Distributed Systems
分布式系统

Mutual Exclusion & Election Algorithms

互斥算法 & 选举算法

Process Synchronization

- Techniques to coordinate execution among processes
  - One process may have to wait for another
  - Shared resource (e.g. critical section) may require exclusive access

Requirements of Mutual Exclusion Algorithms

A mutual exclusion algorithm should satisfy the following properties:

•**Safety Property:** The safety property states that at any instant, only one process can execute the critical section.

•**Liveness Property:** This property states the absence of deadlock and starvation.

•**Fairness:** Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the critical section.

Centralized Systems

- Achieve Mutual exclusion via:
  - Test & set in hardware
  - Semaphores
  - Messages(inter-process)
  - Condition variables

Distributed Mutual Exclusion

Assume there is agreement on how a resource is identified

- Pass identifier with requests
- e.g., lock(“printer”), lock(“table:employees”),
  lock(“table:employees;row:15”)

…and every process can identify itself uniquely

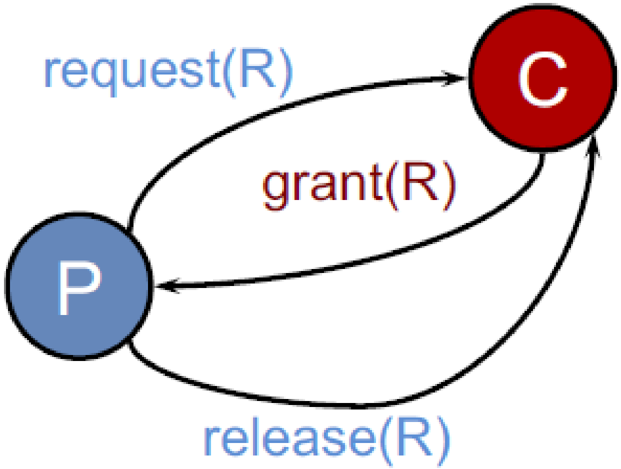
Goal:

Create an algorithm to allow a process to obtain exclusive access to a resource that is available on the network

Centralized algorithm（集中式算法）

- Mimic single processor system
- One process elected as coordinator

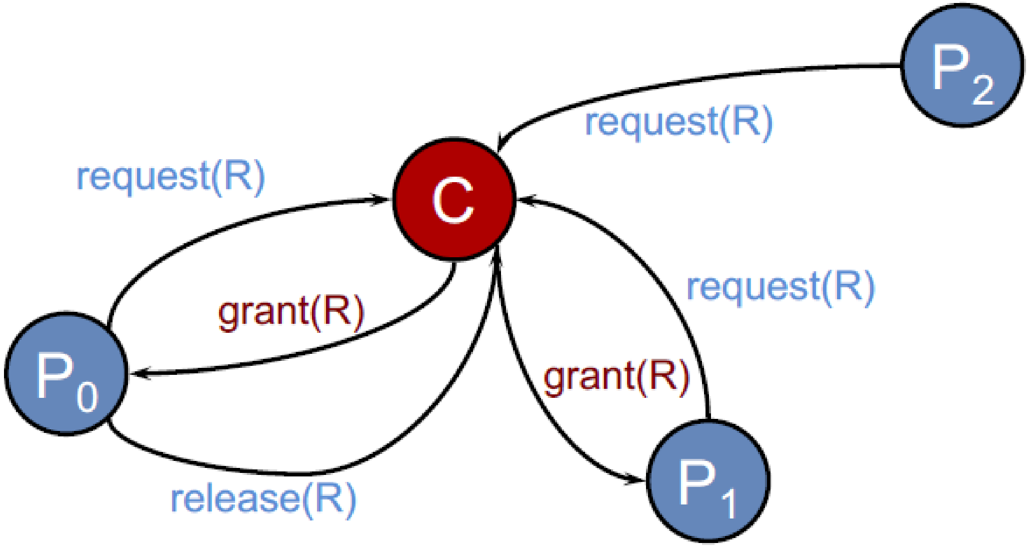
1. **Request** resource
2. Wait for response
3. **Receive grant**
4. Access resource
5. **Release resource**



Centralized algorithm （集中式算法）

- If another process claimed resource:
  - Coordinator does not reply until release
  - Maintain queue

    Service requests in FIFO order



Centralized algorithm （集中式算法）

Benefits

- Fair: All requests processed in order
- Easy to implement, understand, verify
- Processes do not need to know group members – just the coordinator

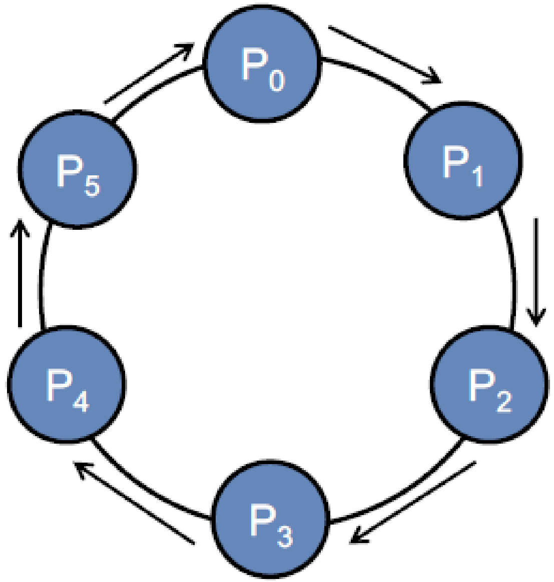
Problems

- Process cannot distinguish being blocked from a dead coordinator – single point of failure
- Centralized server can be a bottleneck

Token Ring algorithm（令牌环算法）

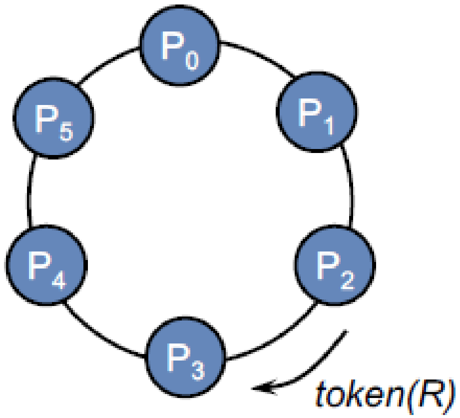
Assume known group of processes

- Some ordering can be imposed on group (unique process IDs)
- Construct logical ring in software
- Process communicates with its neighbor

