);
               treturn (0);
           }
           or terminate;
       }
}

/*
 * This is a distributed deadlock agent.  When a distributed deadlock
 * computation is initiated, an agent is created to tally up the
 * information and report the results.  This method was deemed simpler
 * than trying to get the deadlock process to manage possibly many
 * distributed deadlock computations simultaneously and stay within
 * reasonable memory limits.
 */
process body ddagent (origin, org_seqno)
{
```

```
int nummsgs = 1;
int abort = 0;
int nlist = 0;
int done = FALSE;
register int i, j;
process anytype mypid;
DDMSG UDDmsg; /* union of ddmsgs */

mypid = (process anytype) c_mypid();
printf("ddagent[%#x]: origin= %#x, org_seqno= %d\n", mypid,
    origin, org_seqno);
fflush (stdout);
c_printf("ddagent[%x]: origin= %x, org_seqno= %d\n", mypid,
    origin, org_seqno);

for (; !done;) {
    select {
        /*
         * split (c_pidu Pid)
         *
         * This transaction call is made when a distributed
         * deadlock detection message reaches a point at which
         * it must take two different paths.  In order to detect
         * deadlock, the results of following all paths *must* be
         * known.  Hence the agent must be notified of a split.
         *
         * Pid is the process at which the split is taking place
         * (not the id of the deadlock detector, but the id of
         * the process whose communication flow is causing the
         * split).
         *
         * To identify each message, it (the message) is assigned a
         * sequence number.  The split transaction returns a number
         * to use (since any one message doesn't know about any
         * siblings, nor their activity).  If the agent has already
         * determined that deadlock cannot exist (abort==TRUE) it
         * will return a sequence number of 0 (invalid as a real
         * sequence number) which tells the split point that there
         * is no need to waste resouces and continue; it already
         * failed so stop.
         */
        accept split (Pid) {
            printf(
            "ddagent[%#x]: split (Pid= %#x), nummsgs was %d\n",
                mypid, Pid.u_pid, nummsgs);
            fflush (stdout);
            c_printf(
            "ddagent[%x]: split (Pid= %x), nummsgs was %d\n",
                mypid, Pid.u_pid, nummsgs);
            treturn (abort ? --nummsgs,0 : ++nummsgs);
        }
        or
        /*
         * abort (c_pidu Pid, long Seqno)
         *
```

```
        * The abort call is made whenever a deadlock detection
        * message reaches a point where it is certain deadlock
        * cannot exist (ie.  a running process).  Any one such
        * instance aborts the whole computation.  The abort is
        * reported to the agent.
        *
        * The Pid is the id of the process whose state caused the
        * abort.  The Seqno is the sequence number of the
        * detection message involved.
        */

       accept abort (Pid, Seqno) {
           abort = TRUE;
           printf(
"ddagent[%#x]: abort (Pid= %#x, Seqno= %#x), nummsgs was %d\n",
                mypid, Pid.u_pid, Seqno, nummsgs);
           fflush (stdout);
           c_printf(
    "ddagent[%x]: abort (Pid= %x, Seqno= %x), nummsgs was %d\n",
                mypid, Pid.u_pid, Seqno, nummsgs);
           nummsgs--;
       }
       or
       /*
        * rpt_deadlock (DDMSG DDmsg)
        *
        * This call is made by a detector when a deadlock
        * detection message reaches the same process which it
        * started with, and the state of that process hasn't
        * changed.
        *
        * Note that this does not necessarily mean that deadlock
        * exists.  Any messages resulting from a split must be
        * accounted for.
        *
        * It passes the entire message for the agent's viewing
        * pleasure.
        */
       accept rpt_deadlock (DDmsg) {
           printf(
"ddagent[%#x]: rpt_deadlock, DDmsg.seq= %d, nummsgs was %d\n",
                mypid, DDmsg.msg_seqno, nummsgs);
           c_printf(
"ddagent[%x]: rpt_deadlock, DDmsg.seq= %d, nummsgs was %d\n",
                mypid, DDmsg.msg_seqno, nummsgs);

           for (i=0; i < GDLISTSIZE &&
                    DDmsg.dlist[i].pid.u_pid != 0; i++) {
               for (j=0; j < nlist; j++)
                   if (DDmsg.dlist [i].pid.u_pid ==
                       UDDmsg.dlist[j].pid.u_pid)
                           break;
               if (j == nlist) /* didn't find it */
                   UDDmsg.dlist [nlist++].pid.u_pid =
                       DDmsg.dlist [i].pid.u_pid;
```

```
            }
        }
        printddmsg (mypid, UDDmsg, 0); /* may not work */

        if ((--nummsgs == 0) && !abort) {
            printf("Distributed Deadlock!!!\n");
            c_printf("ddagent [%x]: Distributed Deadlock!!!\n",
                mypid);
        }
        fflush (stdout);
        or
        /*
         * All messages must be accounted for before we can
         * terminate; else someone might try to call us and they
         * would be decidedly upset if we weren't here.
         */
        (nummsgs == 0) : done = TRUE;
    }
}
c_printf("ddagent [%x]: terminating\n", mypid);
dda_id = c_nullpid;
}
```

