W 8 - 12 lecture
W 8 - 11 work
W 8, 10, 12 applied

> W8 GPU vs NPU

> W9 Synchronisation, Mutex, deadlocks

> W10 Election Algorithms, Concurrency control

end of today 12/10/24
notes, workshop q, summary formulas

> W11 Matrix Multiplication, Partitioning

> W12 Exponential growth vs Performance Planning

> GPU vs NPU vs CPU

CPU	GPU	NPU
General tasks.	Numb. lots	Ls AI tasks
Ls Many Instructions	of repetitive tasks	
Lower bandwidth	Very high bandwidth	
ILP	TLP	

>W9 : Synchronisation, Mutual exclusion, deadlocks

Synchronisation → why? We need to sync processes for many things. To order events, log files etc.

↓

How? 2 main types:

logical clocks:
keep ordering of local events on a single node, by relative ordering of events.

physical clocks:
Sync events to a physical clock. UTC

More simple
Partial ordering

VS

More complicated to sync blw processes
Total ordering

↓

Algorithms:

Lamport's Algorithm
↳ determine event order only

↓

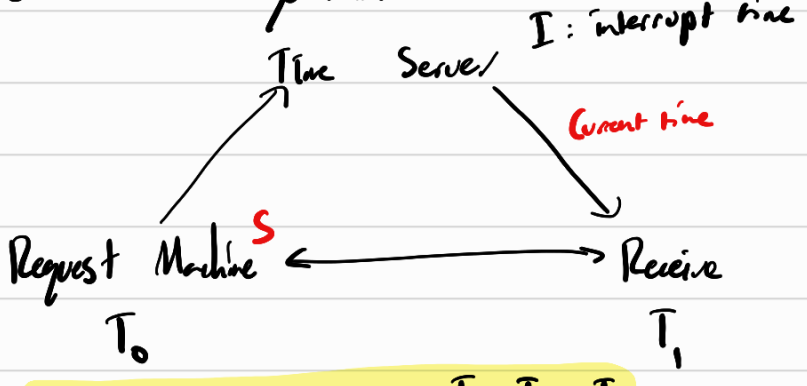
$A \rightarrow B$ A happens before B

↳ when:
- A + B on same process, A occurs before B
- A sends message, B receives
- If x, y are independent on diff processors → concurrent

$x \not\rightarrow y$, $y \not\rightarrow x$

if $A \rightarrow B$, then $C(A) < C(B)$ but not other way

Cristian's Algorithm



$$S_{time} = Curr_{time} + \frac{T_1 - T_0 - I}{2}$$

Averaging Algorithm

- Resync every fixed interval R
  ↳ set local time using average

P_0 should be advanced by Δt_0

$$\Delta t_0 = \frac{t_0 + t_1 + t_2 + t_3}{4} - t_0$$

Berkley's Algorithm → Master polls slaves periodically, sends what slave has to adjust by

- Find time difference b/w master and slave processes.
- Average

Corrected time = T_M + Average time

ΔT_n = Corrected time − T_n

> WQ

Mutual exclusion: A concurrency control property to prevent race conditions

↓

Ensure concurrent accesses of processes to a shared resource is serialised.

↳ Only one process allowed to execute in a critical section

↳ segment of code executed by multiple concurrent threads/processes accessing shared resources

Ways of ensuring Mutex:

> Centralised Algorithm:

↳ A process is a 'co-ordinator'

1) If process enters critical section, sends request to co-ordinator
2) If no other process, send grant
3) Otherwise put in queue
4) Once process exits, process sends release

Pros: - First-come, first serve (fair)
- Simple + easy
- Only 3 messages per critical section use: request, queue, release

Cons: - If co-ordinator fails, system may fail
- Process cannot distinguish b/w crashed co-or or permission denied
- Single co-or performance bottleneck

