

Solve the dining Philosophers problem with distributed algorithm

Solution: A process must be used to represent each dining philosopher and there should be total 5 processes .The duration of thinking and eating phases is to be randomly slected at runtime by each process .Your code must display messages that correspond to the start and stop of each of these phases. These phases must not interfere with communication responsibilities of process related to distributed algorithm . For example the reply to a message must not be delayed because the philosopher is eating or thinking .In addition the completion of a phase must not be delayed because I/O is blocked .

Your solution must be deadlock free.

Shared variables: The shared variables are to be accessed in some manner that could be supported in a distributed environment.

Output: your output must clearly demonstrate that algorithm is effective. Demonstrate an output where more than one philosopher is able to eat at one time.

Error checking: Provide error checking for each system call and take appropreiate steps if an error is encountered

Communications: You mustr use internet type sockets.

Suggestion: you might want to use non-blocking I/O and/or select

Distributed algorithm:

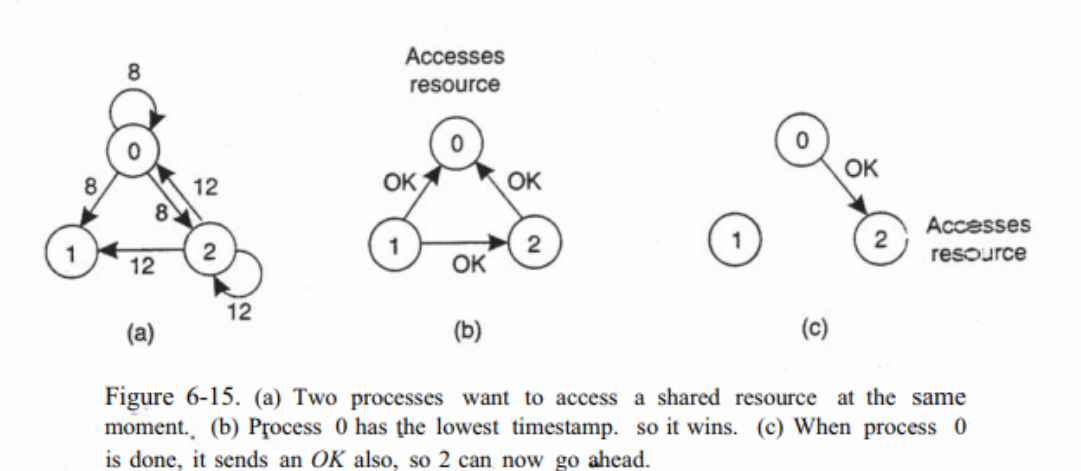
To many, having a probabilistic ally correct algorithm is just not good enough. So researchers have looked for deterministic distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section 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 actually happened first. Lamport's algorithm presented in Sec. 6.2.1 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 access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, no message is lost. When a process receives a request message from another process, the action it takes depends on its own state with respect to the resource named in the message. Three different cases have to be clearly distinguished:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.

2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.

3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with me. one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, 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, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may go ahead. When it is finished, 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 simultaneously access the resource, as shown in Fig. 6-15(a.



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 the resource, 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 access the resource, as shown in Fig. 6-15(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 go ahead, as shown in Fig. 6-15(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. 6-15 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 had already access to the resource 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 at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic as well. 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 multicast 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 accessing the shared resource. 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 is really overkill. All that is needed is a method to prevent two processes from accessing the resource at the same time. The algorithm can be modified to grant permission 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, it cannot grant the same permission to another process until the first one has finished. 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. It may take some time to discover exactly what