```
static char
rcsid[] = "$Header: dead2.cc,v 1.5 88/09/14 02:23:55 scott Locked $";

#include <stdio.h>
#undef PASS1
#include "../../ccc/src/include/ccproc.h"
#include "Pid.h"
#include "defs.h"
#include "spec.h"

    LIST    *Stuck;
    extern LIST table[D_NPROCS];
    extern DEADMEN deadmen;
    extern long d_debug;

/*
 * makelist (LIST *gr, c_pidu proc)
 *
 * makelist adds a LIST structure to a LIST, and puts the proc info
 * in it
 */
#define makelist(gr, proc)                                  \
        {                                                   \
            register LIST *ptr;                             \
            ptr = gr -> next;                               \
            while (ptr != 0) ptr = ptr -> next;             \
            if (ptr == 0) {                                 \
                ptr = (LIST *) malloc (sizeof (LIST));      \
                ptr -> pid.u_pid = proc;                    \
                ptr -> next = gr -> next;                   \
                gr -> next = ptr;                           \
            }                                               \
        }

/*
 * inlist (LIST *head, c_pidu proc)
 *
 * When given the head of a LIST of LIST's, it will return true or
 * false depending on whether it can find proc in the LIST
 */
inlist(head, proc)
    LIST    *head;
    c_pidu  proc;
{
    register LIST *ptr;

    ptr = head;
    while (ptr != 0)
        if (ptr->pid.u_pid ==  proc.u_pid)
            return (TRUE);
        else ptr = ptr -> next;
    return (FALSE);
}
```

```
/*
 * freelist walks a LIST, freeing everything in the list as it goes
 */
void freelist (head)
LIST *head;
{
    register LIST *xptr, *yptr;

    xptr = head;
    yptr = head -> next;
    while (yptr != 0) {
        free (xptr);
        xptr = yptr;
        yptr = yptr -> next;
    }
    free (xptr);
}

/*
 * detect is the main function for detecting local deadlock
 *
 * The basic algorithm is to run through the list of
 * processes-to-watch and stick into the Stuck list each pid which
 * is stuck.  Stuck in this case is any process making a transaction
 * call which I can determine cannot be accepted by the destination
 * process.
 *
 * The next step is to find an accepting process-to-watch and
 * recursively walk a list of its callers.  If each of these callers
 * is also "stuck" then the accepting process is likewise stuck,
 * moreover it is deadlocked.
 */

detect()
{
    int i;
    register c_tcall *tcall;

    /*
     * While we're walking the process table and transactions queues,
     * it is safer to prevent disallow preemption.
     */
    c_DISABLE;

    /*
     * Build a list of stuck processes
     *   The approach is to locate all accepting processes and
     *   evaluate the pending incoming transaction calls.
     */
    Stuck = (LIST *) malloc (sizeof (LIST));
    Stuck -> pid.u_pid = 0;
    Stuck -> next = 0;
    for (i = 0; i < D_NPROCS; i++) {
        /*
         * Only processes to watch for are in "table"
```

```
        */
        if (table [i].pid.u_pid != 0) {
            if (ACCEPTSTATE (table [i].pid)) {
                if DBG(23)
                    printf(
                    "table [%d]= %#x c_procs[].p_tcin.pid= %#x\n",
                        i, table[i].pid.u_pid,
                        c_procs[i].p_tcin->tc_callerpid);
                /*
                 * look at the incoming transaction queue
                 */
                tcall = c_procs[i].p_tcin;
                /*
                 * Walk the queue of incoming transactions
                 */
                while (tcall != 0) {
                    if DBG(23) {
                        printf(
                        "detect: pid=%#x selmsk=%#x tcmsk=%#x\n",
                            c_procs[i].p_pid, c_procs[i].p_selmsk,
                            tcall->tc_msk);
                        fflush (stdout);
                    }
                    /*
                     * The process can only be blocked on select if
                     * the pending transactions couldn't be accepted
                     * either due to a false guard or false suchthat.
                     * Hence it is sufficient that the process be
                     * blocked on wselect, and there be transactions
                     * in the queue.
                     */
                    makelist (Stuck, tcall -> tc_callerpid);

                    tcall = tcall -> tc_next;
                }
            }
        }
    }
/*
 * we're finished with the worst of it, permit preemption
 */
c_ENABLE;
/*
 * Something missing here.  It is possible that no one is
 * accepting and that all are transacting.  In this case we
 * would actually expect this to be a result of incorrect
 * programming (as opposed to an expected event as in the case
 * of discrete event simulation).  Since it is "incorrect" and
 * since this version is aimed at running with the simulator
 * (not so generalized), I haven't added code to look for
 * transacting processes if it can't find any accepting
 * processes.
 */

for (i = 0; i < D_NPROCS; i++)
```

```
        /*
         * if it is a process to watch
         */
        if (table [i].pid.u_pid != 0) {
            if (ACCEPTSTATE (table[i].pid) )
                if (recurse(table[i].pid, table[i].pid)) {
                    if DBG(22) {
                        printf("DEADLOCK DETECTED!!!\n");
                        fflush (stdout);
                    }
                    freelist (Stuck);
                    /* c_ENABLE; */
                    return (table [i].pid.u_pid);
                        /* yup. Well, I hope so anyway */
                }
        }
    freelist (Stuck);
    /* c_ENABLE; */
    return (FALSE); /* nope */
}
/*
 * recurse (c_pidu start, c_pidu pid)
 *
 * Recurse is the heart of local deadlock detection.  It will
 * recursively walk backwards to callers of processes until it
 * either finds a running process (and aborts the deadlock), finds a
 * stuck process (and then the recursion stops and it will go back
 * and find some other path to check), or finds its starting point
 * (considers it stuck)
 */

recurse (start, pid)
c_pidu start, pid;
{
    register LIST *ptr;

    if DBG(24)
        printf("recurse: start= %#x, pid= %#x\n",
            start.u_pid, pid.u_pid);