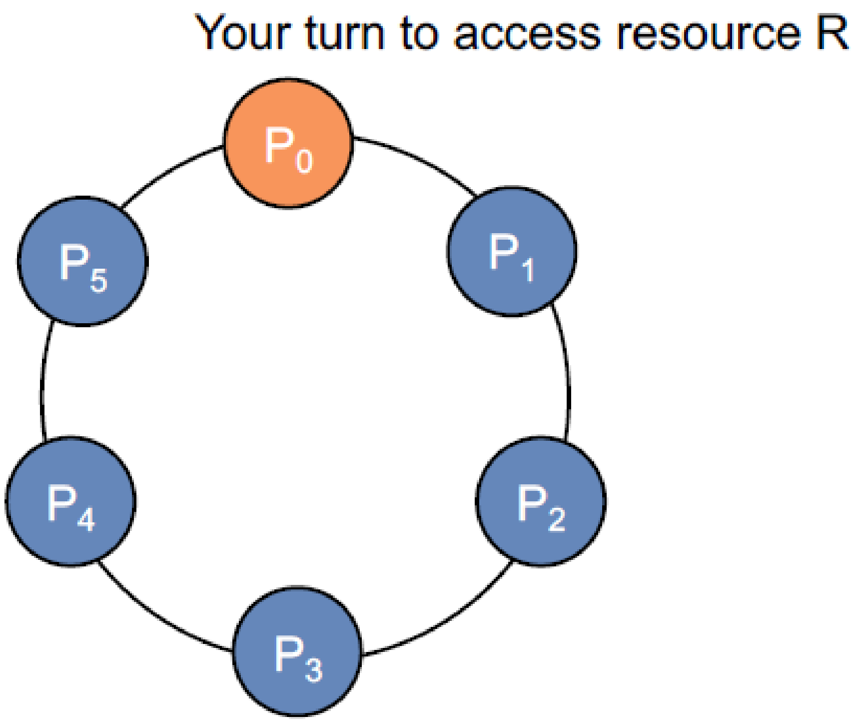


Token Ring algorithm （令牌环算法）

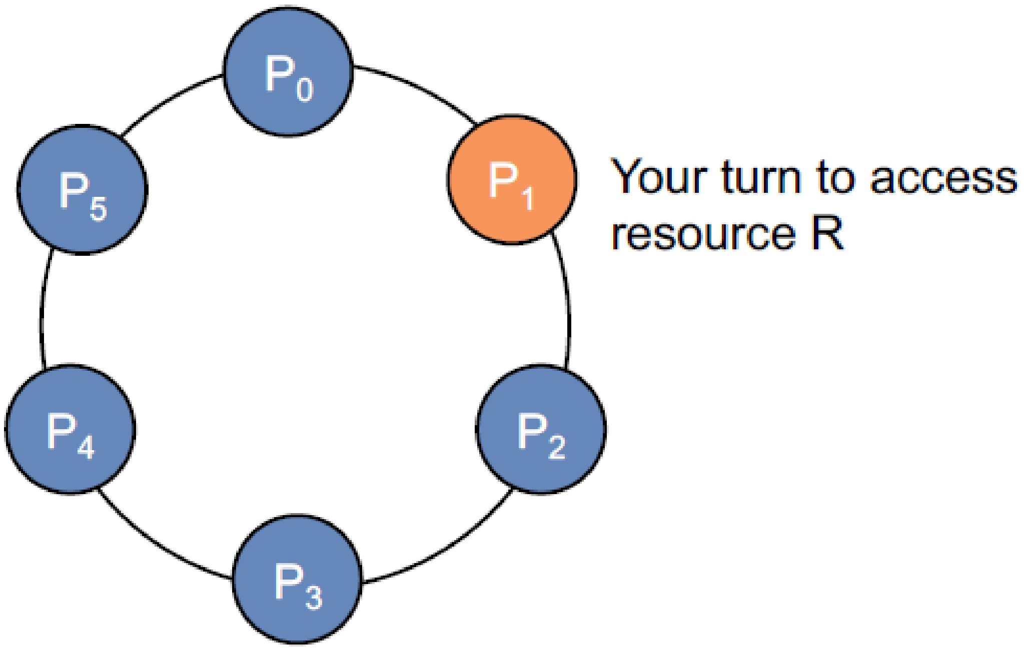
- Initialization
  - Process 0 creates a token for resource R
- Token circulates around ring
  - From P_i to $P_{(i+1)}$mod N
- When process acquires token
  - Checks to see if it needs to enter critical section
  - If no, send ring to neighbor
  - If yes, access resource
    - Hold token until done



