COMPSYS 725 2019
Coordination I

Preamble

Last time, on COMPSYS 725...

- We talked about the need for synchronisation
- We talked about physical time and clock synchronisation
- We talked about logical clocks (scalar and vector)

Quick check!
Are you able to assign vector clock timestamps, and compare them to test if $e_x \rightarrow e_y$?

By the end of today's lecture

- You should be able to recall the properties motivations and fundamental principles of mutual exclusion
- You should be understand central-server, ring-token, and Ricart-Agrawala multicast algorithms for mutual exclusion, and be able to discuss their relative merits

Links to LOs:

- *Understand issues arising from resource sharing and synchronisation in distributed applications, and possible solutions*

1 Principles

Distributed applications involve multiple **processes** that need to **coordinate their actions** and **agree on shared values**.

Challenge

- How do we **safely** share resources?
  - Files, data
  - Physical things ... servers, conveyors...

Example

Consider the following example, where we have two pushers that can move a work piece that descends from above.

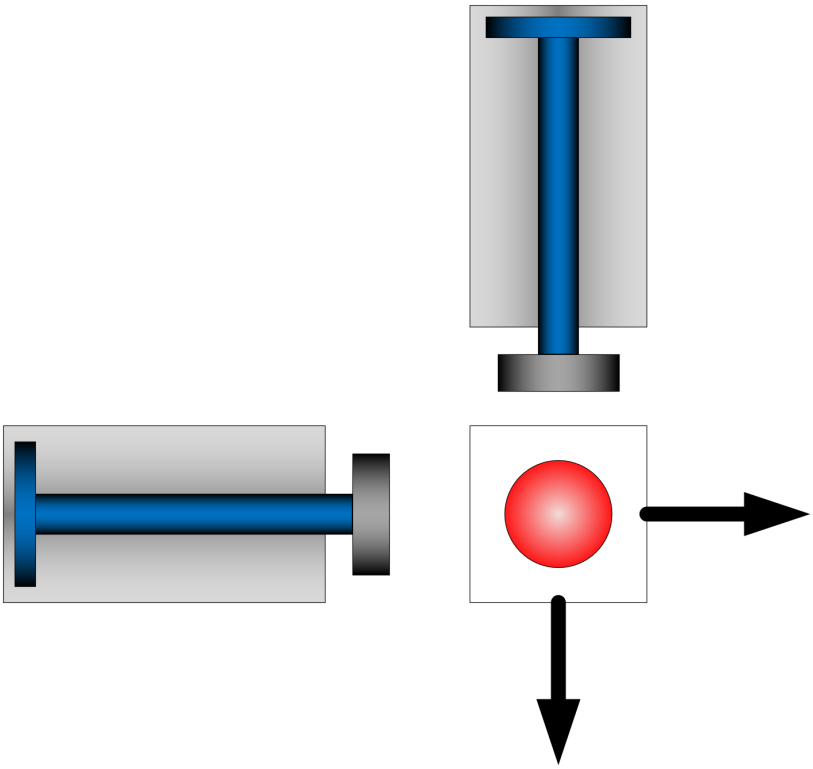

Figure 1: Two Pusher Example

- Two pushers, work piece descends
- Either pusher should activate, but not both
- We want to *prevent interference* and *ensure consistency*

- What is the shared resource?
  - The work piece?
  - The physical space?

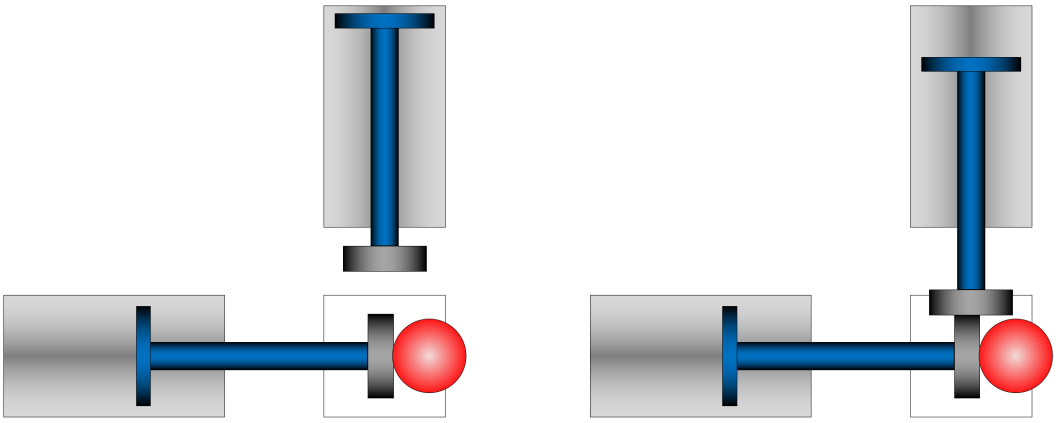

Figure 2: Bad collision

To make sure that the conflict doesn't occur, we need to *serialise* the concurrent access – in other words, ensure that only one process accesses a shared resource at a time