    /*
     * Is this process already stuck?  (note: even if it is a remote
     * process, we might know about it)
     */
    if (inlist(Stuck, pid)) {
        if DBG(24)
            printf("recurse: pid= %#x is stuck\n", pid.u_pid);
        return (TRUE);
    }

    /* if it isn't local we can't tell... yet */
    if (!ISLOCAL(pid)) return (FALSE);

    /* not doing accept, running or transacting */
    if (!ACCEPTSTATE(pid)) return (FALSE);
```

```
    /*
     * At this point we know that the process is ACCEPTING.  We now
     * walk through its list of callers and determine what the
     * state of each is.  If they are all stuck, then we have
     * deadlock.
     *
     * Note that the processes which have called us but whose
     * transactions we cannot accept will already appear in the
     * "Stuck" list so we do NOT have to treat differently
     * processes who have called over process who haven't.
     */

    ptr = table [pid.u_px].next;
    while (ptr != 0) {
        if (ptr -> pid.u_pid == start.u_pid)
            { ptr = ptr -> next; continue; }
        if (!recurse (start, ptr -> pid))
            return (FALSE);
        else
            ptr = ptr -> next;
    }
    /* this node is stuck, so add it to the list */
    makelist(Stuck, pid.u_pid);
    if DBG(24)
        printf("recurse: pid= %#x is stuck (bottom)\n", pid.u_pid);
    return (TRUE);
}

/*
 * char *getxaction (c_pidu src_pid, c_pidu dest_pid)
 *
 * This function is called by both process deadlock and process
 * queryserv.  Its purpose is to get the parameters of a pending
 * synchronous transaction call.  Hence it implements the
 * transaction call "intercept".
 *
 * The idea is to locate the process, and find the transaction in
 * the queue by looking for a transaction with the proper source
 * address.  Since we're only talking synchronous calls, there can
 * be at most one transaction from the source process in the queue.
 * For this reason too, we don't need to care which transaction was
 * called either.
 *
 * getxaction() will return a pointer to a dynamically allocated
 * buffer which should be freed by the caller
 */

char *
getxaction (src_pid, dest_pid)
c_pidu src_pid, dest_pid;
{
    c_proc *p; c_tcall *t;
    int size; char *s;
    register char *s1, *s2;
    extern char *c_malloc();
```

```
    p = &c_procs [dest_pid.u_px];
    t =  p -> p_tcin;

    while (t != 0)
        if (t->tc_callerpid == src_pid.u_pid) {
            /* found the puppy */
            size = p -> p_desc -> p_tdesc[c_log2(t->tc_msk)].t_argsiz;
            s1 = s = c_malloc (size);
            s2 = t -> tc_args;
            for (; size > 0; *s1++ = *s2++, size--);
            return (s);
        }
        else
            t = t-> tc_next;
    return ((char *) 0);
}

#define ACCEPTING (1 << 27)
#define XACTING (2 << 27)
#define RUNNING (3 << 27)

int getstate (pid)
c_pidu pid;
{
    if (ACCEPTSTATE (pid)) {
        if DBG(25)
            printf("getstate: ACCEPTING pid= %#x\n", pid.u_pid);
        return (ACCEPTING);
    }
    if (WAITSTATE (pid)) {
        if DBG(25)
            printf("getstate: XACTING pid= %#x, callee= %#x\n",
                pid.u_pid, c_procs [pid.u_px].p_tccallee);
        return (XACTING | c_procs [pid.u_px].p_tccallee);
    }
    if DBG(25)
        printf("getstate: RUNNING pid= %#x\n", pid.u_pid);
    return (RUNNING);
}
```

```
static char
rcsid[] = "$Header: dead3.cc,v 1.2 88/09/14 02:24:02 scott Locked $";

#include <stdio.h>
#undef PASS1
#include "../../ccc/src/include/ccproc.h"
#include "Pid.h"
#include "defs.h"
#include "spec.h"

extern LIST table [D_NPROCS];
extern DEADMEN deadmen;
extern long d_debug;
       process ddagent dda_id;

