IF3130 – Sistem Terdistribusi

Mutual Exclusion

Achmad Imam Kistijantoro (imam@informatika.org)

Sinkronisasi Proses

- Koordinasi eksekusi antar proses
  - Sebuah proses harus menunggu proses lain selesai
  - Resource bersama membutuhkan akses eksklusif

Sistem Terpusat

- Mutual exclusion
  - Test and Set (hardware instruction)
  - Semaphore
  - Messages
  - Condition variables

Distributed Mutual Exclusion

- Asumsi: resources memiliki identitas
  - Identitas diberikan bersama request
  - e.g. Lock("printer").
  - Lock("table:students")

- Goal
  - Membuat algoritma yang memungkinkan proses mengirim request dan mendapatkan akses eksklusif ke resources yang tersedia pada jaringan

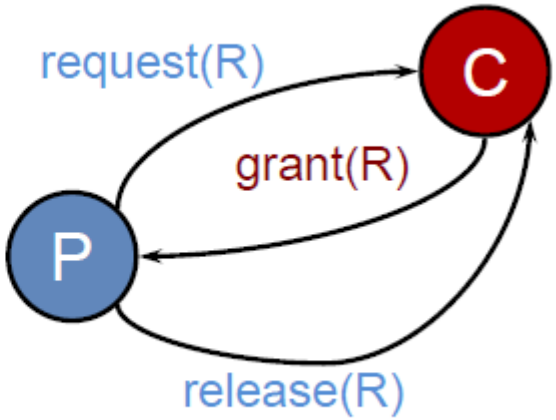
Kategori algoritma

- Centralized
  - Proses dapat mengakses resource setelah meminta akses ke koordinator
- Token based
  - Proses dapat mengakses resource jika proses tersebut memegang token yang mengijinkan untuk mengakses resource
- Contention-based
  - Proses mengakses resource melalui distributed agreement

Centralized Algorithm

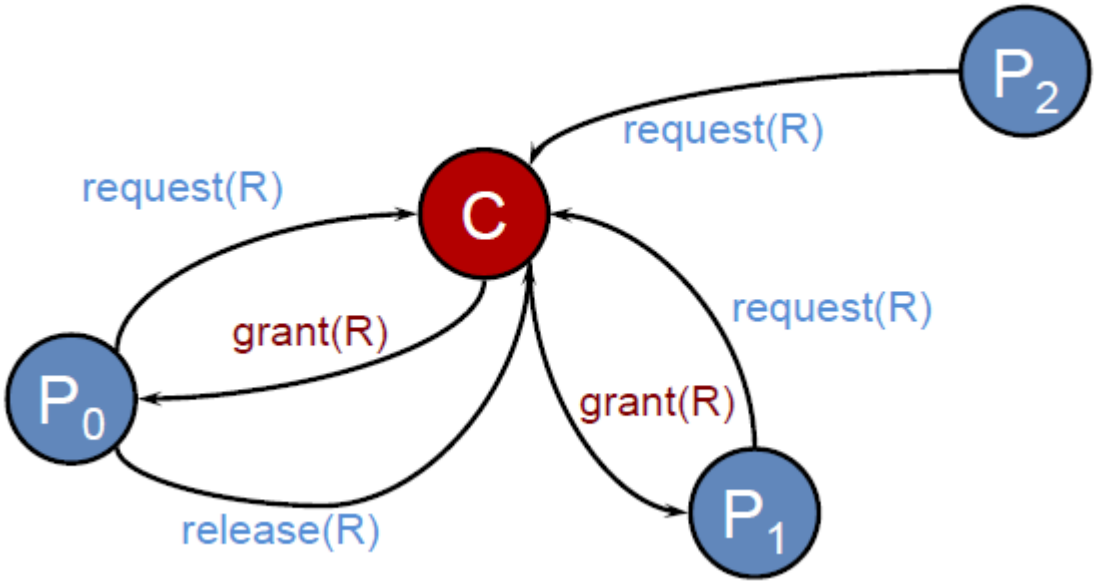
- Meniru sistem prosesor tunggal
- Sebuah proses dipilih menjadi koordinator

