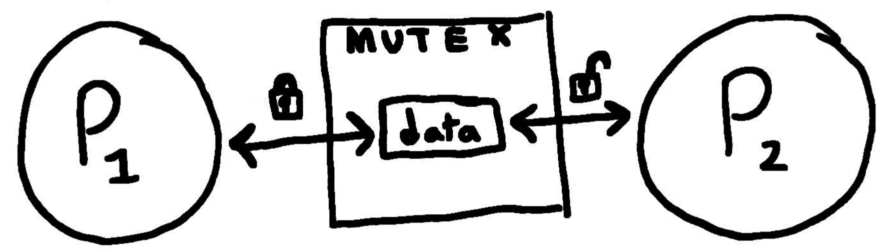
Solving the Mutually Exclusive Problem of Distributed Systems!

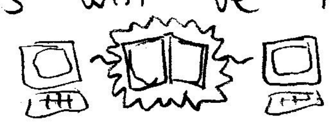


by Bryan Tor

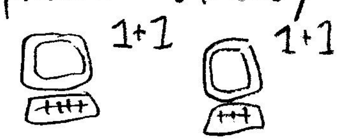
What's this?

In a distributed system, there may be cases where processes have to share resources:

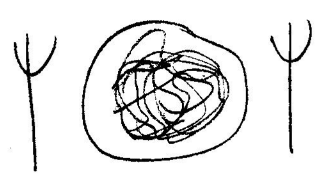
- a common database that multiple processes will be making changes to



- a cache that keeps track of all occurring events, so that processes don't duplicate already occurring events



- not having enough forks on spaghetti night



References

Andersen, Dave. "Distributed Mutual Exclusion." Fall 2009, PowerPoint file.

Kahveci, Ensar Basri. "Distributed Locks Are Dead; Long Live Distributed Locks!" Hazelcast, 16 May 2019.

Kleppmann, Martin. "How to Do Distributed Locking." Martin Kleppmann's Blog, 8 Feb. 2016.

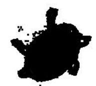
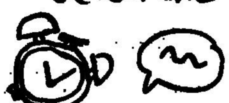
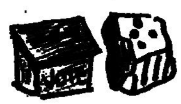
Lamport, Leslie. "Time, Clocks, and the Ordering of Events in a Distributed System." Commun ACM 21 (1978): 558-565

Maekawa, Mamoru. "A √N Algorithm for Mutual Exclusion in Decentralized Systems." ACM Transactions on Computer Systems. 3.2 (1985): 145-159. Crossref. Web.

Xu, Dongyan. "CS603 Process Synchronization." 11 February 2002, PowerPoint file.

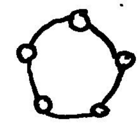
Final Word!

This zine has covered the variety of ways mutual exclusion can be implemented.

- single coordinator: one process keeps track of shared resources! 
- token ring: tokens associated with a resource are shared in a circle 
- timestamps: processes talk to each other, and use timestamps to determine the priority of requests 
- quorum: processes talk to subsets to determine how to allocate resources for everyone 

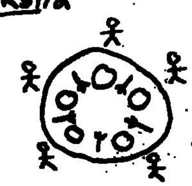
Hopefully this zine helps you figure out how to manage mutual exclusion with your system.

SO... we want to make sure that a shared resource is mutually exclusive for only one process at a time. Common approaches to implementing mutual exclusion in a distributed system include:

- single coordinator 
- token ring 
- timestamps 
- quorum 

Lets learn about them!

Dining Philosophers Problem by Dijkstra

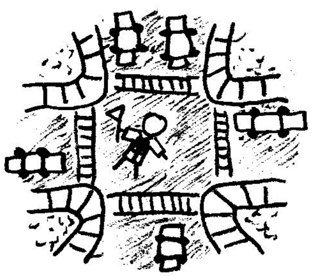


A group of philosophers sit at a round table with bowls of spaghetti. However, there are a limited number of forks. Each philosopher needs two forks to eat, and we don't want philosophers to starve. How can forks be shared between the philosophers? Each philosopher can be like a process, a fork like a shared resource

Single Coordinator

For this approach, there is a single coordinator process that tells everyone what resources are free, and what resources are in use.