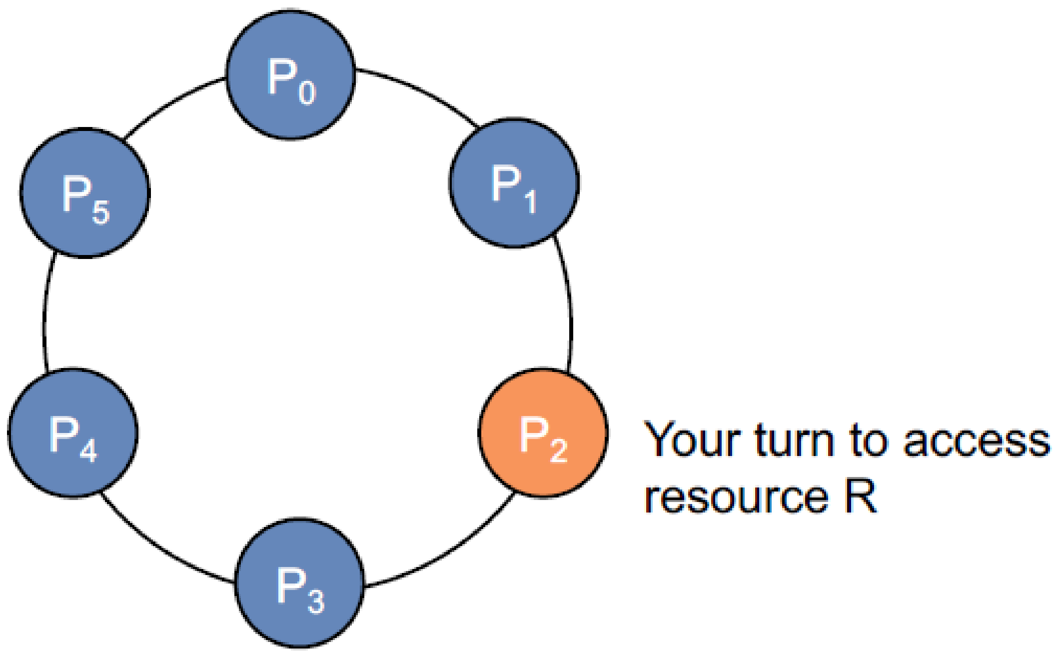
Token Ring algorithm （令牌环算法）



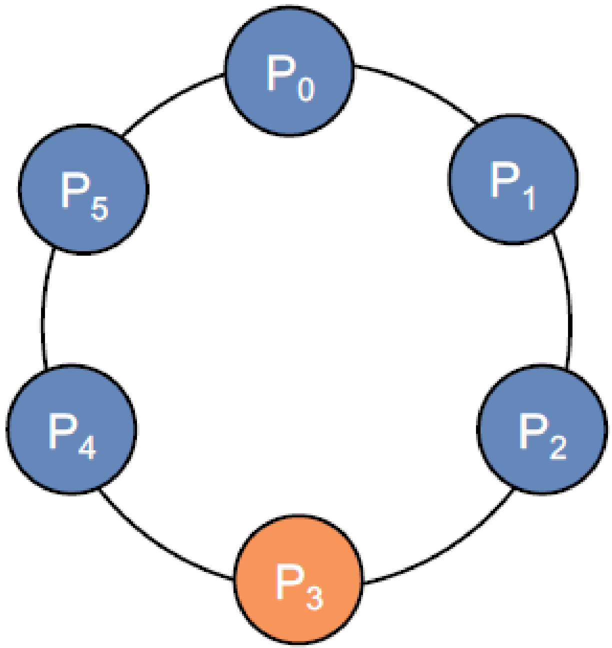
Token Ring algorithm （令牌环算法）



Token Ring algorithm （令牌环算法）

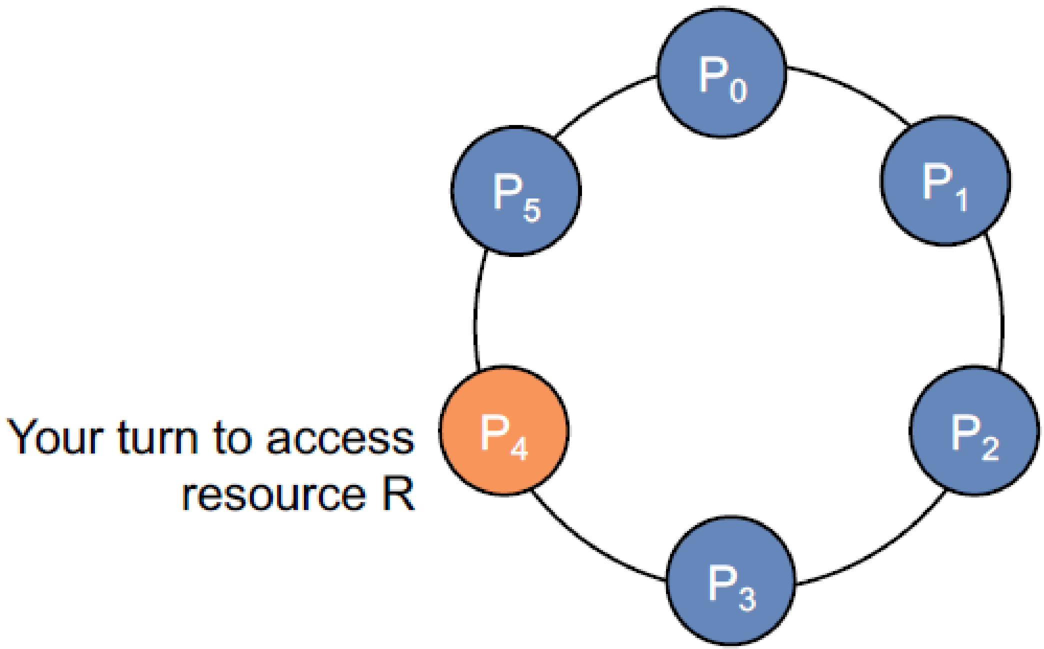


Token Ring algorithm （令牌环算法）

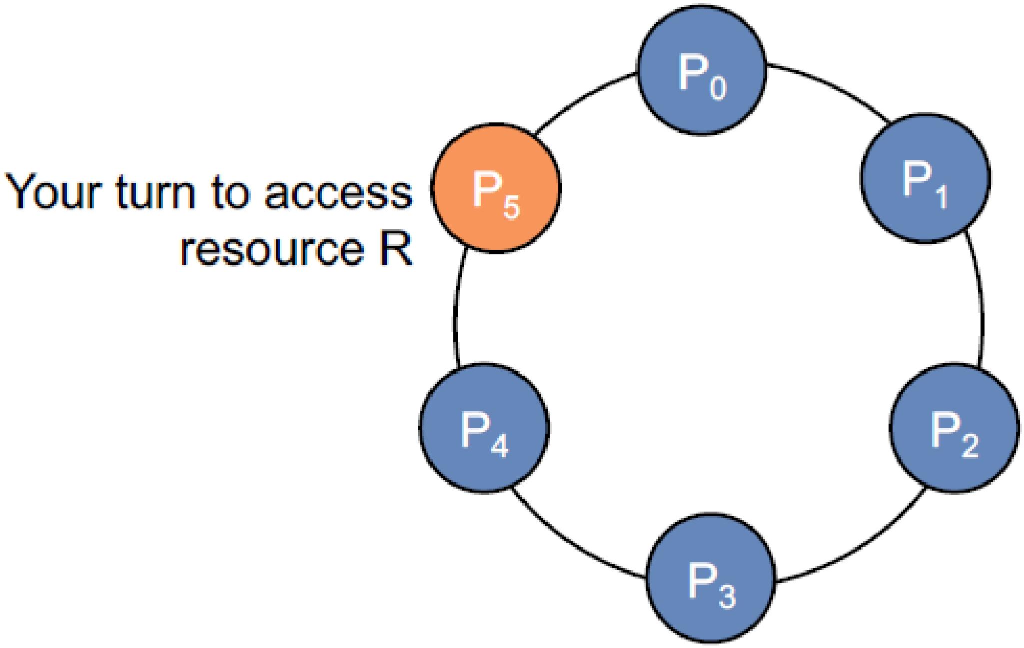


Your turn to access resource R

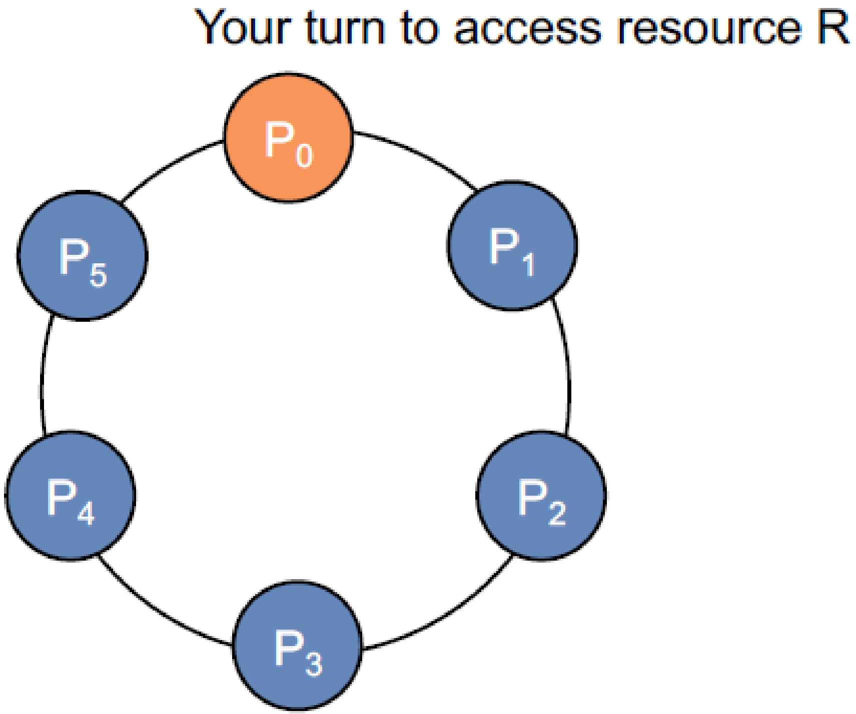
Token Ring algorithm （令牌环算法）



Token Ring algorithm （令牌环算法）



Token Ring algorithm （令牌环算法）



Token Ring algorithm summary

- 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
- Problems
  - Token loss (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?*)
  - Process loss: what if you can't talk to your neighbor?