1. **Request** resource
2. Wait for response
3. **Receive grant**
4. *access resource*
5. **Release resource**



Centralized algorithm

- Jika ada proses lain yang meminta akses,
  - koordinator tidak memberikan akses sebelum resource tersebut dilepas oleh proses sebelumnya
  - Mengelola queue



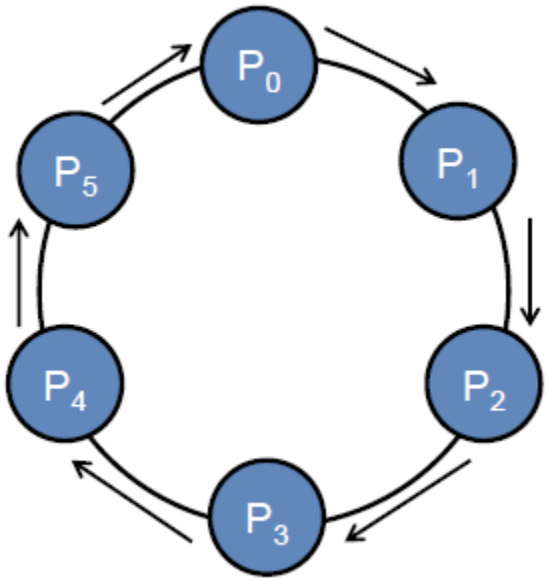
Centralized algorithm

- Benefit
  - Fair: setiap request diproses sesuai urutan
  - Mudah diimplementasikan, dan diverifikasi

- Problem
  - Proses tidak dapat membedakan antara koordinator dead dengan terblock
  - Centralized server dapat menjadi bottleneck

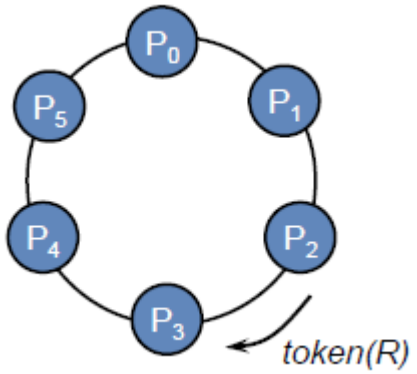
Token ring algorithm

- Asumsi: diketahui sejumlah proses
  - Terdapat ordering di antara proses tersebut
  - Bangun logical ring di antara proses
  - Proses berkomunikasi dengan tetangganya

