happen immediately. In Fig. 3-22(f), some interloper has gotten in there and is trying to write the file. The interloper's timestamp is lower than beta's, so beta simply waits until the interloper commits, at which time it can read the new file and continue.

In Fig. 3-22(g), gamma has changed the file and already committed. Again beta must abort. In Fig. 3-22(h), gamma is in the process of changing the file, although it has not committed yet. Still, beta is too late and must abort.

Timestamping has different properties than locking. When a transaction encounters a larger (later) timestamp, it aborts, whereas under the same circumstances with locking it would either wait or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus.

All in all, transactions offer many advantages and thus are a promising technique for building reliable distributed systems. Their chief problem is their great implementation complexity, which yields low performance. These problems are being worked on, and perhaps in due course they will be solved.

3.5. DEADLOCKS IN DISTRIBUTED SYSTEMS

Deadlocks in distributed systems are similar to deadlocks in single-processor systems, only worse. They are harder to avoid, prevent, or even detect, and harder to cure when tracked down because all the relevant information is scattered over many machines. In some systems, such as distributed data base systems, they can be extremely serious, so it is important to understand how they differ from ordinary deadlocks and what can be done about them.

Some people make a distinction between two kinds of distributed deadlocks: communication deadlocks and resource deadlocks. A communication deadlock occurs, for example, when process A is trying to send a message to process B, which in turn is trying to send one to process C, which is trying to send one to A. There are various scenarios in which this situation leads to deadlock, such as no buffers being available. A resource deadlock occurs when processes are fighting over exclusive access to I/O devices, files, locks, or other resources.

We will not make that distinction here, since communication channels, buffers, and so on, are also resources and can be modeled as resource deadlocks because processes can request them and release them. Furthermore, circular communication patterns of the type just described are quite rare in most systems. In client-server systems, for example, a client might send a message (or perform an RPC) with a file server, which might send a message to a disk server. However, it is unlikely that the disk server, acting as a client, would send a message to the original client, expecting it to act like a server. Thus the circular wait condition is unlikely to occur as a result of communication alone.

Various strategies are used to handle deadlocks. Four of the best-known ones are listed and discussed below.

1. The ostrich algorithm (ignore the problem).
2. Detection (let deadlocks occur, detect them, and try to recover).
3. Prevention (statically make deadlocks structurally impossible).
4. Avoidance (avoid deadlocks by allocating resources carefully).

All four are potentially applicable to distributed systems. The ostrich algorithm is as good and as popular in distributed systems as it is in single-processor systems. In distributed systems used for programming, office automation, process control, and many other applications, no system-wide deadlock mechanism is present, although individual applications, such as distributed data bases, can implement their own if they need one.

Deadlock detection and recovery is also popular, primarily because prevention and avoidance are so difficult. We will discuss several algorithms for deadlock detection below.

Deadlock prevention is also possible, although more difficult than in single-processor systems. However, in the presence of atomic transactions, some new options become available. Two algorithms are discussed below.

Finally, deadlock avoidance is never used in distributed systems. It is not even used in single-processor systems, so why should it be used in the more difficult case of distributed systems? The problem is that the banker's algorithm and similar algorithms need to know (in advance) how much of each resource every process will eventually need. This information is rarely, if ever, available. Thus our discussion of deadlocks in distributed systems will focus on just two of the techniques: deadlock detection and deadlock prevention.

3.5.1. Distributed Deadlock Detection

Finding general methods for preventing or avoiding distributed deadlocks appears to be quite difficult, so many researchers have tried to deal with the simpler problem of just detecting deadlocks, rather than trying to inhibit their occurrence.

However, the presence of atomic transactions in some distributed systems makes a major conceptual difference. When a deadlock is detected in a conventional operating system, the way to resolve it is to kill off one or more processes. Doing so invariably leads to one or more unhappy users. When a deadlock is detected in a system based on atomic transactions, it is resolved by aborting one or more transactions. But as we have seen in detail above, transactions have been designed to withstand being aborted. When a transaction is aborted

because it contributes to a deadlock, the system is first restored to the state it had before the transaction began, at which point the transaction can start again. With a little bit of luck, it will succeed the second time. Thus the difference is that the consequences of killing off a process are much less severe when transactions are used than when they are not used.

Centralized Deadlock Detection

As a first attempt, we can use a centralized deadlock detection algorithm and try to imitate the nondistributed algorithm. Although each machine maintains the resource graph for its own processes and resources, a central coordinator maintains the resource graph for the entire system (the union of all the individual graphs). When the coordinator detects a cycle, it kills off one process to break the deadlock.