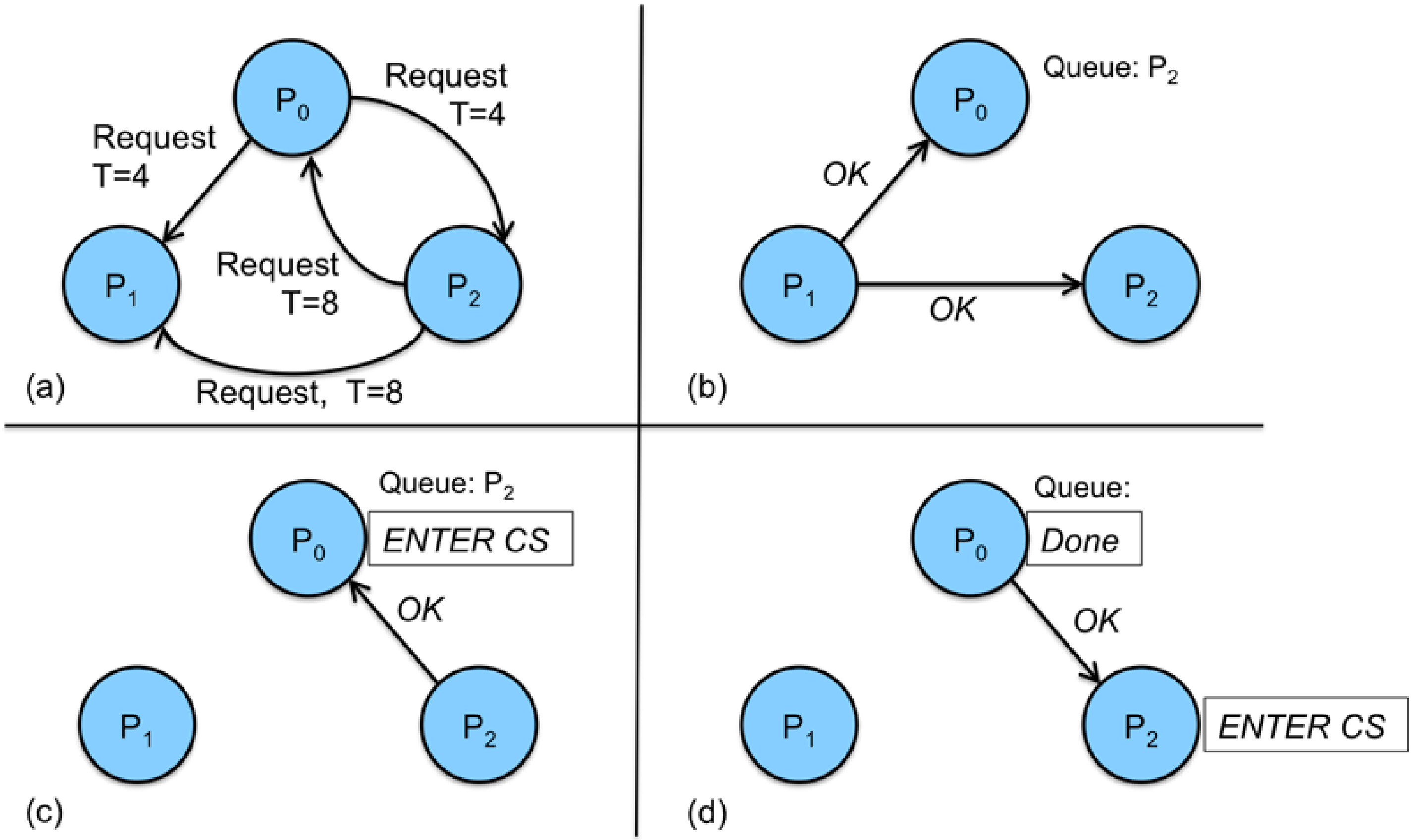
Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks
- Process wants to enter critical section:
  - Compose message containing:
    - Identifier (machine ID, process ID)
    - Name of resource
    - Timestamp (totally-ordered Lamport）
  - Send request to all processes in group
  - Wait until everyone gives permission
  - 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**:
    - Compare timestamps: received & sent messages
    - Earliest wins
    - If receiver is loser, send **OK**
    - If receiver is winner, do not reply, **queue**
  - **When done** with critical section
    - Send **OK** to **all queued requests**