/*
 * ddstart1
 *
 * The purpose of this function is to initiate distributed deadlock
 * detection
 *
 * The approach is to scan the table of processes-to-watch "table"
 * and check each process' state.  If it is making a transaction
 * call to a process which is non-local, then we become suspicious
 * that there is a possible deadlock situation.  Likewise, if the
 * process is accepting (WAITSTATE) and it has a potential caller
 * who has not called, then we try to detect deadlock.
 */

void
ddstart1()
{
    register int i,j;
    c_pidu tmppid;
    DDMSG ddmsg;
    void ddstart2();

    c_printf("deadlock: Entering ddstart1\n");
    for (i = 0; i < D_NPROCS; i++)
        /*
         * processes to watch are non-zero
         */
        if (table [i].pid.u_pid != 0)
            if WAITSTATE (table [i].pid) {
                ASSERT (i==table[i].pid.u_px, "ddstart1:");
                tmppid.u_pid = c_procs[i].p_tccallee;
                c_printf("ddstart1: WAITSTATE\n");
                c_prptp (&c_procs[i]);
                if (!ISLOCAL(tmppid)) {
                    /*
                     * out bound transaction call
                     */
                    dda_id = create ddagent
                        (table [i].pid,c_procs [i].p_tcseq);
```

```
                    c_printf("deadlock: created ddagent %x\n",
                        dda_id);
                    /*
                     * make sure the list starts with zeroes
                     */
                    for (j=0; j < GDLISTSIZE;
                        ddmsg.dlist[j++].pid.u_pid = 0);
                    ddmsg.dda_id = dda_id;
                    ddmsg.msg_seqno = 1;
                    ddmsg.dlist [0].pid = table [i].pid;
                    ddmsg.dlist[0].seqno = c_procs[i].p_tcseq;
                    ddmsg.dlist [1].pid.u_pid = 0;
                    tmppid.u_pid = c_procs [i].p_tccallee;
                    /*
                     * send a global_detect transaction to the next
                     * guy (might be myself--too painful to do right
                     * for this version)
                     */
                    c_printf(
                    "deadlock: calling deadmen[%d] {=0x%x}.gd()\n",
                        tmppid.u_por, deadmen [tmppid.u_por]);
                    deadmen [tmppid.u_por].global_detect
                        (ddmsg, tmppid, TRUE);
                    return();
                }
            }
            else {
                if ACCEPTSTATE (table [i].pid) {
                    /*
                     * is there a possible off-site caller?
                     */
                    if (possible (i)) {
                        /*
                         * go for it
                         */
                        ddstart2 (i);
                        return();
                    }
                }
            }
}
/*
 * Possible is used when an accepting process has been located and we
 * want to know if there is a possible caller off-site AND if that
 * caller has NOT called.
 */
possible (i)
int i;
{
    LIST *ptr;
    c_pidu pid;
    c_tcall *tcptr;

    pid = table [i].pid;
    ptr = table [i].next;
```

```
    while (ptr != NULL) {
        /*
         * is the possible caller local or remote?
         */
        if (!ISLOCAL (ptr -> pid)) {
            /*
             * offsite; is it in the queue?
             */
            ASSERT (pid.u_px == i, "possible:");
            tcptr = c_procs [i].p_tcin;
            while (tcptr != NULL) {
                if (tcptr -> tc_callerpid == ptr -> pid.u_pid)
                    break; /* yup, found it in the queue */
                else
                    tcptr = tcptr -> tc_next;
            }
            if (tcptr == NULL)
                /*
                 * not found in queue
                 */
                return (TRUE);
        }
        /*
         * go to next possible caller
         */
        ptr = ptr -> next;
    }
    if (ptr == NULL)
        return (FALSE); /* didn't find anything */
}


/*
 * ddstart2 is used when an accepting process has been located and
 * we've determined that there is a possibility (offsite caller who
 * hasn't called)> This function does the real work of splitting the
 * message off on all paths.
 */

void
ddstart2 (i)
int i;
{
    register int j;
    int firstone = TRUE;
    c_pidu pid, tmppid;
    /* ptr is used to point to processes in the dependent set */
    LIST *ptr;
    c_tcall *tcptr;
    DDMSG ddmsg;

    pid = table [i].pid; /* the process we are working on */
    ptr = table [i].next;
    ASSERT (table [i].pid.u_pid == c_procs[i].p_pid, "ddstart2:");
    dda_id = create ddagent(pid, c_procs [i].p_tcseq);
```

```
c_printf("deadlock: created ddagent %x\n", dda_id);
/*
 * make sure the list starts with zeroes
 */
for (j=0; j < GDLISTSIZE; ddmsg.dlist[j++].pid.u_pid = 0);
ddmsg.dda_id = dda_id;
ddmsg.msg_seqno = 1;
ddmsg.dlist [0].pid = table [i].pid;
ddmsg.dlist [0].seqno = c_procs[i].p_tcseq;
ddmsg.dlist [1].pid.u_pid = 0;
while (ptr != NULL) {
    /*
     * start at the beginning of the incoming transaction queue
     */
    tcptr = c_procs [pid.u_px].p_tcin;
    while (tcptr != NULL)
        /*
         * If a possible caller has called, then we don't
         * need to flow a message to it.  It is already in
         * the ACCEPTSTATE so we know the caller's
         * transaction couldn't be accepted (else the callee
         * would be running).
         */