Unlike the centralized case, where all the information is automatically available in the right place, in a distributed system it has to be sent there explicitly. Each machine maintains the graph for its own processes and resources. Several possibilities exist for getting it there. First, whenever an arc is added or deleted from the resource graph, a message can be sent to the coordinator providing the update. Second, periodically, every process can send a list of arcs added or deleted since the previous update. This method requires fewer messages than the first one. Third, the coordinator can ask for information when it needs it.

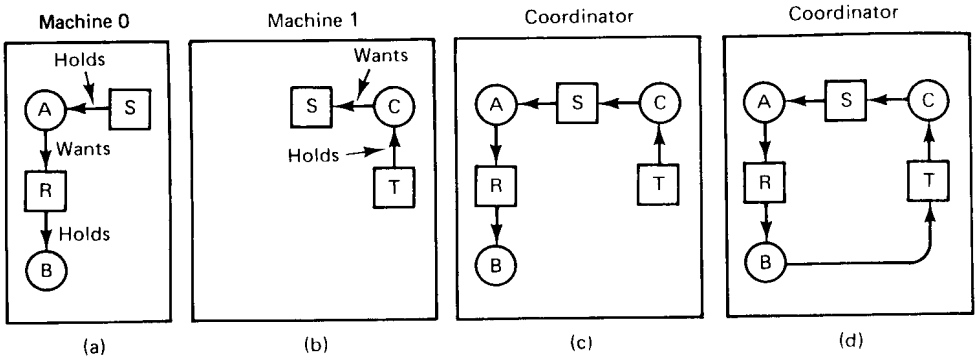

Fig. 3-23. (a) Initial resource graph for machine 0. (b) Initial resource graph for machine 1. (c) The coordinator's view of the world. (d) The situation after the delayed message.

Unfortunately, none of these methods work well. Consider a system with processes *A* and *B* running on machine 0, and process *C* running on machine 1. Three resources exist: *R*, *S*, and *T*. Initially, the situation is as shown in Fig. 3-23(a) and (b): *A* holds *S* but wants *R*, which it cannot have because *B* is using it;

C has *T* and wants *S*, too. The coordinator's view of the world is shown in Fig. 3-23(c). This configuration is safe. As soon as *B* finishes, *A* can get *R* and finish, releasing *S* for *C*.

After a while, *B* releases *R* and asks for *T*, a perfectly legal and safe swap. Machine 0 sends a message to the coordinator announcing the release of *R*, and machine 1 sends a message to the coordinator announcing the fact that *B* is now waiting for its resource, *T*. Unfortunately, the message from machine 1 arrives first, leading the coordinator to construct the graph of Fig. 3-23(d). The coordinator incorrectly concludes that a deadlock exists and kills some process. Such a situation is called a **false deadlock**. Many deadlock algorithms in distributed systems produce false deadlocks like this due to incomplete or delayed information.

One possible way out might be to use Lamport's algorithm to provide global time. Since the message from machine 1 to the coordinator is triggered by the request from machine 0, the message from machine 1 to the coordinator will indeed have a later timestamp than the message from machine 0 to the coordinator. When the coordinator gets the message from machine 1 that leads it to suspect deadlock, it could send a message to every machine in the system saying: "I just received a message with timestamp *T* which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately." When every machine has replied, positively or negatively, the coordinator will see that the arc from *R* to *B* has vanished, so the system is still safe. Although this method eliminates the false deadlock, it requires global time and is expensive. Furthermore, other situations exist where eliminating false deadlock is much harder.

Distributed Deadlock Detection

Many distributed deadlock detection algorithms have been published. Surveys of the subject are given in Knapp (1987) and Singhal (1989). Let us examine a typical one here, the Chandy-Misra-Haas algorithm (Chandy et al., 1983). In this algorithm, processes are allowed to request multiple resources (e.g., locks) at once, instead of one at a time. By allowing multiple requests simultaneously, the growing phase of a transaction can be speeded up considerably. The consequence of this change to the model is that a process may now wait on two or more resources simultaneously.

In Fig. 3-24, we present a modified resource graph, where only the processes are shown. Each arc passes through a resource, as usual, but for simplicity the resources have been omitted from the figure. Notice that process 3 on machine 1 is waiting for two resources, one held by process 4 and one held by process 5.