Example for Ricart & Agrawala algorithm



Ricart & Agrawala algorithm

- N points of failure
- A lot of messaging traffic
- Demonstrates that a fully distributed algorithm is possible

Lamport's Mutual Exclusion

- Each process maintains request queue
  - Contains mutual exclusion requests

- <u>Requesting critical section</u>
  - Process Pi sends request(i, Ti) to all nodes
  - Places request on its own queue
  - When a process Pj receives a request, it returns a timestamped ack

Lamport time

Lamport's Mutual Exclusion

Entering critical section (accessing resource):

- Pi received a message (ack or release) from every other process with a timestamp larger than Ti
- Pi's request has the earliest timestamp in its queue

Difference from Ricart-Agrawala:

- Everyone responds (acks)… always - no hold-back
- Process decides to go based on whether its request is the earliest in its queue

Lamport's Mutual Exclusion

Releasing critical section

- Remove request from its own queue
- Send a timestamped release message

- When a process receives a release message
  - Removes request for that process from its queue
  - This may cause its own entry have the earliest timestamp in the queue, enabling it to access the critical section

Election algorithms (选举算法）

Elections

- Need one process to act as coordinator
- Processes have no distinguishing characteristics
- Each process has a unique ID to identify itself

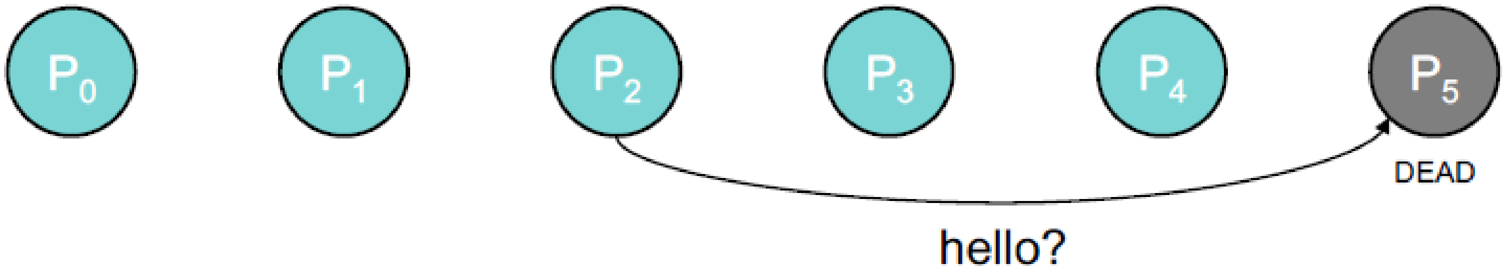
Bully algorithm （霸道选举算法)

- 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 （霸道选举算法)

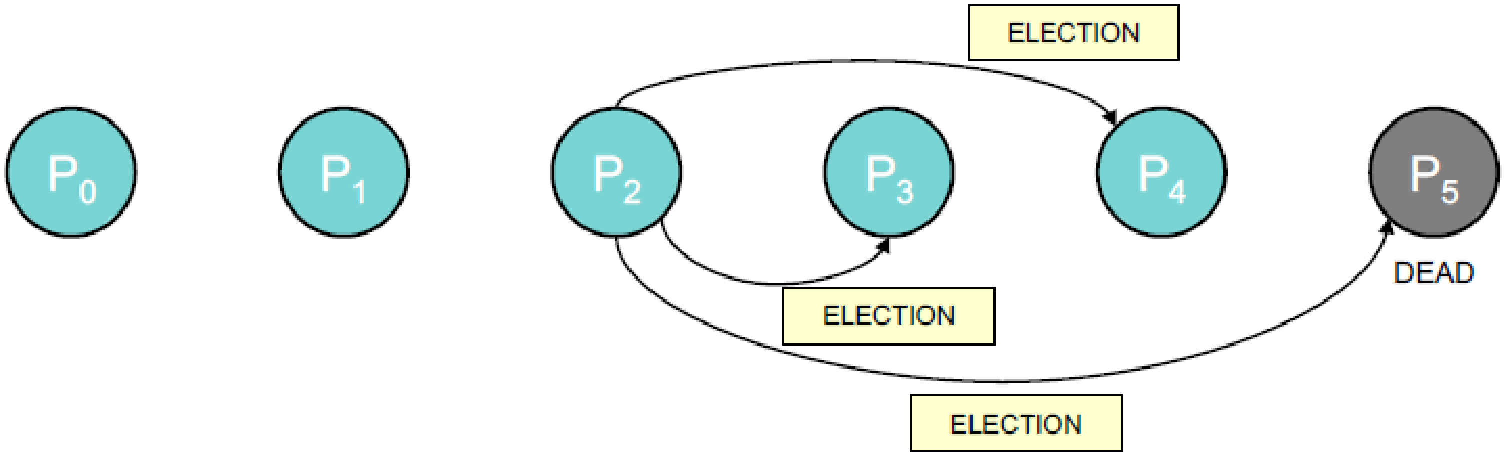
- A process announces victory by sending all processes a message telling them that it is the new coordinator
- If a dead process recovers, it holds an election to find the coordinator.