A traffic director is like a single coordinator.



- good for managing a busy neighborhood
- don't send them to the highway

Pro: easy to implement, and doesn't require many messages (2) to lock and release resources

Con: Single point of failure, won't scale well if there are several processes making requests

Example Take processes 0, 1, 2, 3, 4, 5

Subsets

S_0 {0,1,2}

S_1 {1,3,5}

S_2 {2,4,5}

S_3 {0,3,4}

Like before, 0, 1, and 2 make requests.

1 recieves a "Failed" from 0.

1 recieves a "inquire" from 5.

1 then sends out a "relinquish", which frees S_0 to accept 1's request, thus evading the deadlock from before.

Overall, the Maekawa algorithm requires about $6\sqrt{N}$ messages to send out requests and release resources.

There is a concern that any failure when a subset tries to process a request can cause everything to fail.

We can avoid these DEADLOCKS by adding three new message types:

Failed: upon recieving a new request, reply "Failed" if there is already an outstanding request with a higher priority

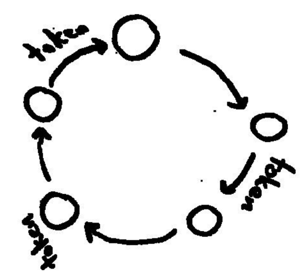
Inquire: upon recieving a new request, reply "Inquire" if there is already an outstanding request with a lower priority

Relinquish = If a process recieves both an "inquire" and "failed" message, send a "relinquish" to your subset. The subset will then release its current resource and move on to the next request, if any.

Token Ring

One (unique) token for every shared resource,
one (unique) token to find them,
One (unique) token to bring them all,
and in a circle share them.

Every shared resource gets one unique token. Each token is passed between resources in a circle, like a game of hot potato.



One you have the potato (token), you can access the shared resource. Pass on the token otherwise if you don't need it.

Pro: All processes have equal access to resources, and is well organized to avoid conflicts

Con: Does not scale well with more processes (N-1 waiting time, where N= number of processes). If one process fails to correctly pass, everything fails

Timestamp-based Requests
Lamport Edition ☆

In 1978, Leslie Lamport proposed a solution to the mutual exclusion problem using time stamps:

- A process will ask all other processes if it can have access to a resource, and adds its own request to a queue.
- When a process gets a request, add it to the queue, and send an acknowledgement back
- When a processes own request is at the top of the queue, and it has recieved a message from everyone else timestamped after the initial request, it can use the resource
- Once finished broadcast a release message
- When a process gets a release, it removes the associated request out of its queue

This is more complex than the last two approaches, so here's an example:

Algorithm (continued)

- When a process recieves a release message, it moves on to the next outstanding request in its queue.

Example Take processes 0, 1, 2, 3, 4, and 5

Subsets

S_0 {0,1,2}

S_1 {1,3,5}

S_2 {2,4,5}

S_3 {0,3,4}

Processes 0, 1, and 2 issue requests.

S_0 = 0,2 send an OK to 0.
1 sends an OK to 1.

S_1: 1,3 send an OK to 1.
5 sends an OK to 2.

S_2: 4,5 send an OK to 2
2 sends an OK to 0.

Oh no! Process 0 is waiting for an OK from 1. Process 1 is waiting for an OK from 2. Process 2 is waiting for an OK from 0.

This is a DEADLOCK an it is bad. ☹

Our subsets are stuck, trying to gain access to resources locked by other subsets, who are in turn waiting for other resources to open up.