        if (tcptr -> tc_callerpid == ptr -> pid.u_pid)
            break;
        else
            tcptr = tcptr -> tc_next;
    if (tcptr == NULL)
        /* didn't find it
         *
         * Note that we flow the first message, but have to split
         * for any others
         */
        if (firstone) {
            tmppid.u_pid = ptr -> pid.u_pid;
            c_printf(
            "deadlock: calling deadmen[%d] {=0x%x}.gd()\n",
                tmppid.u_por, deadmen [tmppid.u_por]);
            deadmen [tmppid.u_por].global_detect(ddmsg, ptr ->pid,
                FALSE);
            firstone = FALSE;
        }
        else {
            ddmsg.msg_seqno = ddmsg.dda_id.split (pid);
            if (ddmsg.msg_seqno != 0) {
                tmppid.u_pid = ptr -> pid.u_pid;
                c_printf(
                "deadlock: calling deadmen[%d] {=0x%x}.gd()\n",
                    tmppid.u_por, deadmen [tmppid.u_por]);
                deadmen [tmppid.u_por].global_detect
                    (ddmsg, ptr -> pid, FALSE);
            }
        }
        ptr = ptr -> next;
```

```
    }

}
void
propagate (ddmsg, newpid)
DDMSG ddmsg; c_pidu newpid;
{
    int last;
    c_pidu tmppid;
    LIST *ptr;
    c_tcall *tcptr;

    c_printf("propagate: ddmsg.a= %x, msgseq= %x, newpid= %x >>\n",
            ddmsg.dda_id, ddmsg.msg_seqno, newpid);
    c_prptp(&c_procs[newpid.u_px]);
    for (last = 0; (last < GDLISTSIZE) &&
                    (ddmsg.dlist[last].pid.u_pid != 0); last++);

    if WAITSTATE (newpid) {
        ddmsg.dlist [last].pid = newpid;
        ddmsg.dlist [last].seqno = c_procs [newpid.u_px].p_tcseq;
        last++;
        tmppid.u_pid = c_procs [newpid.u_px].p_tccallee;
        c_printf("deadlock: calling deadmen[%d] {=0x%x}.gd()\n",
            tmppid.u_por, deadmen [tmppid.u_por]);
        deadmen [tmppid.u_por].global_detect (ddmsg, tmppid, TRUE);
    }
    else {
        if ACCEPTSTATE (newpid) {
            ptr = table [newpid.u_px].next;
            while (ptr != NULL) {
                tcptr = c_procs [newpid.u_px].p_tcin;
                while (tcptr != NULL)
                    if (tcptr -> tc_callerpid == ptr -> pid.u_pid)
                        break;
                    else
                        tcptr = tcptr -> tc_next;
                if (tcptr == NULL) {
                    /* didn't find it */
                    ddmsg.msg_seqno = ddmsg.dda_id.split (newpid);
                    if (ddmsg.msg_seqno != 0) {
                        ddmsg.dlist [last].pid = newpid;
                        ddmsg.dlist [last].seqno =
                            c_procs[newpid.u_px].p_tcseq;
                        tmppid.u_pid = ptr -> pid.u_pid;
                        c_printf(
                    "deadlock: calling deadmen[%d] {=0x%x}.gd()\n",
                            tmppid.u_por, deadmen [tmppid.u_por]);
                        deadmen [tmppid.u_por].global_detect
                            (ddmsg, ptr -> pid, FALSE);
                    }
                }
                ptr = ptr -> next;
            }
        }
```

```
        else
            /* abort */
            ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
    }
}

void
ddetect(ddmsg, newpid, flowtcall)
DDMSG ddmsg; c_pidu newpid; char flowtcall;
{
    register int i;

    ASSERT (ISLOCAL (newpid), "ddetect:");
    c_printf("ddetect:\n");
    c_prptp(&c_procs[newpid.u_px]);
    /*
     * Is newpid a process-to-watch?  It might not be if the
     * message was flowed from a transacting process which was
     * making some kind of special call (ie.  to a statistics
     * collector or something) which does not really participate in
     * the communication which might deadlock.
     */

    if (table [newpid.u_px].pid.u_pid == 0) {
        c_printf("deadlock: [%x] not a process to watch\n",
            newpid.u_pid);
        /*
         * The process is calling someone we aren't watching, so we
         * must conclude that it is active; no deadlock
         */
        ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
    }
    if (WAITSTATE (newpid) || ACCEPTSTATE (newpid)) {

        if (flowtcall) {
            register int last;
            register c_tcall *tcptr;
            c_pidu tmppid;
            /*
             * this msg arrived by following a transaction call
             * our job is to verify that the call is still here
             * (necessary in distributed environment)
             */
            for (last = 0; (last < GDLISTSIZE) && (ddmsg.dlist
                [last].pid.u_pid != 0); last++);
            last--;
            tmppid.u_pid = ddmsg.dlist [last].pid.u_pid;
            tcptr = c_procs[newpid.u_px].p_tcin;
            while (tcptr != NULL)
                if (tcptr -> tc_callerpid == tmppid.u_pid)
                    break;
                else
                    tcptr = tcptr -> tc_next;
            if (tcptr == NULL)
                /*
```