Bully algorithm （霸道选举算法)



Rule: highest # process is the leader
Suppose P_5 dies
P_2 detects P_5 is not responding

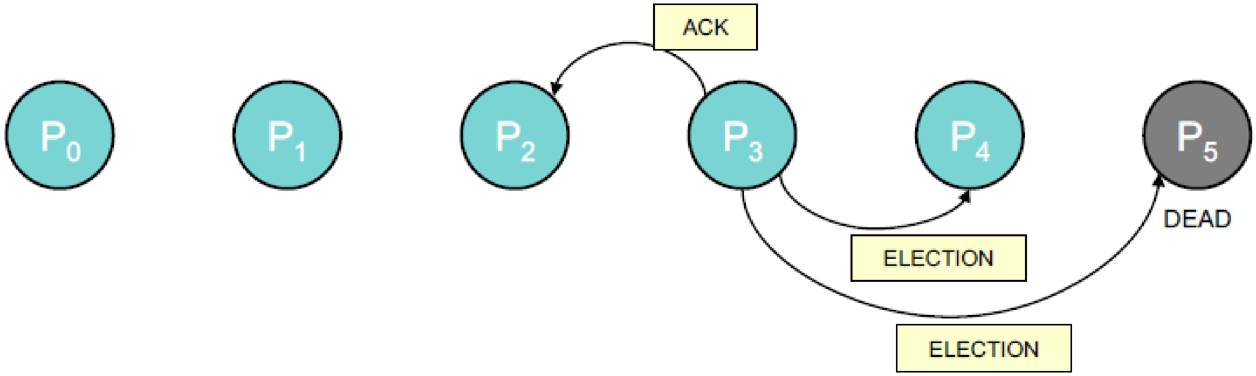
Bully algorithm （霸道选举算法)



P_2 starts an election

Contacts all higher-numbered systems

Bully algorithm （霸道选举算法)



Everyone who receives an ELECTION message responds

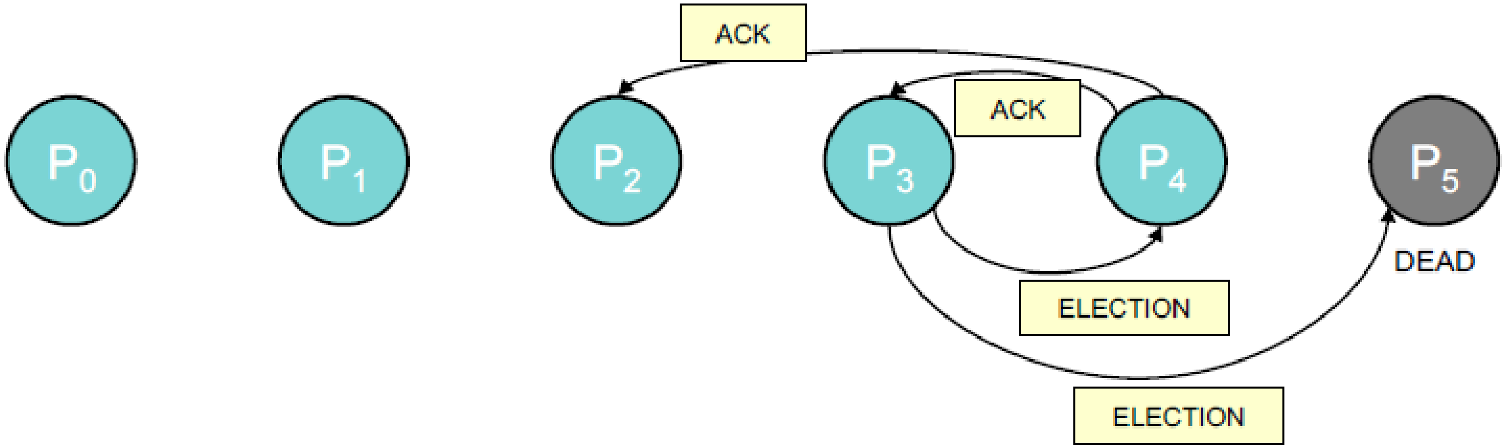
… and holds their own election, contacting higher # processes

Example: P_3 receives the message from P_2

Responds to P_2

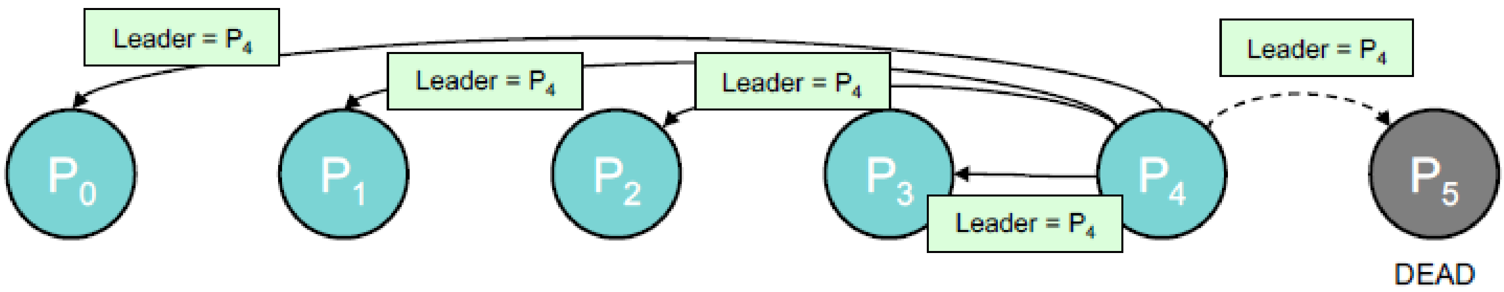
Sends ELECTION messages to P_4 and P_5

Bully algorithm （霸道选举算法)



