3.2. MUTUAL EXCLUSION

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems. For a taxonomy and bibliography of other methods, see (Raynal, 1991). Other work is discussed in (Agrawal and El Abbadi, 1991; Chandy et al., 1983; and Sanders, 1987).

3.2.1. A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig. 3-8(a). When the reply arrives, the requesting process enters the critical region.

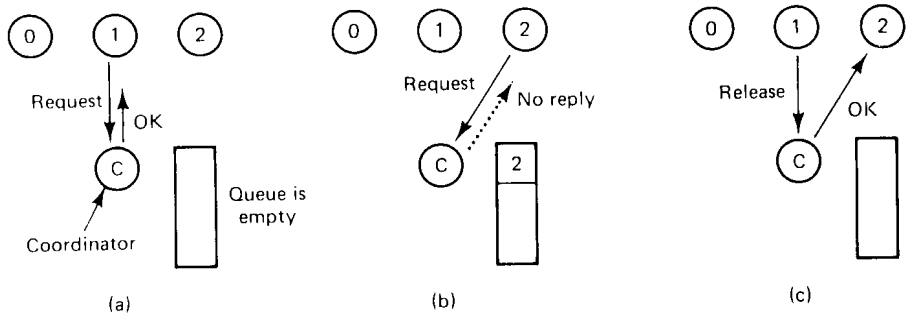

Fig. 3-8. (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Now suppose that another process, 2 in Fig. 3-8(b), asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig. 3-8(b), the coordinator just

refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. 3-8(c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

3.2.2. A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method.

Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one happened first. Lamport's algorithm presented in Sec. 3.1.1 is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 3-9(a).

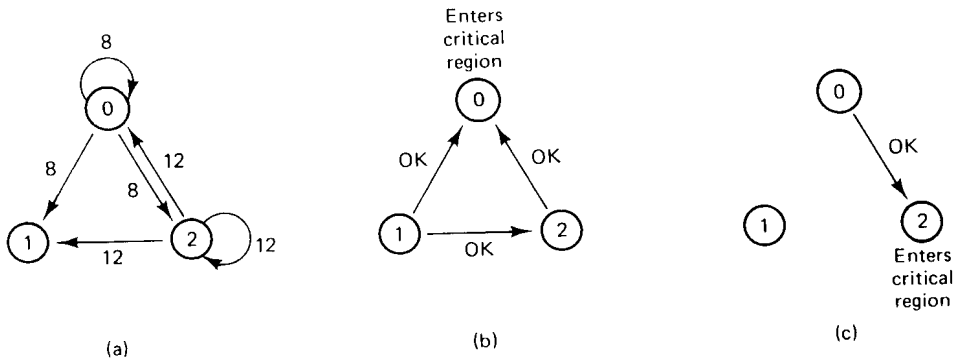

Fig. 3-9. (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not

interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 3-9(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig. 3-9(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig. 3-9 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now $2(n - 1)$, where the total number of processes in the system is n. Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is n times worse and requires much more network traffic to boot.

The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, *all* processes are involved in *all* decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example,

getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible (e.g., Maekawa et al., 1987).

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust that the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

3.2.3. A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. 3-10. Here we have a bus network, as shown in Fig. 3-10(a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig. 3-10(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a **token**. The token circulates around the ring. It is passed from process k to process $k+1$ (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to

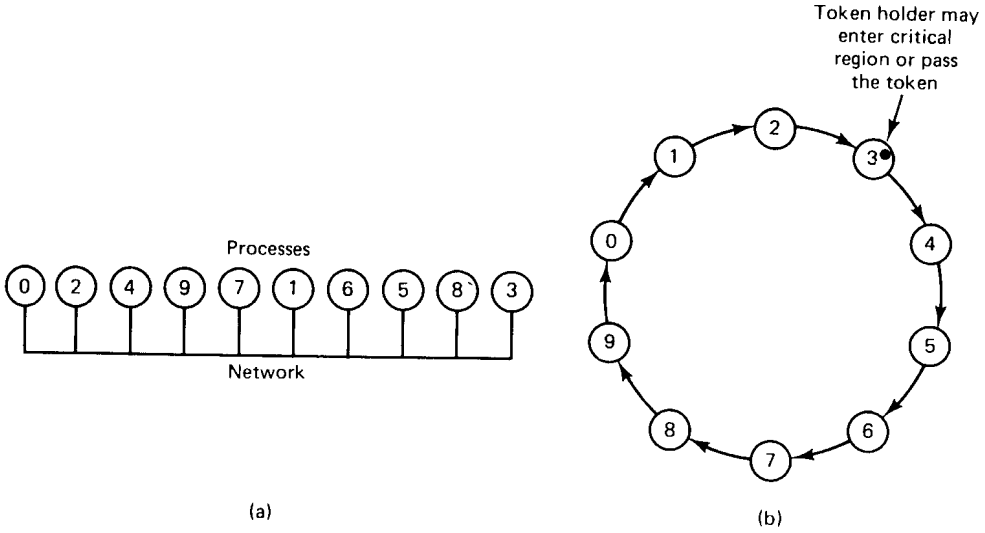

Fig. 3-10. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

wait for every other process to enter and leave one critical region.