Quorum-Based: Maekawa's Algorithm

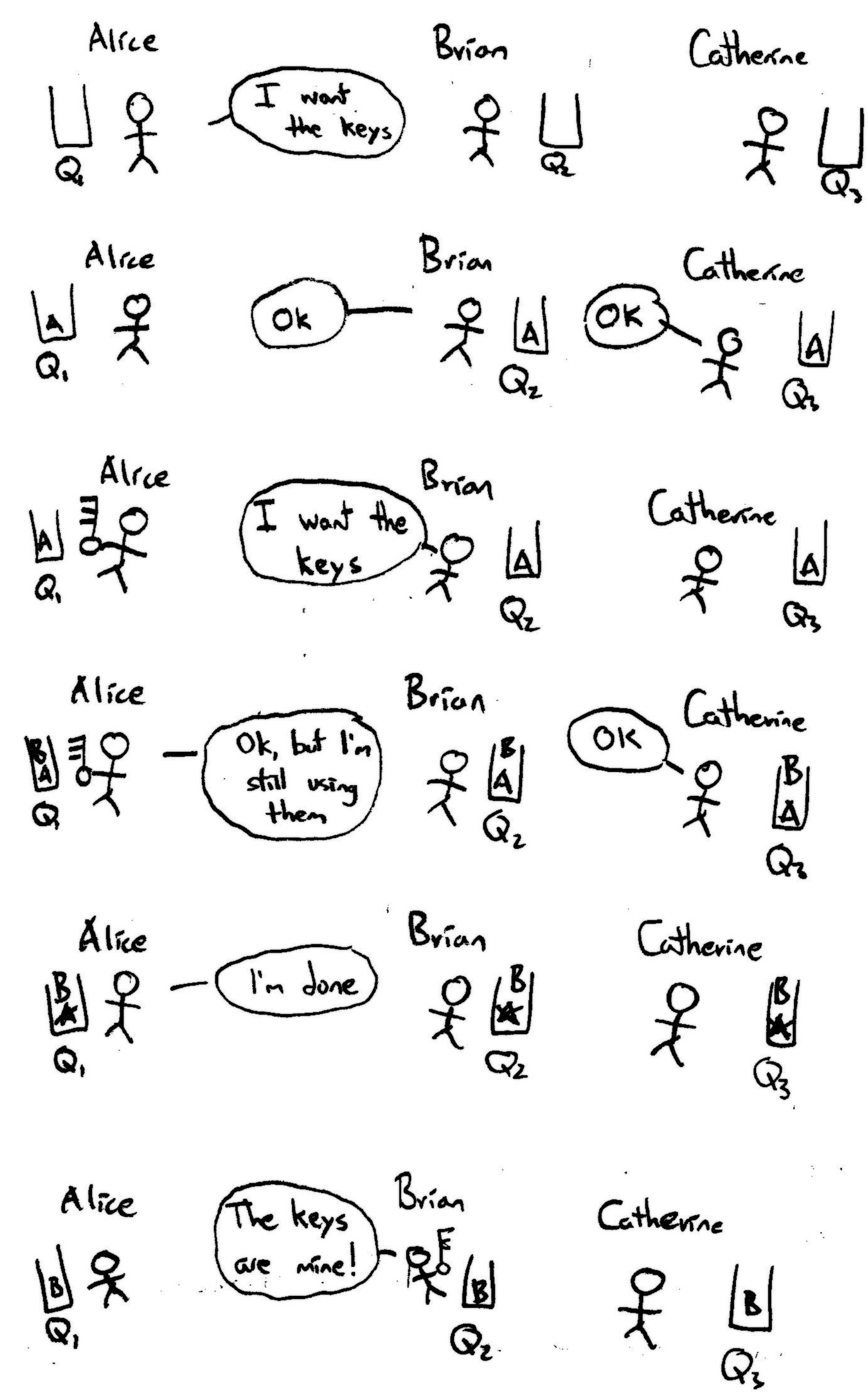
In 1985, Professor Momoru Maekawa proposed the first quorum-based algorithm-based algorithm for mutual exclusion.

In a quorum-based approach, requesting processes only ask a small subset of process for access to resources. For each subset, only one request can be accepted at a time. Every subset will have at least one process in common with another subset.