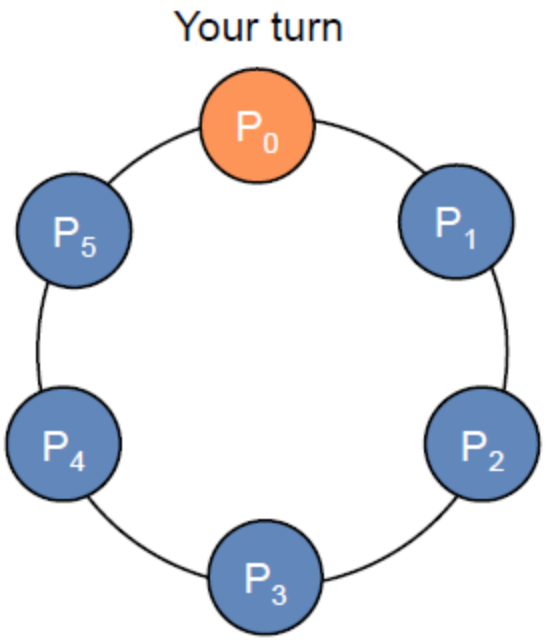


Token ring algorithm

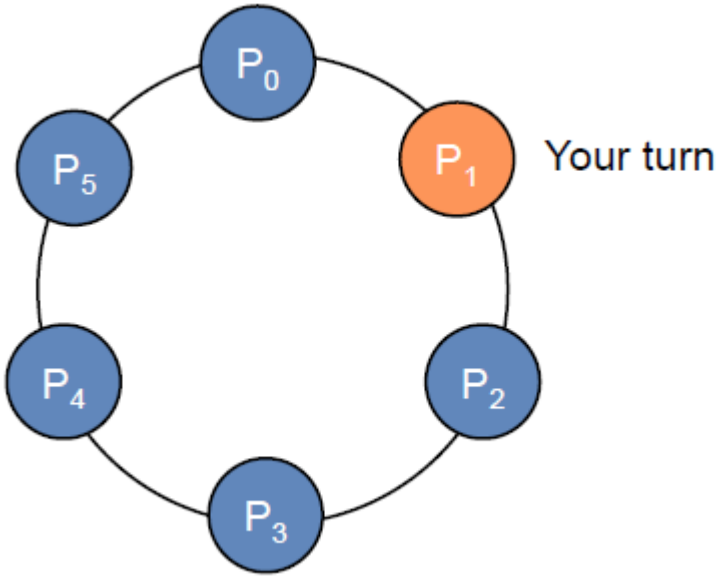
- Initialization
  - Process 0 gets token for resource R
- Token circulates around ring
  - From Pi to $P_{(i+1)}$mod N
- When process acquires token
  - Checks to see if it needs to enter critical section
  - If no, send ring to neighbor
  - if yes, access resource
    - Hold token until done



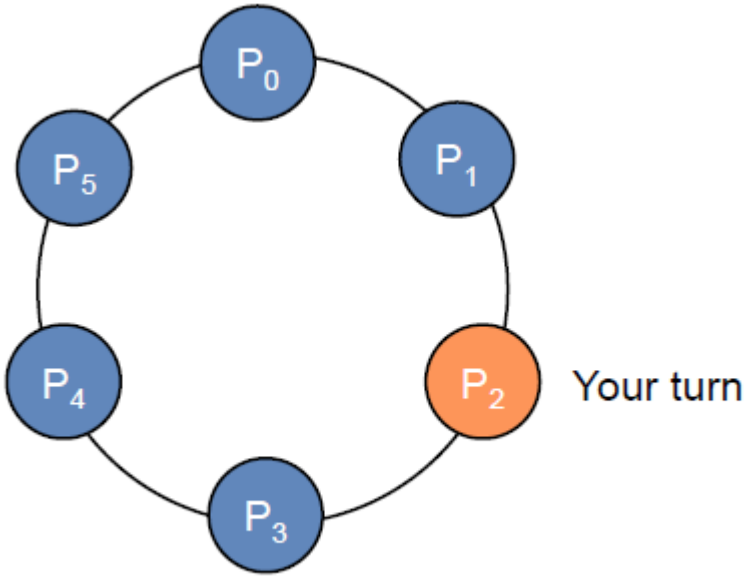
Token ring algorithm



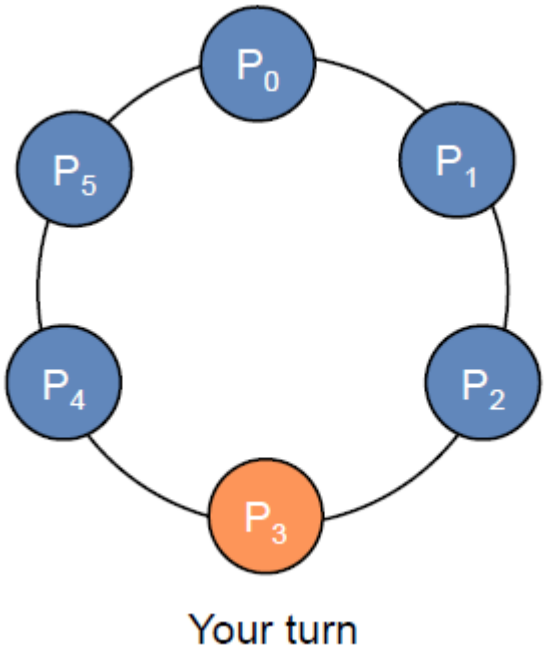
Token ring algorithm



Token ring algorithm



Token ring algorithm



Token ring algorithm

- Only one process at a time has token
  - Mutual exclusion guaranteed
- Order well-defined (but not necessarily first-come, first-served)
  - Starvation cannot occur
  - Lack of FCFS ordering may be undesirable sometimes
- If token is lost (e.g., process died)
  - It will have to be regenerated
  - Detecting loss may be a problem

(is the token lost or in just use by someone?)