As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

3.2.4. A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In Fig. 3-11 we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a LAN), and some problems associated with each algorithm.

The centralized algorithm is simplest and also most efficient. It requires

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Fig. 3-11. A comparison of three mutual exclusion algorithms.

only three messages to enter and leave a critical region: a request and a grant to enter, and a release to exit. The distributed algorithm requires $n - 1$ request messages, one to each of the other processes, and an additional $n - 1$ grant messages, for a total of $2(n - 1)$. With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. 3-11 we show the former case. It takes only two message times to enter a critical region in the centralized case, but $2(n - 1)$ message times in the distributed case, assuming that the network can handle only one message at a time. For the token ring, the time varies from 0 (token just arrived) to $n - 1$ (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is slightly ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they are all acceptable.

3.3. ELECTION ALGORITHMS

Many distributed algorithms require one process to act as coordinator, initiator, sequencer, or otherwise perform some special role. We have already seen several examples, such as the coordinator in the centralized mutual exclusion

algorithm. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process).

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be. Various algorithms are known, for example, (Fredrickson and Lynch, 1987; Garcia-Molina, 1982; and Singh and Kurose, 1994).

3.3.1. The Bully Algorithm

As a first example, consider the **bully algorithm** devised by Garcia-Molina (1982). When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, *P*, holds an election as follows:

1. *P* sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

At any moment, a process can get an *ELECTION* message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an *OK* message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Fig. 3-12 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends *ELECTION* messages to all the processes higher than it, namely 5, 6, and 7, as shown in Fig. 3-12(a). Processes 5 and 6 both respond with *OK*, as shown in Fig. 3-12(b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

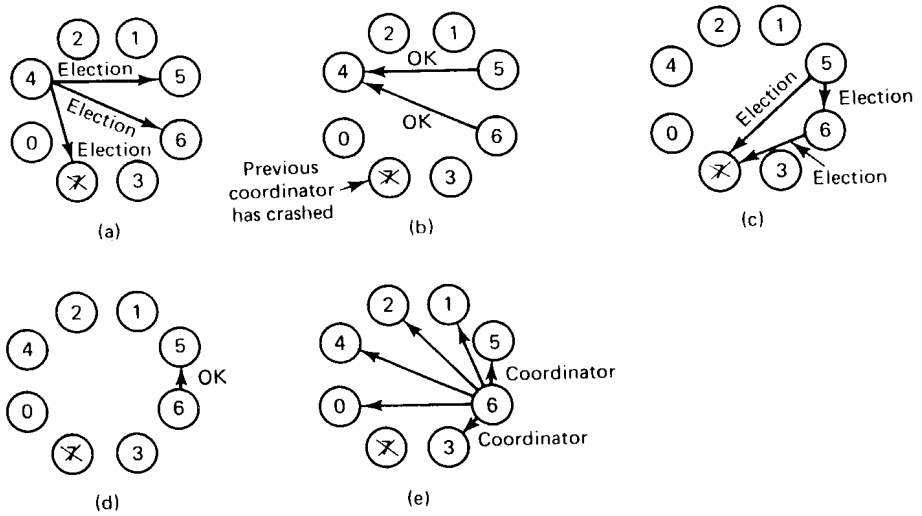

Fig. 3-12. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

In Fig. 3-13(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. 3-13(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a *COORDINATOR* message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send all the others a *COORDINATOR* message and bully them into submission.

3.3.2. A Ring Algorithm

Another election algorithm is based on the use of a ring, but without a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an *ELECTION* message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to *COORDINATOR* and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

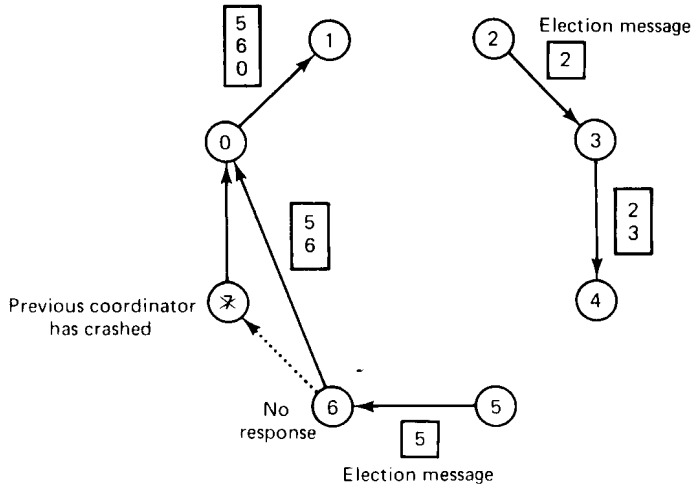

Fig. 3-13. Election algorithm using a ring.

In Fig. 3-13 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an *ELECTION* message and starts circulating it. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into *COORDINATOR* messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at most it wastes a little bandwidth.