- We can call the part of a process' execution that uses a shared resource as a **critical section**
  - You might find the concept/term familiar from previous study – Mutexes, shared variables etc. are often discussed in the context of Operating Systems
- In a distributed system, we have to use solutions that are based solely on message passing

First, let's establish the assumptions for today's discussion:

Distributed system model

- N processes $p_i, i = 1, 2, ..., N$
- For now, assume one critical section
- Asynchronous system – i.e., physical clocks aren't synchronised, message times are finite but unbounded, execution times are finite but unbounded
- Processes are failure-free
- Message delivery is reliable – i.e., messages are eventually delivered intact, exactly once
- Processes are *well behaved* – i.e., they spend a finite time in the critical section

We can think about execution of critical sections as having three phases/operations:

1. `enter()` – enter the critical section (blocks if necessary)
2. `resourceAccess()` – access the shared resource in the critical section
3. `exit()` – leave the critical section

- The different algorithms essentially determine the sorts of messages that need to be passed during `enter()` and `exit()` to enable mutual exclusion.
- For example, `enter()` might involve sending a series of REQUEST messages, and waiting for a certain number of responses.

There are certain properties that we want to try and guarantee in our mutex algorithms:

Key requirements

- ME1: Safety Property – at any instant, only one process can execute the critical section
- ME2: Liveness Property – requests to enter/exit the critical section eventually succeed. We might also say that this property implies the absence of **deadlock** or **starvation**
- ME3: Fairness Property Each process gets a "fair" chance to run the critical section. In distributed systems, if all requests are related by happened-before, then it's not possible for a process to enter the critical section more than once while others wait.

  – If one request to enter the critical section happened-before another, then entry to the critical section is granted in that order

Deadlock: two
stuck as a resul
terdependence
Starvation: i
ment of a proce
a critical sectio

In some circles, ME1 is considered to be absolutely necessary, while ME2 and ME3 are generally considered to be very important.

We also need to establish some terms of reference for comparing algorithms. These *might* include:

- **Message complexity** or ***bandwidth*** – proportional to the number of messages sent in `entry()` and `exit()`
- **Synchronisation delay** – the time taken between one process exiting the critical section, and another process entering it
- **Client delay** – time taken by a process at each `entry()` and `exit()` operation
- **Response time** – the time taken for the responses to come back after a request has been sent out

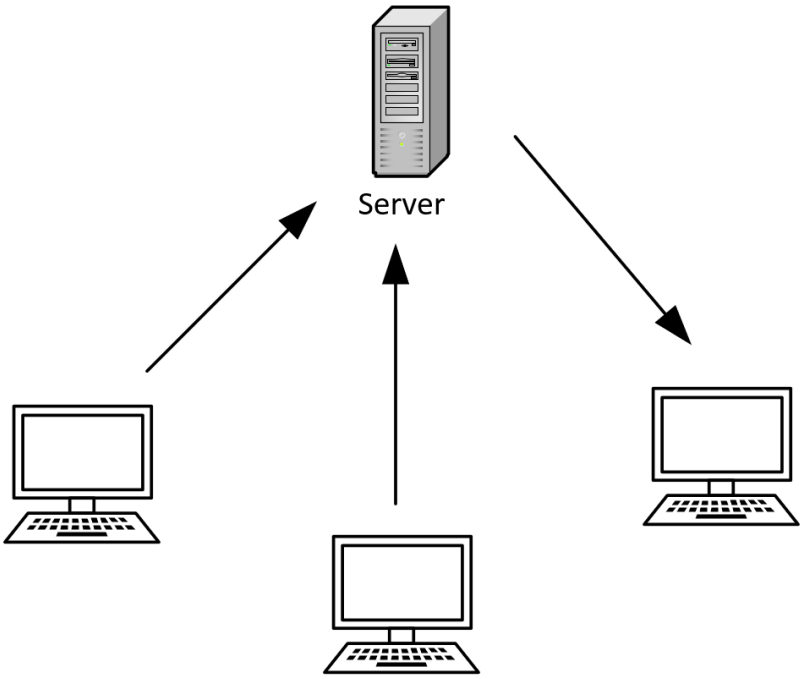

Figure 3: Central Server

2 Central Server

The simplest mechanism is *Central Server*, where a process is used to manage access to the critical section

