CONTENTS

CONCURRENCY CONTROL

Distributed Mutual Exclusion

Access to resources can be controlled by placing all references to them inside critical sections and then using mutual exclusion to control entry into those sections.

Mutual exclusion requires:

- at most one process may access the critical section at any one time (**safety**)
- a process requesting access to the critical section will eventually be given it (**liveness**)

In addition, there may sometimes be the requirement that:

- access is granted in a specific **order**.

A way of categorising distributed mutual exclusion algorithms:

- Permission based
- Token based (this includes systems with a central coordinator or distributed agreement using logical clocks e.g., Ricart and Agrawala — see text book)

Permission Based

When a process wants to enter the critical section it asks the other processes for permission and does not proceed until it has all of them.

If a process receives a request and it isn't interested in the critical section it returns its permission immediately. Otherwise a priority has to be established between them.

Safety (mutual exclusion is never violated) is ensured by having enough permissions obtained by the process before entering the critical section.

Liveness (each request is eventually granted) is satisfied by the priority mechanism (such as the use of timestamps or a distributed acyclic directed graph of requesting processes).

Token Based

The right to enter a critical section is controlled by the possession of a token.

Safety is ensured by making the token unique.

The movement of the token can be categorised as:

- perpetually in motion
  Processes form a logical ring and the token is passed around the ring from process to process. If the process is not interested in the token it passes it on to the next process in the ring, otherwise it enters the critical section and passes the token on when it has finished.
- Token asking
  The token does not move unless it is requested by a process. When a process wants to enter the critical section it requests the token and waits until it arrives.

Recovery may, for example, involve regenerating a token and reforming the logical ring.

Distributed Agreement

Processes wanting the critical section multicast a request message to other relevant processes and can only have it when all other processes have replied. Conditions for processes to reply ensure that mutual exclusion is satisfied.

This assumes that all relevant processes know each other and cooperate.

Each process keeps a state (**held**, **wanted**, or **released**) to identify its interest in the token.

- if a process requests the token and the state is **released** everywhere, it gets the token immediately
- if any process holds the token it will not reply until it has finished with it.
- if more than one process requests the token then the reply is passed to the one with the lowest timestamp (or those being equal, the one with the lowest process id)

Central Server

The server controls access using a **token** which represents permission to enter the critical section.

To enter the section the client sends a request to the server and waits.

If no other client holds the token the server responds immediately with the token. Otherwise the request is queued and on return of the token the server passes it to the next request in its queue.

Recovery:

- If the server crashes then the recovery needs to take into account that a client holds the token or may think it has returned it, and
- that there may have been queued requests (note that when recovered they may be in a different order from the original queue)

Ring Based

Mutual exclusion between **n** cooperating processes arranged in a logical ring. The token is passed between processes around the logical ring in one direction.

If a process doesn't need the token it passes it on, otherwise the process retains it until it has finished. On exit from the critical section it passes the token on.

Processes do not necessarily receive the token in the order of their requests.

On failure of one of the processes the token cannot keep circulating until the failed process is removed from the ring.

If a process fails when holding the token, the token has to be regenerated somehow.

Regenerating a token involves electing a process to do the regeneration. Only one process should be chosen.

A simple ring election algorithm that chooses the process with the largest process identifier is:

- initially all processes are non-participants.
- Any process may begin an election by sending its id to its neighbour in an election message and marking itself as a participant.
- When a process receives an election message:
  - if the id in the message is greater than its id, it passes the message to its neighbour and marks itself as a participant.
  - if the id is less and the receiver is not a participant, it substitutes its id, forwards the message and marks itself as a participant..
  - if the id in the message is its own id, then it becomes the elected process and sends an **elected** message around the ring to terminate the election. It can then regenerate the token.

Logical and Physical Clocks

Clocks are used for synchronising activities (among other things).

A clock can be

- a physical device (e.g., quartz crystal clock) that maintains an idea of time by producing regular events.
- Logical clocks are based on monotonically increasing software counters and capture the relationships between events but not an absolute sense of time.

Logical Clocks

These are based on the observations (Lamport)

- if two events occur in the same process then they occur in the observed order
- when a message is sent between processes, the send happens before the receive.

Which is sometimes called **causal ordering** (there are other names)

It works as follows:

- the processes logical clock is incremented before each event
- when a process sends a message it sends the logical clock value with it
- when a process receives a message (plus logical time) it sets its clock to the maximum of the current value and the received time before it receives the message (incrementing the logical clock).

e.g.,
$$\begin{aligned} LC(e_i) &:= LC + 1 \text{ where } e_i \text{ is an internal or send event} \\ &:= max\{LC, TS(m)\} + 1 \text{ where } e_i \text{ is a receive} \end{aligned}$$

LC is the logical clock, e_i is an event, and TS is a timestamp.

Physical Clocks

Physical clocks tend to work at slightly different rates which may vary due to physical conditions such as temperature. The clocks on a distributed system tend to drift apart and need to be resynchronised periodically to maintain a single global view of the time.

Compensating for Clock Drift.

- local clock behind reference clock (needs to be adjusted forward):
  - in one jump
  - a series of small jumps until the right time is reached (effectively continuously)
- local clock ahead of reference clock (Cannot be adjusted in one jump because time will go backwards and this will upset some software. Have to cause the clock to run slow e.g., ignore some hardware clock ticks).

Physical Clocks — Cristians Algorithm

Message transmission takes non-zero time, so a processor receives the time from a time server after some transmission time t_{trans} at a time $t + t_{trans}$ where t is the time that the server sent the message containing the time.

The transmission time may vary.

The processor may estimate the value of t_{trans} as $(t_{receive} - t_{send})/2$.

Accuracy can be improved by making several requests and using the fastest reply (least likely to have been delayed en-route).

Problems:

- malfunctioning time server (no redundancy or checks)
- malicious interference (no authentication)