```
                * didn't find it: abort
                */
               ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
           else
               /*
                * found it, now see if it is the same call by
                * comparing the sequence numbers
                */
               if (ddmsg.dlist [last].seqno != tcptr->tc_tcseq)
                   ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
       }
       /*
        * newpid is still blocked, now we want to know if this is the
        * process the message started at
        */
       if (ddmsg.dlist [0].pid.u_pid == newpid.u_pid)
           /*
            * yes, same process, are the sequence numbers the same?
            */
           if (ddmsg.dlist [0].seqno == c_procs[newpid.u_px].p_tcseq)
               /*
                * yes, report deadlock to agent
                */
               ddmsg.dda_id.rpt_deadlock (ddmsg);
           else
               /*
                * seqno has changed, abort
                */
               ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
       else {
           /* have we been here before? */
           for (i=1; i < GDLISTSIZE && ddmsg.dlist[i].pid.u_pid!=0;
                 i++)
               if (ddmsg.dlist[i].pid.u_pid == newpid.u_pid) {
                   /* Yes we've passed this node before... */
                   if (ddmsg.dlist[i].seqno ==
                           c_procs[newpid.u_px].p_tcseq)
                       /* the seqno is the same, report to dda */
                       ddmsg.dda_id.rpt_deadlock (ddmsg);
                   else
                       /* seqno has changed, abort computation */
                       ddmsg.dda_id.abort (newpid, ddmsg.msg_seqno);
                   return();
               }
           propagate (ddmsg, newpid);
       }
   }
}
void
printddmsg (mypid, ddmsg, newpid)
process anytype mypid;
DDMSG ddmsg;
c_pidu newpid;
{
    register int i;
```

```
    c_printf("\tdda_id= %x, msg_seq= %d\n",
            ddmsg.dda_id, ddmsg.msg_seqno);
    for (i=0; i < GDLISTSIZE && ddmsg.dlist [i].pid.u_pid != 0; i++)
        c_printf("\t\tdlist[%d]: pid= %x, seq= %d\n",
            i, ddmsg.dlist[i].pid, ddmsg.dlist [i].seqno);
}

/*
 * Application processes need to call bumpseq() prior to doing a
 * select or an accept (in the absence of a select).  This is to
 * note activity in the process.
 */
void
bump_seq()
{
    (c_cur -> p_tcseq) ++;
}
```

```
static char
rcsid[]= "$Header: mkgraph.cc,v 1.5 88/09/14 02:24:31 scott Locked $";
typedef long    c_pid, c_tp; /* defined in ccproc.h, need for pid */
#include <stdio.h>
#include "Pid.h"
#include "defs.h"
#include "spec.h"

#define MAXPROCS    100

#define add(gr, proc)                                           \
        {                                                       \
            LIST *ptr;                                          \
            ptr = gr -> next;                                   \
            while (ptr != 0 && ptr -> pid.u_px != proc)         \
                ptr = ptr -> next;                              \
            if (ptr == 0) {                                     \
                ptr = (LIST *) malloc (sizeof (LIST));          \
                ptr -> pid.u_pid = proc;                        \
                ptr -> next = gr -> next;                       \
                gr -> next = ptr;                               \
            }                                                   \
        }

process body buildgraph(deadmen, Query_tab)
{
    long proc;
    DEADMEN_S Deadmen;
    CALLERS_S Callers;
    CALLERS callers;
    LIST    *graph;
    LIST    *j, *k;
    int nprocs = 0, i, done = 0;
    TPTRS_putcallers    tptrs;

    graph = (LIST *) malloc (MAXPROCS * (sizeof (LIST)));

    for (i=0; i < D_NPORS; i++)
        if (deadmen [i] != 0)
            c_printf(
                "buildgraph: deadmen [%d]= %x\n", i, deadmen[i]);

    for (; !done; ) select {
        /*
         * load_callers is given a pid, and a table of pid's
         * calls is a table of pids which proc may call
         *
         * It then inverts this information, building a graph listing
         * each process and the process(es) which call(s) it.
         */
        accept load_callers (Proc, Calls) {
            proc = Proc;
            for (i = 0; i < CALLERSIZE; i++) callers [i] = Calls [i];
        } /* free caller */
```

```
        /*
         * If proc is not in the graph yet, put it in
         */
        for (j = graph; (j < graph + nprocs) &&
            (j -> pid.u_pid != proc); j++);
        if (j -> pid.u_pid != proc)
            if (nprocs == 0) {
                graph -> pid.u_pid = proc;
                graph -> next = NULL;
                nprocs++;
            }
            else {
                (graph + nprocs) -> pid.u_pid = proc;
                (graph + nprocs) -> next = NULL;
                nprocs++;
            }
        /*
         * now add proc to the list of callers to callers [i]
         */
        for (i = 0; (i < CALLERSIZE) && (callers[i] != 0); i++) {
            for (j = graph; j < graph + nprocs; j++) {
                /*
                 * if callers [i] is in graph then add proc and move
                 * to next i
                 */
                if (j -> pid.u_pid == callers[i]) {
                    add (j, proc);
                    break;
                }
            }
            /*
             * if end of graph then didn't find callers[i] so add both
             */
            if (j == graph + nprocs) {
                (graph + nprocs) -> pid.u_pid = callers [i];
                add ((graph + nprocs), proc);
                nprocs++;
            }
        }
        or
        /*
         * This function is to be called after all load_callers
         * transactions are finished.
         */
        accept done() {
            done = TRUE;
            printf("Buildgraph: received DONE\n");
            fflush (stdout);
        }
    }
    for (j = graph; j < graph + nprocs; j++) {
        /* send callers to process pid */
        k = j -> next;
        for (i = 0; i < CALLERSIZE; i++)
            Callers.callers [i] = 0;
```

