Mutual Exclusion (contd.)

We now prove correctness of algorithm (introduced last day). Source: E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. Communications of the ACM, 8(9), page 569, September, 1965.

Our algorithm has three shared variables:

1. k: Identity of the next process that will enter the critical section.
2. $b[1..N]$: Boolean array; $b[i]$ is false if and only if process i is trying to gain access to the critical region or is in the critical section.
3. $c[1..N]$: Boolean array; $c[i]$ is false if and only if process i has proceeded beyond the *phase 1* (roughly speaking).

The algorithm is given below:

```
AlgorithmMutex()
begin
   b[i] ← false
   done ← false
   loop
      exit when done

      ▷ Phase 1 begins here
      loop
         exit when k = i
         c[i] ← true
         if b[k] then k ← i
      endloop

      ▷ Phase 1 ends and phase 2 begins
      c[i] ← false
      done ← true
      for j from 1 to N, j ≠ i
         if c[j] = false then
            done ← false
         endif
      endfor
      ▷ Phase 2 ends
   endloop

   ▷ Now we are in the critical region.  Access resource
         ⋮
   ▷ Get out of the critical region
   c[i] ← true
   b[i] ← true
end ■
```

Proof of Mutual Exclusion

We need to show that no two processes could enter the critical region at the same time. We do this by contradiction; let us say that two processes, namely i and j are in the critical region at the same time.

For this to happen the main loop must have been executed at least once for each process. Thus both $c[i]$ and $c[j]$ must have been set to false in the last iteration of i and j. Suppose $c[j]$ was set false after $c[i]$ had already been set false. Then j would have found this out in the phase 2 `for` loop and could not have exited the main loop; hence we have a contradiction.

Progress Property

This property states that if some processes want access to the critical region, then *some* process does eventually get access. Please note that this property does not preclude starvation on the part of some particular process; we would have to modify the algorithm given here to make it fair.

Now we prove the progress property. Assume that some processes are in the main loop trying to gain access to the critical region. We have to show that k will eventually point to one of the processes trying to gain access to the critical section.

Note that $b[i]$ of those processes i not trying to gain access is `true`. The process currently in the critical region will also eventually get out setting its own $b[i]$ to `true`. Thus, if k does not already point to one of the processes trying to enter the critical section, eventually some of the processes will see that $b[k]$ is true. Those processes will then be allowed to update k.

Once k has changed for the first time, no process will ever see $b[k] = true$ again until some process has entered the critical section. All the processes that saw $b[k] = true$ will eventually finish updating k and, after that, k will never change again until some process enters the critical section.

Let's consider what happens once the value of k stops changing. If more than one process are in phase 2, they all proceed back to phase 1 (or one might get into the critical section). Later, only process k can get back into the phase 2. Process k will eventually be the only process in phase 2 and will then gain access to the critical section.

Models of Distributed Computing

We characterize distributed systems by several different characteristics.

1. Synchronous, Asynchronous and Semi-Synchronous

A distributed system could be one of the following:

1. **Synchronous.** The various systems run in a lockstep manner.

2. **Asynchronous.** Each system runs at its own speed.

3. **Semi-synchronous.** The various systems run at their own speeds but some bounds are placed on the speed of the systems.

Asynchronous behaviour can arise due to a variety of factors:

1. Cache misses cause unexpected delays.

2. Different computers could be running at different speeds.

3. Propagation delays in Wide Area Networks (WANs).

2. Communication Media

The communication media connecting the processes can have any of the following types.

1. Point-to-point communications.
2. Broadcast communications.
3. Communication through shared memory. Shared memory consists of a collection of data structures, called objects, that each process can access.
4. Remote object activation or remote procedure calls.

3. Failure Model

Distributed systems can be studied under different failure models. The system should continue to work even if some failures occur. The various components failing could be of the following types:

1. Process failures.
2. Channel failures.
3. Object failures.

Process failures themselves could be of the following types:

1. **Transient Failures.** In this case the system fails for some time and then recovers.
2. **Halting Failures.** In this case the system completely dies and never comes back.
3. **Arbitrary (or Byzantine) Failures.** In this case the system can start doing arbitrary things and may even act maliciously.

The channel failures can be of the following types:

1. Packets can get dropped.
2. A channel may completely die and not send any more packets.
3. The channel may reorder packets.
4. It can generate bogus messages.
5. It may deliver the same message multiple times.

In shared memory systems a faulty object may die or become spontaneously corrupted or even send incorrect responses to operations.

Sometimes the various processes are equipped with failure detectors (of varying degrees of reliability). The information processes get from failure detectors can be of use when designing distributed algorithms. For example, if process A knows B might by experiencing a Byzantine failure, A might decide not to trust information it is getting from B.

(Example failure detector: A process might periodically broadcasts a “heartbeat” message. If the heartbeat stops, everyone knows the process has failed. There has been lots of research on implementing and using failure detectors, but we won't talk about them much in this course.)

4. Deterministic vs. Randomized Algorithms

An algorithm could be deterministic or randomized. In distributed systems randomization adds more power and helps in the solution of problems.

The environment (eg. shared objects) may be deterministic or non-deterministic. Non-determinism makes it harder to design algorithms for distributed systems, since the algorithm has to cope with many possible outcomes of a single action.

5. Amount of Knowledge that Processes have

The processes may have differing amount of knowledge about the system.

1. The exact topology of the network could be known.

2. The number of processes could be known.

3. The local topology of the network might be known.

4. Leader of the network might be known.

5. It might be the case that the processes have unique identifiers.

More knowledge makes it easier to solve problems but it is harder to keep the knowledge up-to-date.

Conclusion

There are a **lot** of models. Unlike the different mathematical models for single-process systems (Turing machines, Random Access Machines, etc.), which are all equivalent (Church's Thesis), all these models of distributed systems are not reducible to each other. Some are more powerful than the others. A good way to understand all these models is to know which are more powerful than others.

A model A is at least as powerful as a model B if A can solve all the problems that B can solve. We can prove such a result by showing that any algorithm for system B can be adapted to run in model A (while still solving the same problem).

Eg. Let A be synchronous shared memory system that can tolerate $n/3$ halting failures where n is the number of processes in the system. Let B be the same except that it is asynchronous and only $n/2$ processes can fail. Then A is more powerful than B, since any algorithm that works in model B works (without any alteration) to solve the same problem in model A.

This notion of relative power of models is useful for studying whether or not a problem is solvable in the different models. If we're also interested in the *complexity* of the solutions, we have to take into account the amount of complexity that gets added when transforming the algorithm from one model to the other