> Distributed Algorithm

↳ Process sends message to all other process,

containing: Critical section name + Process no. + curr time

↳ All other processes receive

if process not in critical section

↳ send Ok

else check process msg send time vs received msg time

if rec time < send time

↳ Ok

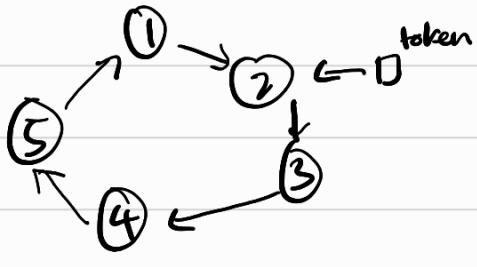
else → put at end of queue

if one process does not respond, system doesn't continue

2(n-1) messages n = no. processes

Ring - Token:

- Nodes in ring. One token held and passed b/w processes.
  - ↳ if node has token and needs critical section, it can
  - ↳ otherwise pass token in ring



> Summary

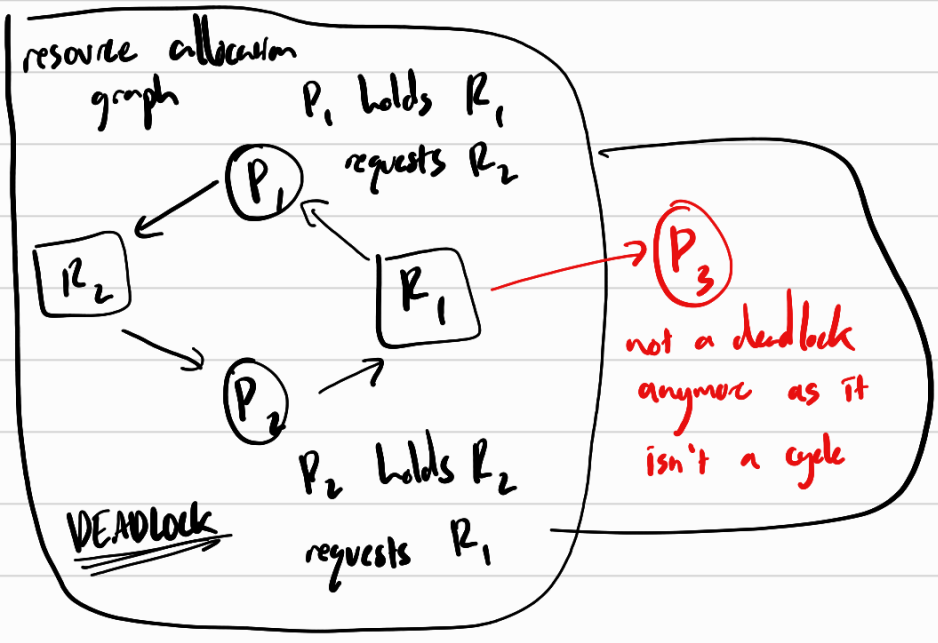
	Messages	Problems
Centralised	3	- Co-ordinator crash / bottleneck
Decentralised	3mk (k=1,2...)	- Starvation / low efficiency
Distributed	2(n-1)	- Crash of any process
Token-ring	1 to ∞	- lose token / process crash

> Dead locks : When a process cannot proceed because it needs resource held by other process which current process holds.

Conditions:

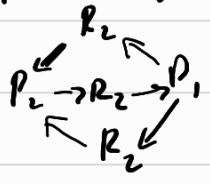
1) Mutual Exclusion: Resource held by at most 1 process
2) Hold and wait: Process that holds resource is waiting for another
3) Non-preemption: A resource can only be released by process after completion
4) Circular wait: $P_0 \rightarrow P_1 \dots \rightarrow P_0$

P_0 waits for P_1, waits for P_2 .... waits for P_0



Deadlocks in graph

if 1 copy of all resources:
↳ cycle
if multiple copies:
↳ knot: All vertices in knot have outgoing edge that terminate at another vertex in knot.