Lamport mutual exclusion

- Each process maintains request queue
  - Queue contains mutual exclusion requests
  - Messages are sent reliably and in FIFO order
  - Each message is time stamped with totally ordered Lamport timestamps
- Ensures that each timestamp is unique
- Every node can make the same decision by comparing timestamps
  - Queues are sorted by message timestamps

Lamport mutual exclusion

- Request a critical section:
  - Process *Pi sends request(i,Ti) to all nodes*
    - … and places request on its own queue
  - When a process *Pj receives a request:*
    - It returns a timestamped *ack*
    - Places the request on its request queue
- Enter a critical section (accessing resource)
  - *Pi has received acks from everyone*
  - *Pi's request has the earliest timestamp in its queue*
- Release a critical section:
  - Process Pi removes its request from its queue
  - sends *release(i,Ti) to all nodes*
  - Each process now checks if its request is the earliest in its queue
    - If so, that process now has the critical section

Process	Time stamp
P_4	1021
P_8	1022
P_1	3944
P_6	8201
P_{12}	9638

Sample request queue
Identical at each process

Lamport mutual exclusion

- *N points of failure*
- A lot of messaging traffic
  - Requests & releases are sent to the entire group
- Not great … but demonstrates that a fully distributed algorithm is possible

Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks
- When a process wants to enter critical section:
  1. Compose message containing:
     - Identifier (machine ID, process ID)
     - Name of resource
     - Timestamp (e.g., totally-ordered Lamport)
  2. Reliably multicast request to all processes in group
  3. Wait until everyone gives permission
  4. Enter critical section / use resource

Ricart & Agrawala algorithm

- When process receives request:
  - If receiver not interested:
    - Send OK to sender
  - If receiver is in critical section
    - Do not reply; add request to queue
  - If receiver just sent a request as well: (*potential race condition*)
    - Compare timestamps on received & sent messages
    - Earliest wins
    - If receiver is loser, send OK
    - If receiver is winner, do not reply, queue it
- When done with critical section
  - Send OK to all queued requests

Ricart & Agrawala algorithm

- Not great either
  - *N points of failure*
  - A lot of messaging traffic
  - Also demonstrates that a fully distributed algorithm is possible

Lamport vs Ricart & Agrawala

- Lamport
  - Everyone responds (acks) … always – no hold-back
  - *3(N-1) messages*
    - Request – ACK – Release
  - Process decides to go based on whether its request is the earliest in its queue
- Ricart & Agrawala
  - If you are in the critical section (or won a tie)
    - Don't respond with an ACK until you are done with the critical section
  - *2(N-1) messages*
    - Request – ACK
  - Process decides to go if it gets ACKs from everyone

Election algorithms

Elections

- Need one process to act as coordinator
- Processes have no distinguishing characteristics
- Each process can obtain a unique ID

Bully algorithm

- Select process with largest ID as coordinator
- When process P detects dead coordinator:
  - Send *election message to all processes with higher IDs.*
    - If nobody responds, P wins and takes over.
    - If any process responds, P's job is done.
  - Optional: Let all nodes with lower IDs know an election is taking place.
- If process receives an election message
  - Send *OK message back*
  - Hold election (unless it is already holding one)

Bully algorithm

- A process announces victory by sending all processes a message telling them that it is the new coordinator
- If a dead process recovers, it holds an election to find the coordinator.