P_4 responds to P_3 and P_2's messages
… and holds an election

Bully algorithm （霸道选举算法)



Nobody responds to P_4

After a timeout, P_4 declares itself the leader

Ring algorithm （环选举算法）

- 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 （环选举算法）

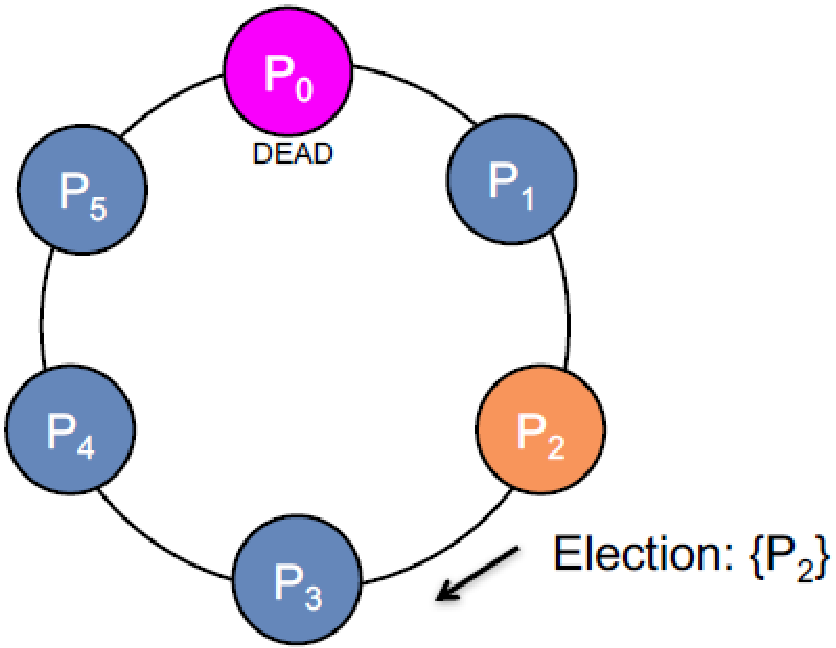
Eventually message returns to originator

- Process sees its ID on list
- Circulates (or multicasts) a **coordinator** message announcing coordinator

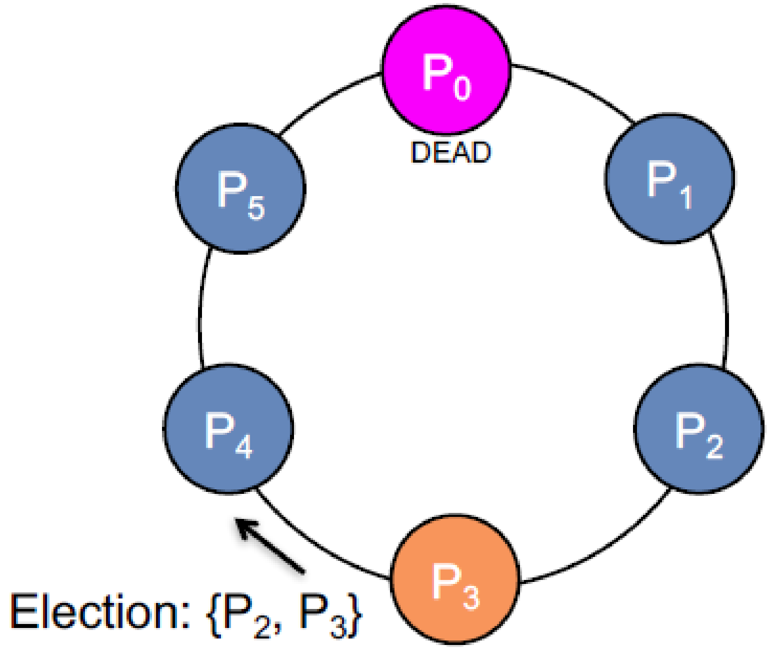
- E.g. lowest numbered process

Ring algorithm （环选举算法）

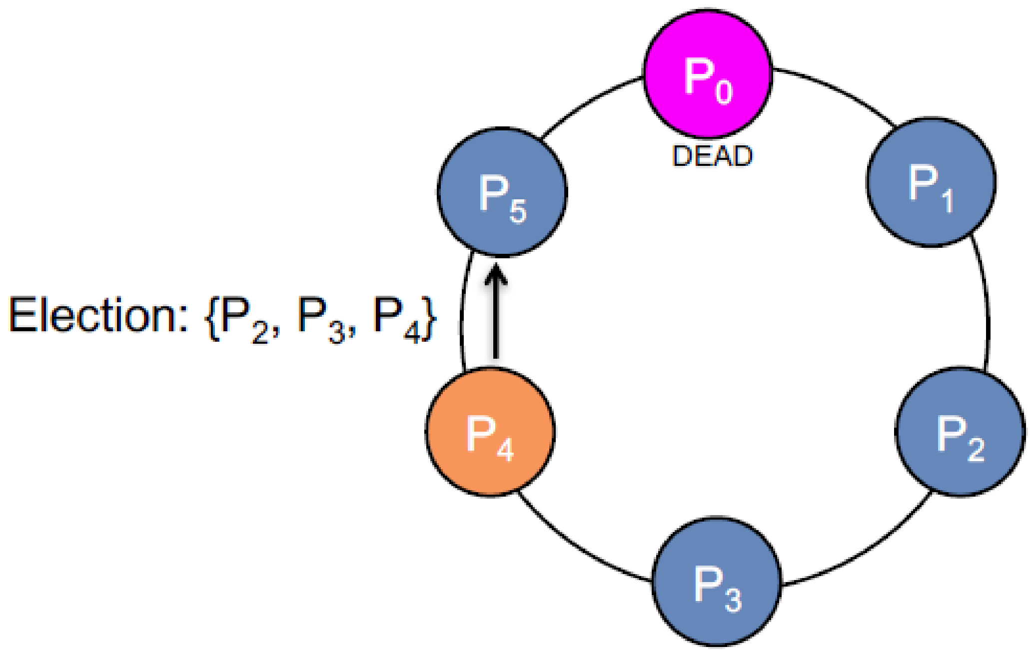
Assume P_2 discovers that the coordinator, P_0, is dead
P_2 starts an election