Algorithm

- A process requests access to a resource by asking all other process in its subset
- If the process doesn't have any outstanding requests, it sends an OK to the requester. Otherwise, it adds the new request into a queue composed of all other made requests
- Processes gain access to a resource when it gets a reply from all other processes in its subset
- After finishing with a resource, a process will send a release message to its subset

Example



A variation of this algorithm was created by Ricart and Agrawala:

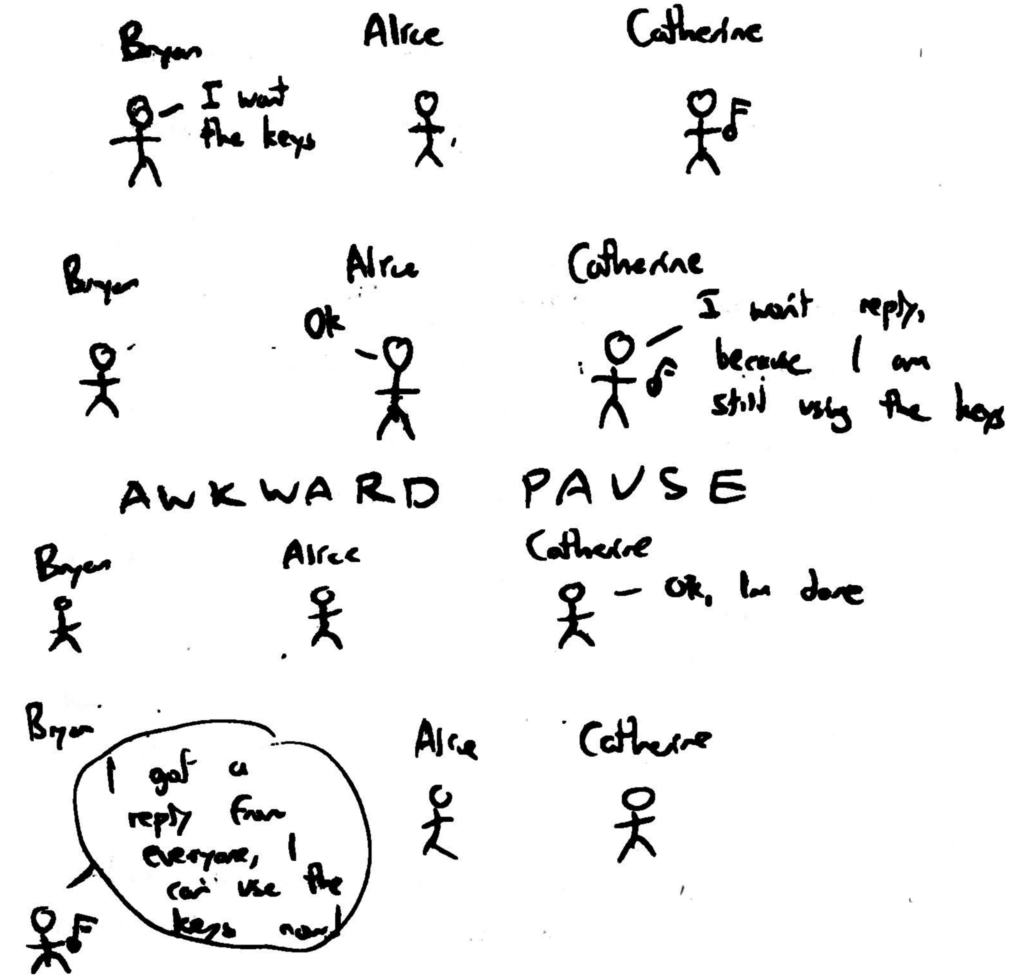
Processes still broadcast requests for a resource

Upon recieving a request:

- If you are using the resource, delay your reply until you're finished with the resource
- If you are waiting for the resource as well, compare request timestamps. If the incoming request was made earlier than your own request, reply to the sender that they can use the resource. Otherwise, delay your reply until you're done with the resource yourself.
- If you aren't waiting on or using the resource, just reply ok

Once a process gets replies from all other processes related to a resource request, that process now can use the resource.

For example:



Yay! Processes get access to resources on a first-come first-serve basis.

Oh no! You need to talk to all the other processes when making a request, which will take a while. (a total of 3(N-1) messages for Lamport, and 2(N-1) messages for Ricart-Agrawala where N = number of processes)

MANY FAILURES ALSO