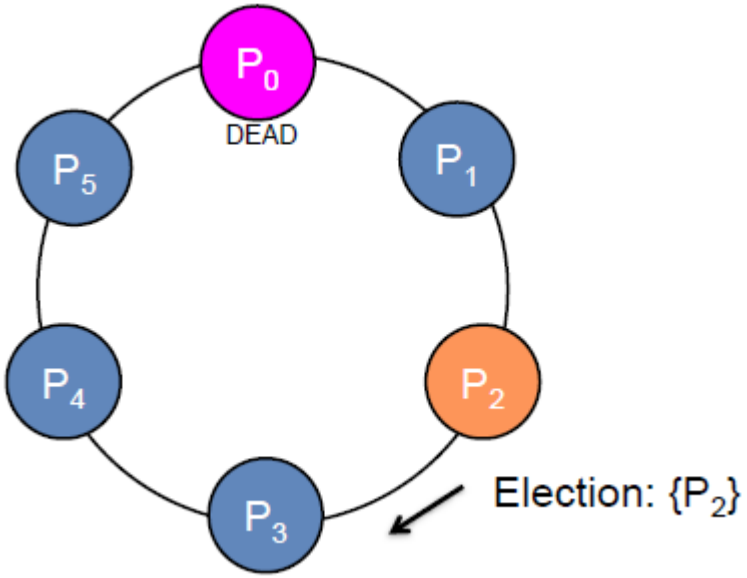
Ring algorithm

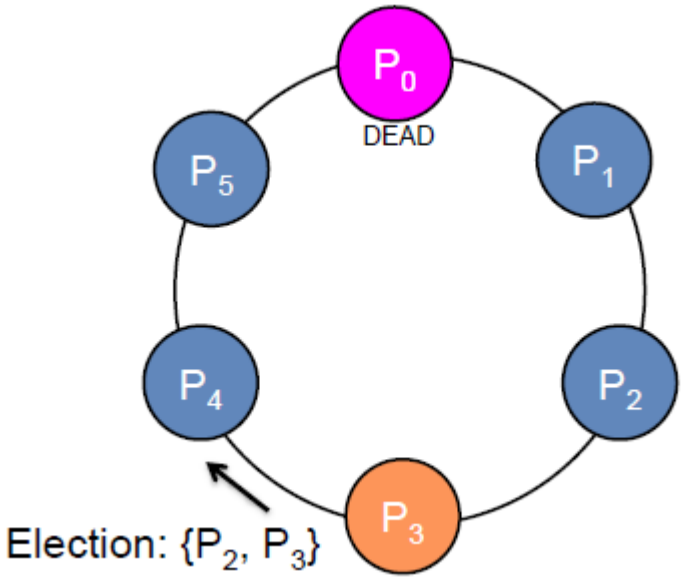
- Ring arrangement of processes
- If any process detects failure of coordinator
  - Construct election message with process ID and send to next process
  - If successor is down, skip over
  - Repeat until a running process is located
- Upon receiving an election message
  - Process forwards the message, adding its process ID to the body