Some of the processes are waiting for local resources, such as process 1, but

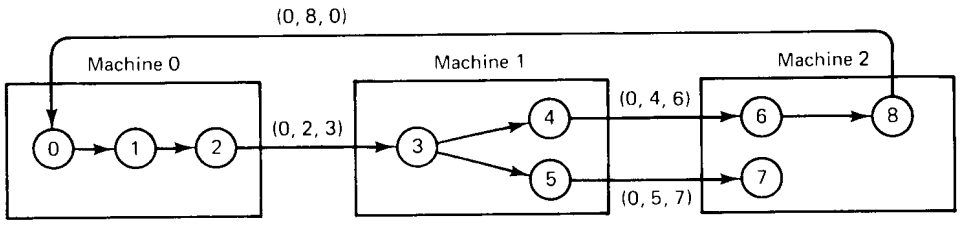

Fig. 3-24. The Chandy-Misra-Haas distributed deadlock detection algorithm.

others, such are process 2, are waiting for resources that are located on a different machine. It is precisely these cross-machine arcs that make looking for cycles difficult. The Chandy-Misra-Haas algorithm is invoked when a process has to wait for some resource, for example, process 0 blocking on process 1. At that point a special **probe** message is generated and sent to the process (or processes) holding the needed resources. The message consists of three numbers: the process that just blocked, the process sending the message, and the process to whom it is being sent. The initial message from 0 to 1 contains the triple (0, 0, 1).

When the message arrives, the recipient checks to see if it itself is waiting for any processes. If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for. The message is then sent to the process on which it is blocked. If it is blocked on multiple processes, all of them are sent (different) messages. This algorithm is followed whether the resource is local or remote. In Fig. 3-24 we see the remote messages labeled (0, 2, 3), (0, 4, 6), (0, 5, 7), and (0, 8, 0). If a message goes all the way around and comes back to the original sender, that is, the process listed in the first field, a cycle exists and the system is deadlocked.

There are various ways in which the deadlock can be broken. One way is to have the process that initiated the probe commit suicide. However, this method has problems if several processes invoke the algorithm simultaneously. In Fig. 3-24, for example, imagine that both 0 and 6 block at the same moment, and both initiate probes. Each would eventually discover the deadlock, and each would kill itself. This is overkill. Getting rid of one of them is enough.

An alternative algorithm is to have each process add its identity to the end of the probe message so that when it returned to the initial sender, the complete cycle would be listed. The sender can then see which process has the highest number, and kill that one or send it a message asking it to kill itself. Either way, if multiple processes discover the same cycle at the same time, they will all choose the same victim.

There are few areas of computer science in which theory and practice

diverge as much as in distributed deadlock detection algorithms. Discovering yet another deadlock detection algorithm is the goal of many a researcher. Unfortunately, these models often have little relation to reality. For example, some of the algorithms require processes to send probes when they are blocked. However, sending a probe when you are blocked is not entirely trivial.

Many of the papers contain elaborate analyses of the performance of the new algorithm, pointing out, for example, that while the new one requires two traversals of the cycle, it uses shorter messages, as if these factors balanced out somehow. The authors would no doubt be surprised to learn that a typical "short" message (20 bytes) on a LAN takes about 1 msec, and a typical "long" message (100 bytes) on the same LAN takes perhaps 1.1 msec. It would also no doubt come as a shock to these people to realize that experimental measurements have shown that 90 percent of all deadlock cycles involve exactly two processes (Gray et al., 1981).

Worst of all, a large fraction of all the published algorithms in this area are just plain wrong, including those proven to be correct. Knapp (1987) and Singhal (1989) point out some examples. It often occurs that shortly after an algorithm is invented, proven correct, and then published, somebody finds a counterexample. Thus we have an active research area in which the model of the problem does not correspond well to reality, the solutions found are generally impractical, the performance analyses given are meaningless, and the proven results are frequently incorrect. To end on a positive note, this is an area that offers great opportunities for improvement.

3.5.2. Distributed Deadlock Prevention

Deadlock prevention consists of carefully designing the system so that deadlocks are structurally impossible. Various techniques include allowing processes to hold only one resource at a time, requiring processes to request all their resources initially, and making processes release all resources when asking for a new one. All of these are cumbersome in practice. A method that sometimes works is to order all the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.

However, in a distributed system with global time and atomic transactions, two other practical algorithms are possible. Both are based on the idea of assigning each transaction a global timestamp at the moment it starts. As in many timestamp-based algorithms, in these two it is essential that no two transactions are ever assigned exactly the same timestamp. As we have seen, Lamport's algorithm guarantees uniqueness (effectively by using process numbers to break ties).

The idea behind the algorithm is that when one process is about to block

waiting for a resource that another process is using, a check is made to see which has a larger timestamp (i.e., is younger). We can then allow the wait only if the waiting process has a lower timestamp (is older) than the process waited for. In this manner, following any chain of waiting processes, the timestamps always increase, so cycles are impossible. Alternatively, we can allow processes to wait only if the waiting process has a higher timestamp (is younger) than the process waited for, in which case the timestamps decrease along the chain.