```
        i = 0;
        while (k != 0) {
            Callers.callers [i++] = k -> pid.u_pid;
            k = k -> next;
            if (i == CALLERSIZE) {
                printf("build: deadmen [%d] == %#x\n", j->pid.u_por,
                    deadmen [j->pid.u_por]); fflush (stdout);
                deadmen [j->pid.u_por].putcallers (j -> pid, Callers);
                i = 0;
            }
        }
        printf("build: deadmen [%d] == %#x\n", j->pid.u_por,
            deadmen [j->pid.u_por]); fflush (stdout);
        Callers.callers [i] = 0; /* I think */
        deadmen [j->pid.u_por].putcallers (j -> pid, Callers);
    }
    for (i = 0; i < D_NPORS; i++)
        Deadmen.deadmen [i] = deadmen [i];
    printf("Buildgraph: calling putdeadguys\n"); fflush (stdout);
    for (i = 0; i < D_NPORS; i++) {
        if (Query_tab [i] != 0)
            Query_tab [i].putdeadguys (Deadmen);
        if (deadmen [i] != 0)
            deadmen [i].done();
    }
    printf("Buildgraph is finished\n"); fflush (stdout);

}
```

```
#ident "$Header: defs.h,v 1.5 88/09/14 02:23:20 scott Locked $"

  /* must be same as NPROCS in libmpcc: nprocs.c */
#define D_NPROCS    50

  /* must be same as NPORS in libmpcc: cc.h */
#define D_NPORS     16

#define CALLERSIZE  6
#define OUTBUFSIZE  64
#define GDLISTSIZE  50
#define TRUE    1
#define FALSE   0
#define DBG(x)  (d_debug & (1 << (x)))

struct L_LIST {
    c_pidu pid;
    L_LIST *next;
};
typedef struct L_LIST LIST;

struct Dead_ID {
    c_pidu pid;
    long    seqno;
};
typedef struct Dead_ID DEADID;
typedef struct Dead_ID GDLIST [GDLISTSIZE];

struct ddmsg_t {
    process ddagent dda_id;
    long msg_seqno;
    DEADID dlist [GDLISTSIZE];
};
typedef struct ddmsg_t DDMSG;

typedef long CALLERS [CALLERSIZE];
typedef struct {
            CALLERS callers;
        } CALLERS_S;

typedef char OUTBUF [OUTBUFSIZE];
typedef struct {
            OUTBUF outbuf;
        } OUTBUF_S;

typedef trans   void (*TPTR_putcallers) (c_pidu, CALLERS_S);
typedef TPTR_putcallers TPTRS_putcallers [D_NPORS];

typedef process deadlock DEADMEN [D_NPORS];
typedef struct {
            DEADMEN deadmen;
        } DEADMEN_S;

typedef process queryserv Queryserv_tab [D_NPORS];
```

```
typedef struct {
        Queryserv_tab query_tab;
    } Queryserv_tab_S;

typedef async trans (*TPTR_intercept_response) (OUTBUF, int);
typedef int trans (*TPTR_report_deadlock)(c_pidu);

extern int c_por;

#define ISLOCAL(pid)    (pid.u_por == c_por)


#define ACCEPTSTATE(pid) (c_procs[pid.u_px].p_state == c_wselect)
#define WAITSTATE(PID) (c_procs [PID.u_px].p_state == c_wservice || \
        (c_procs [PID.u_px].p_state == c_wmsg &&                    \
        c_procs [PID.u_px].p_tccallee != 0))

#define ASSERT(X,Y) if (!(X)) \
    fprintf(stderr,"%s Assertion failed\n",Y)
```

```
#ident "$Header: spec.h,v 1.6 88/09/13 03:38:22 scott Locked $"

process spec buildgraph(DEADMEN deadmen, Queryserv_tab Query_tab)
{   void trans load_callers (long Proc, CALLERS Callers);
    void trans done();
};


process spec queryserv () {
    void trans putdeadguys (DEADMEN_S Deadmen);
    async trans intercept (c_pidu src_pid, c_pidu dst_pid,
        TPTR_intercept_response response_tptr);
    long trans getstate (c_pidu pid);
};

process spec deadlock (TPTR_report_deadlock report_deadlock,
                    Queryserv_tab_S Query_tab, long D_debug)
{
    trans void putcallers (c_pidu Proc, CALLERS_S Callers);
    async trans intercept (c_pidu src_pid, c_pidu xptr,
        TPTR_intercept_response response_tptr);
    void trans done();
    long trans getstate (c_pidu pid);
    async trans global_detect (DDMSG Ddmsg, c_pidu Newpid,
        char Flowtcall);
    async trans term();
    void trans enable_detect();
};

process spec resolver () {
    async trans intercept_response (OUTBUF_S outbuf, int status);
    int trans report (c_pidu pid);
};

process spec ddagent (c_pidu origin, long seqno) {
    int trans split (c_pidu Pid);
    void trans abort (c_pidu Pid, long Seqno);
    void trans rpt_deadlock (DDMSG DDmsg);
};
```

Overview:

Startup:

First a queryserv process should be started on each processor, saving the pid in a table of type Queryserv_tab. Next a deadlock process should be started on each processor, passing it a transaction pointer to call when when reporting deadlock, and the Queryserv table. Then on the local (main) processor (0) the process "buildgraph" should be started. Buildgraph should be called repeatedly giving process/callee information, and finally a table of process ids of the deadlock processes started originally.