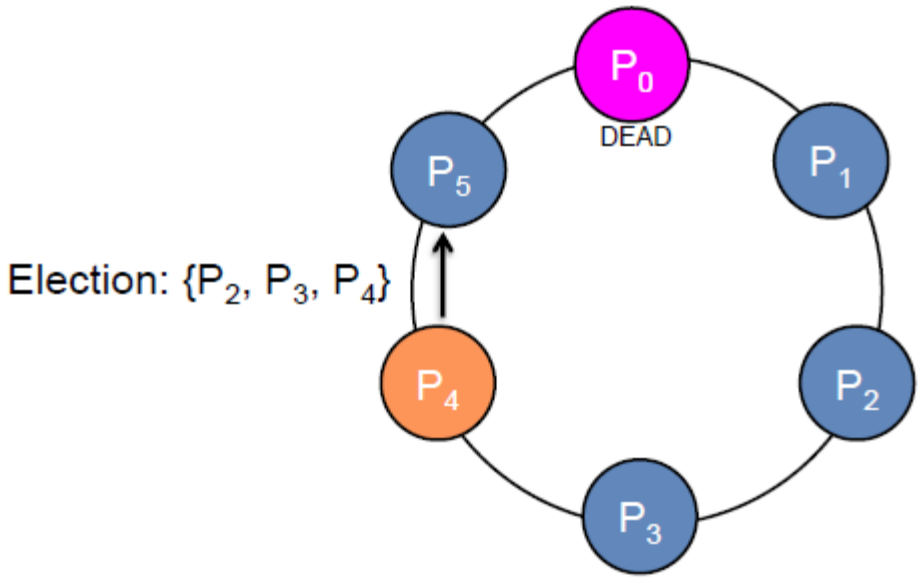
Ring algorithm

- Eventually message returns to originator
  - Process sees its ID on list
  - Circulates (or multicasts) a **coordinator message announcing** coordinator
    - E.g. lowest numbered process