Although both methods prevent deadlocks, it is wiser to give priority to older processes. They have run longer, so the system has a larger investment in them, and they are likely to hold more resources. Also, a young process that is killed off will eventually age until it is the oldest one in the system, so this choice eliminates starvation. As we have pointed out before, killing a transaction is relatively harmless, since by definition it can be restarted safely later.

To make this algorithm clearer, consider the situation of Fig. 3-25. In (a), an old process wants a resource held by a young process. In (b), a young process wants a resource held by an old process. In one case we should allow the process to wait; in the other we should kill it. Suppose that we label (a) *dies* and (b) *wait*. Then we are killing off an old process trying to use a resource held by a young process, which is inefficient. Thus we must label it the other way, as shown in the figure. Under these conditions, the arrows always point in the direction of increasing transaction numbers, making cycles impossible. This algorithm is called **wait-die**.

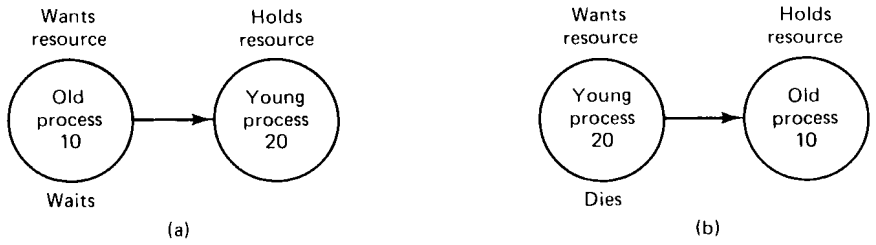

Fig. 3-25. The wait-die deadlock prevention algorithm.

Once we are assuming the existence of transactions, we can do something that had previously been forbidden: take resources away from running processes. In effect we are saying that when a conflict arises, instead of killing the process making the request, we can kill the resource owner. Without transactions, killing a process might have severe consequences, since the process might have modified files, for example. With transactions, these effects will vanish magically when the transaction dies.

Now consider the situation of Fig. 3-26, where we are going to allow

preemption. Given that our system believes in ancestor worship, as we discussed above, we do not want a young whippersnapper preempting a venerable old sage, so Fig. 3-26(a) and not Fig. 3-26(b) is labeled *preempt*. We can now safely label Fig. 3-26(b) *wait*. This algorithm is known as **wound-wait**, because one transaction is supposedly wounded (it is actually killed) and the other waits. It is unlikely that this algorithm will make it to the Nomenclature Hall of Fame.

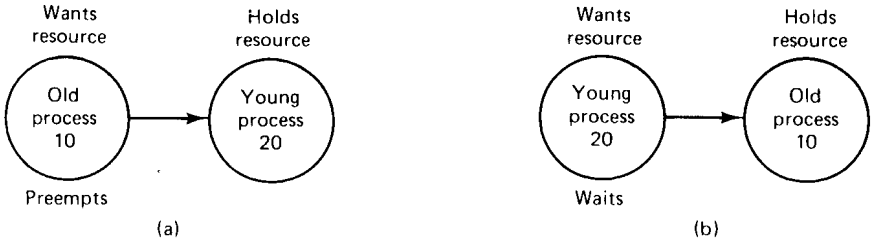

Fig. 3-26. The wound-wait deadlock prevention algorithm.

If an old process wants a resource held by a young one, the old process preempts the young one, whose transaction is then killed, as depicted in Fig. 3-26(a). The young one probably starts up again immediately, and tries to acquire the resource, leading to Fig. 3-26(b), forcing it to wait. Contrast this algorithm with wait-die. There, if an oldtimer wants a resource held by a young squirt, the oldtimer waits politely. However, if the young one wants a resource held by the old one, the young one is killed. It will undoubtedly start up again and be killed again. This cycle may go on many times before the old one releases the resource. Wound-wait does not have this nasty property.

3.6. SUMMARY

This chapter is about synchronization in distributed systems. We started out by giving Lamport's algorithm for synchronizing clocks without reference to external time sources, and later saw how useful this algorithm is. We also saw how physical clocks can be used for synchronization when real time is important.

Next we looked at mutual exclusion in distributed systems and studied three algorithms. The centralized algorithm kept all the information at a single site. The distributed algorithm ran the computation at all sites in parallel. The token ring algorithm passed control around the ring. Each has its strengths and weaknesses.

Many distributed algorithms require a coordinator, so we looked at two ways of electing a coordinator, the bully algorithm and another ring algorithm.