```
    /* must be same as NPROCS in libmpcc: nprocs.c */
#define D_NPROCS    24
    /* must be same as NPORS in libmpcc: cc.h */
#define D_NPORS        16

#define CALLERSIZE    6
#define OUTBUFSIZE    64
#define TRUE    1
#define FALSE    0

struct L_LIST {
    c_pidu pid;
    L_LIST *next;
};
typedef struct L_LIST LIST;

typedef long CALLERS [CALLERSIZE];
typedef char OUTBUF [OUTBUFSIZE];

typedef trans    void (*TPTR_putcallers) (c_pidu, CALLERS);
typedef TPTR_putcallers    TPTRS_putcallers [D_NPORS];

typedef process deadlock DEADMEN [D_NPORS];
typedef process queryserv Queryserv_tab [D_NPORS];
typedef async trans (*TPTR_intercept_response) (OUTBUF, int);
typedef async trans (*TPTR_report_deadlock)();

extern int c_por;

#define ISLOCAL(pid)    (pid.u_por == c_por)
```

```
process spec buildgraph(DEADMEN deadmen, Queryserv_tab Query_tab)
{    void trans load_callers (long proc, CALLERS callers);
    void trans done();
};
```

Startup

Above is the process spec for buildgraph(). Buildgraph is called with a table of process ids of deadlock processes, indexed by processor number. Someone will call buildgraph with the load_callers transaction call, once for each relevant process (a relevant process is a process which may participate in deadlock). "proc" will be the pid of the process in consideration, and CALLERS is a table of processes (pids) which this process can call.

"loadcallers()" may be called as many times as necessary for a single process. If ever there are fewer processes than will fill the callers table, place a 0 pid in the first invalid location. If a process calls no one, it is *not* necessary to call loadcallers() (at least not necessary at the moment).

When the process calling buildgraph() with all this juicy information is finished, it needs to make one last transaction call "done()". This tells buildgraph() to distribute its version of the graph to all of the deadlock processes.

```
process spec deadlock (TPTR_report_deadlock report_deadlock,
            Queryserv_tab Query_tab)
{
    trans void putcallers (c_pidu Proc, CALLERS Callers);
    void trans done();
    async trans intercept (c_pidu src_pid, c_pidu xptr,
            TPTR_intercept_response response_tptr);
    long trans getstate (c_pidu pid);
}
```

Querying the network state. The resolver may contact the local deadhead using the transaction calls "intercept" and "querystate". Here is the spec:

```
async trans intercept (c_pidu src_pid, c_pidu xptr,
        TPTR_intercept_response response_tptr);
trans long getstate (c_pidu pid);
```

The purpose of intercept is to snag the parms of a pending transaction call, given the caller and the callee processes. The intercept transaction is given the pid of the process making the xaction call (src_pid), the pid of the accepting process (dst_pid), and a transaction call to send the response to. Due to the problems of implementing the return of arbitrary information in a distributed environment, a call back mechanism (response) is used to return the intercept result. Hence the resolver should do "deadman.intercept (x, y, tptr); accept response (outbuf, status);" or something like that. Just don't forget that intercept is an async call. The intercepted message (or as much of it as would fit) will be in "outbuf", and a "status" value indicating the relative success of the intercept.

If it went ok (found the message) then status will be 0, if it couldn't find the message then the status will be -1.

Getstate is a synchronous transaction call and returns the state of the process in question. There are three possible states: ACCEPTING, XACTING, and RUNNING, and they occupy bits 27-31 of the return value.

```
#define ACCEPTING (1 << 27)
#define XACTING (2 << 27)
#define RUNNING    (3 << 27)
```

ACCEPTING means that the process is doing an accept, probably just waiting on a transaction call. XACTING means that the process is making a transaction call. In this case the process id of the transactee will be returned in the first 27 bits (0-26) of the long. RUNNING is pretty much everything else.

Distributed Deadlock Detection in Concurrent C

by

Scott William Hammond

B. S., Kansas State University, 1986

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Abstract

One often finds that extremely useful tools or applications can be abstracted into language constructs whose methods can be hidden from the user. Interprocess communication is a good example of this as shown in the Concurrent C transaction call [GeRo86], and the Ada rendezvous [DoD83]. Because programmers now have the ability to write communicating programs easily, it would be nice if the programmer didn't have to worry about the details of deadlock detection. We believe deadlock detection is appropriate for implementation at the language level.

In this thesis we describe an implementation of a distributed deadlock detection algorithm in the distributed kernel of Concurrent C. In particular, we have considered the relationship between local and global deadlock detection and made some special improvements on published deadlock detection algorithms [CMH83, Nat86] that can be made because we have special implementation (kernel) knowledge. Our specific implementation is in support of a distributed discrete event simulator [Vop88] for which we detect deadlock.