Ring algorithm

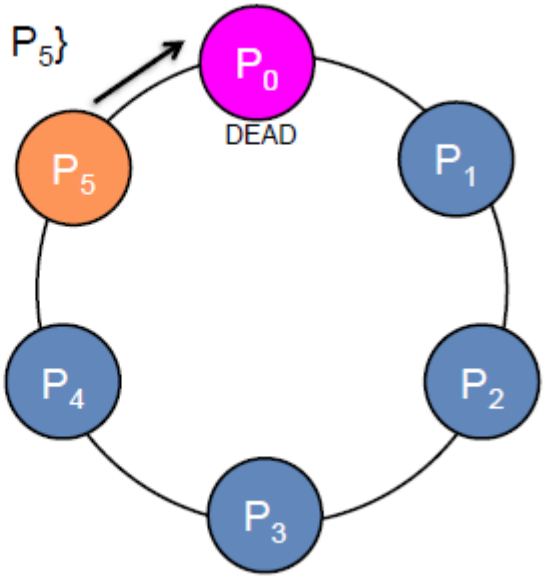



P0
DEAD
P1
P2
P3
P4
P5
Election: {P2, P3}


P0
DEAD
P1
P2
P3
P4
P5
Election: {P2, P3, P4}

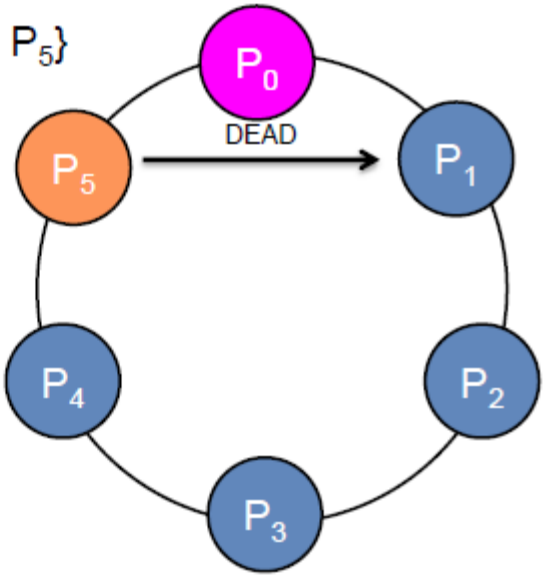
Election: {P_2, P_3, P_4, P_5}

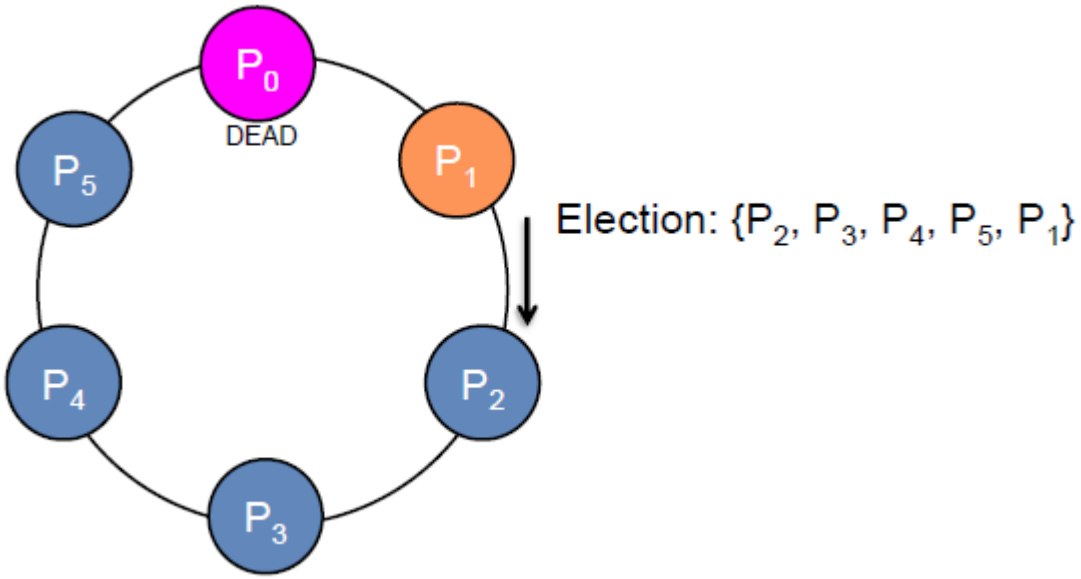
Fails: P_0 is dead



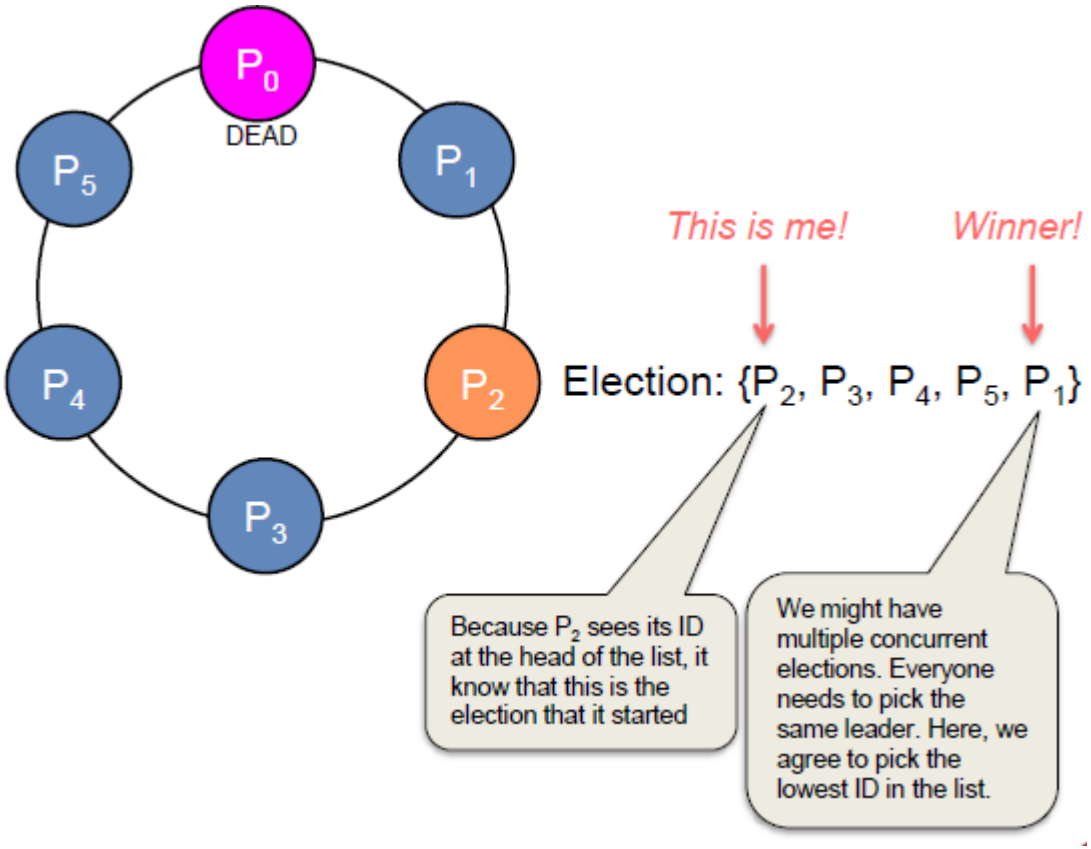
Election: {P_2, P_3, P_4, P_5}

Skip to P_1

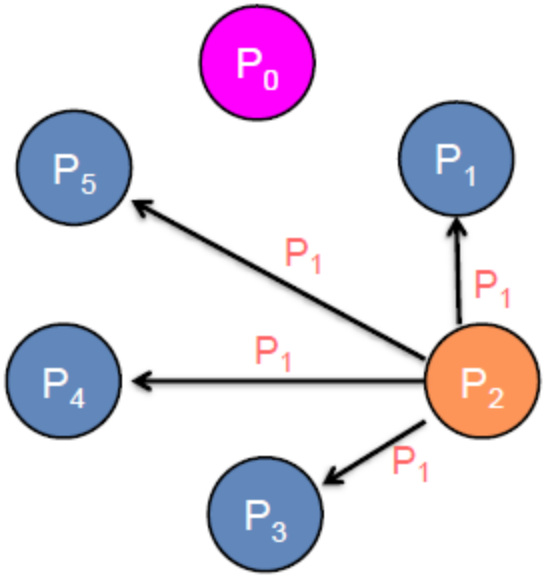



P0
DEAD
P1
P2
P3
P4
P5
Election: {P2, P3, P4, P5, P1}

- P_2 receives the election message that it initiated
- P_2 now picks a leader (e.g., lowest or highest ID)



- P_2 announces the new coordinator to the group

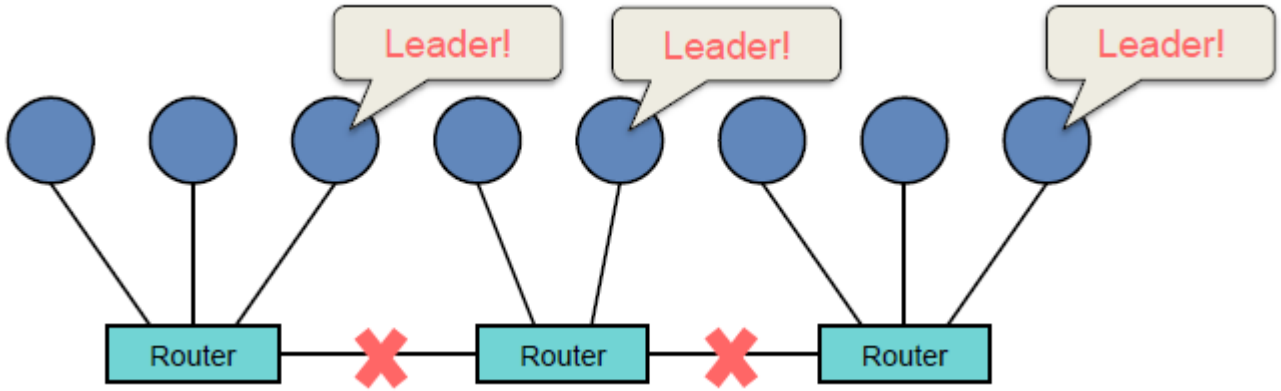


Chang & Robert ring algorithm

- Optimize the ring
  - Message always contains *one process ID*
  - Avoid multiple circulating elections
  - If a process sends a message, it marks its state as a *participant*
- Upon receiving an election message:
  - If PID(message) > PID(process)
    - forward the message
  - If PID(message) < PID(process)
    - replace PID in message with PID(process)
    - forward the new message
  - If PID(message) < PID(process) AND process is *participant*
    - discard the message
  - If PID(message) == PID(process)
    - the process is now the leader

Split brain

- Network **partitioning (segmentation)**
  - **Split brain**
  - Multiple nodes may decide they're the leader



- Dealing with partitioning
  - Insist on a majority → if no majority, the system will not function
  - Rely on alternate communication mechanism to validate failure
    - Redundant network, shared disk, serial line, SCSI