- Server maintains a queue of requests
- The server is responsible for sending a *token* to grant access
- Clients can send a REQUEST for access to the server (`as part of enter()`)
- If there's no queue of processes, the server sends the token to the requesting process
- The server receives the token back from the process when it's finished (as part of `exit()`)

What happens if the queue is **not** empty?

- The server adds the newly arrived REQUEST to the queue
- When then token is returned, the server finds the next process on the queue, and then gives that process the token
- In the best case, how many messages are transmitted before a process can access the critical section?
  - Two! A REQUEST message, and the accompanying GRANT (i.e., with the token)
- How many messages after the critical section?
  - One, the RELEASE message

- What is the synchronisation delay?
  - The time taken for a round-trip – the RELEASE to the server, and the GRANT to the next process

Does the central server satisfy **all three** ME properties?

- Remember, processes can send messages to each other (when they're not wanting to use the critical section...)

3 Ring Token

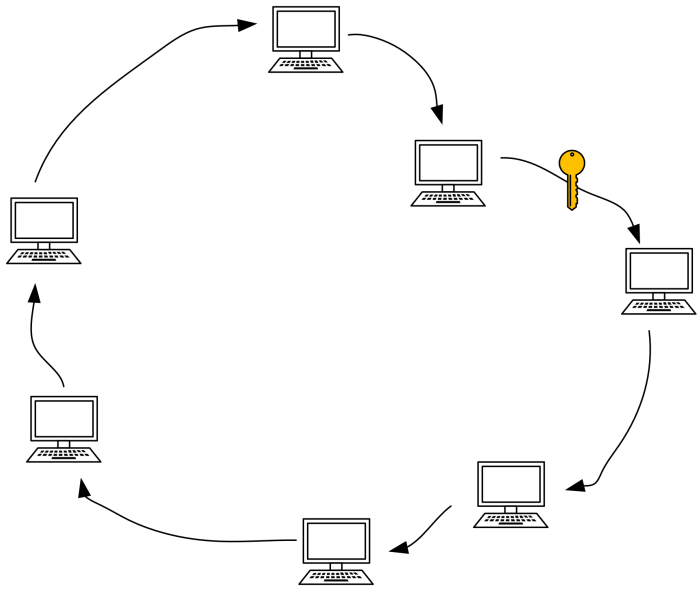

Figure 4: Ring Token

- A token is passed around in a *logical* ring
  - This can be unrelated to the physical topology of the network!
- If a process has the token, and wants to execute the critical section, it uses does
- Otherwise, it passes the token on

Does this algorithm satisfy ME3?

4 Multicast

Ricart and Agrawala's algorithm builds on top of the notion of logical clocks so that it can satisfy ME1–ME3.

- A process that wants entry to a critical section first multicasts a REQUEST
- Access is "granted" when the requesting process has received replies from all the other processes

Each process has an identifier

Process states
Processes in the system are said to be in one of three states:

- RELEASED – outside the critical section
- WANTED – wanting entry to the critical section
- HELD – inside the critical section

Each process maintains a Lamport (scalar) clock

Procedure

- Let's say a process wants to gain access to the critical section
- It multicasts a REQUEST with the information $< T, p_i >$ being its timestamp and process identifier
- If a process receives a REQUEST and it is currently in the RELEASED state, it sends a reply straight away
- If a process receives a REQUEST and it is currently in the HELD state, it doesn't send a reply until it is done with the critical section

Concurrent requests
If more than one process tries to gain access (i.e., more than one process is in WANTED):

- Whichever process has the lowest timestamp will be granted entry first
- So, if a process in WANTED receives a REQUEST from a process, and the REQUEST has a lower timestamp, it will send back a reply
- If REQUEST comes from a process that has a higher timestamp, it defers the response until after it has entered the critical section

Does Ricart-Agrawala achieve ME1? ME2? ME3?

- 2(N-1) messages for gaining entry: N-1 requests, and N-1 replies
- Synchronisation delay is transmission time for one message

5 Discussion

- What happens when messages are lost?
- What happens when a process crashes?