Wait for graph: remove resources

> Handling Deadlocks

- Ostrich algorithm: Ignore problem

> Detection: Usually easier than prevention

↳ after: kill processes or abort transactions

- Centralised detection

↳ Each machine maintains resource graph

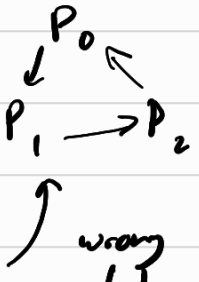
↳ Co-ordinator merges graphs of machines together

↳ Message sent to co-ordinator to update

- False deadlocks:

If P_1 releases P_0 first,
then P_1 asks P_2.

↳ Message sent can detect
P_1 wait P_2 before P_1 release P_0



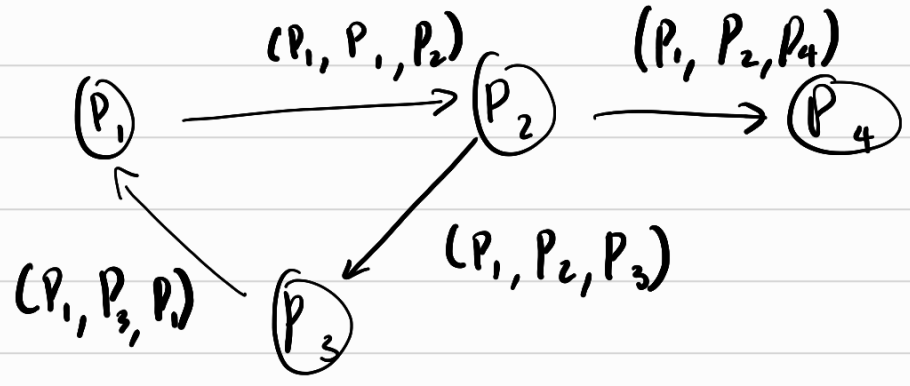
↑ wrong WFG, false deadlock!

↳ Must use global ordering
coordinator or ask each machine if it has release

- Distributed - CMH

Probe message generated when process needs to wait for resource.

↳ Contains: - Process being blocked
- Process sending message
- Process to send to



> W 10

Election Algorithms: To decide a coordinator
- responsible for ensuring mutex and detecting deadlocks in centralised algorithms

Assumptions: - Processes have unique id.
- All processes know each other
- Process do not know which other process is alive.

Election: 1) Maximum number process selected
2) Elected to be coordinator
3) All processes must agree

> Bully Algorithm (Hector Molina)
↳ 1) Process P sends election msg to higher processes
2) If higher process responds → new leader, ok message sent to P
3) No response → P wins election (timeout) ↳ send coordinator message

cons:
- Timeout overhead - slow
- May have delayed response after timeout

> Ring algorithm
↳ ring topology of processes.
1) Process finds communicator is dead
2) Send election msg
3) Passed down until it reaches original sender
4) Coordinator msg passed through, signalling new coordinator

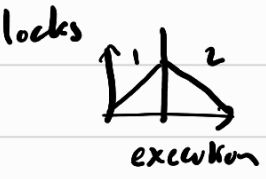
Dead processes → Process needs to wait for timeout before bypassing process,
↳ delays

> Concurrency Control
↳ Ensure multiple transactions can execute without data inconsistencies / interference.

> Conflicts
Read-Write Conflict: Shared - exclusive conflict
Write-Write: exclusive - exclusive

Methods to manage conflicts:
> Lock based
↳ Read locks: Data only to be read
Shared Mode → can be accessed by other shared
↳ Write locks: Data only to be updated
Exclusive Mode → cannot be accessed until lock removed

2-phase locking: GUARANTEES Serializability
1) Gain all locks needed
2) Releases locks as it goes



> Timestamp: Uses ordering (Logical or physical)
T can read x when:
$T \geq x_{write\ time}$

T can write to x:
$T \geq x_{write\ time}$
and
$T \geq x_{read\ time}$

> Optimistic: Assumes no conflicts until end
3 phases: 1) read → local computations
2) validate → check no conflicts
3) write → to database
Validate: ↳ checks for any T_i, T_j that either
1) T_i finish 3 phase before T_j or
2) T_i finish before T_j write or 3) T_i finish read before T_j read