Ring algorithm （环选举算法）



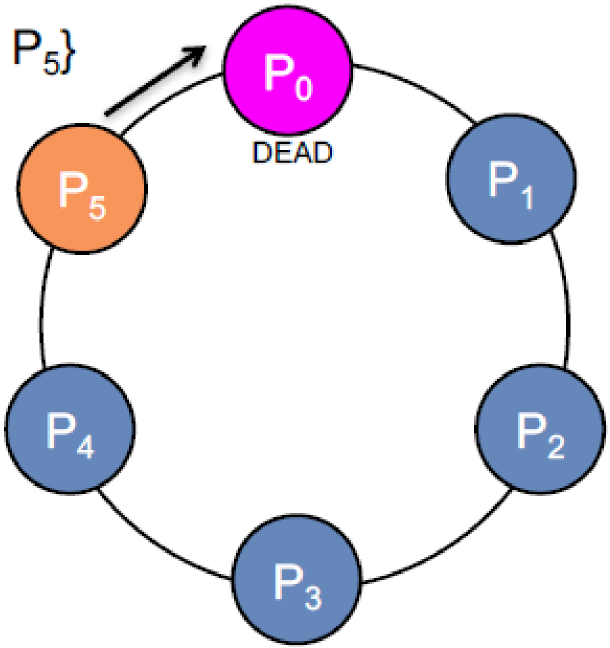
Ring algorithm （环选举算法）



Ring algorithm （环选举算法）

Election: {P_2, P_3, P_4, P_5}

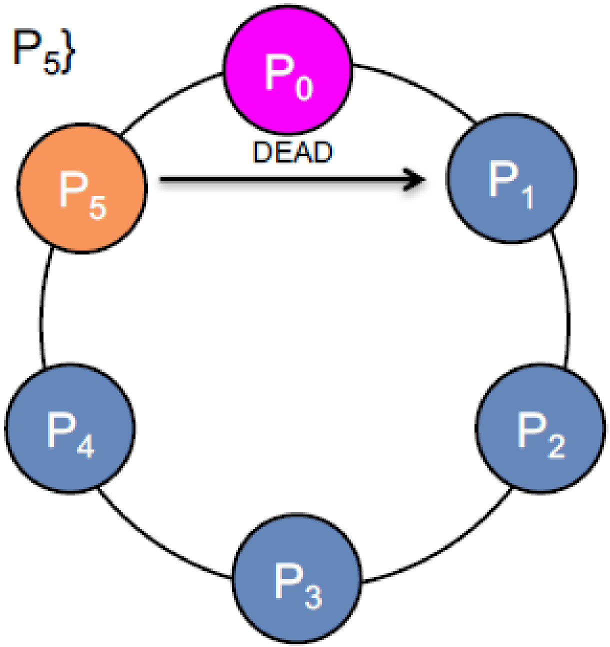
Fails: P_0 is dead



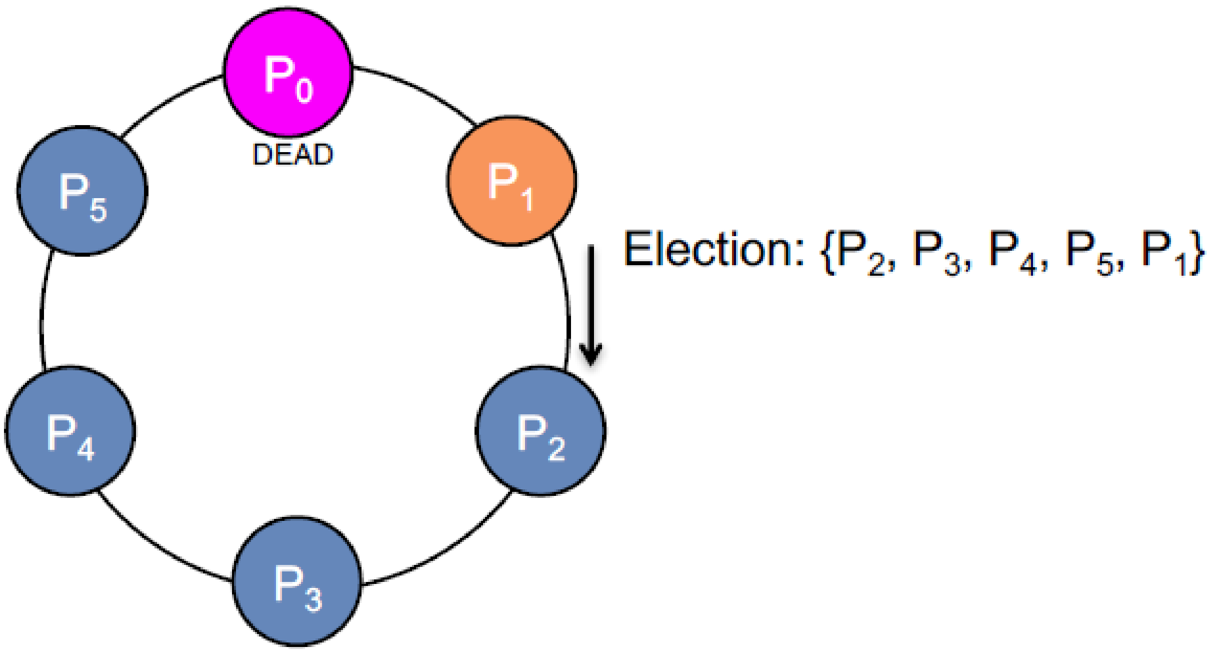
Ring algorithm （环选举算法）

Election: {P_2, P_3, P_4, P_5}

Skip to P_1



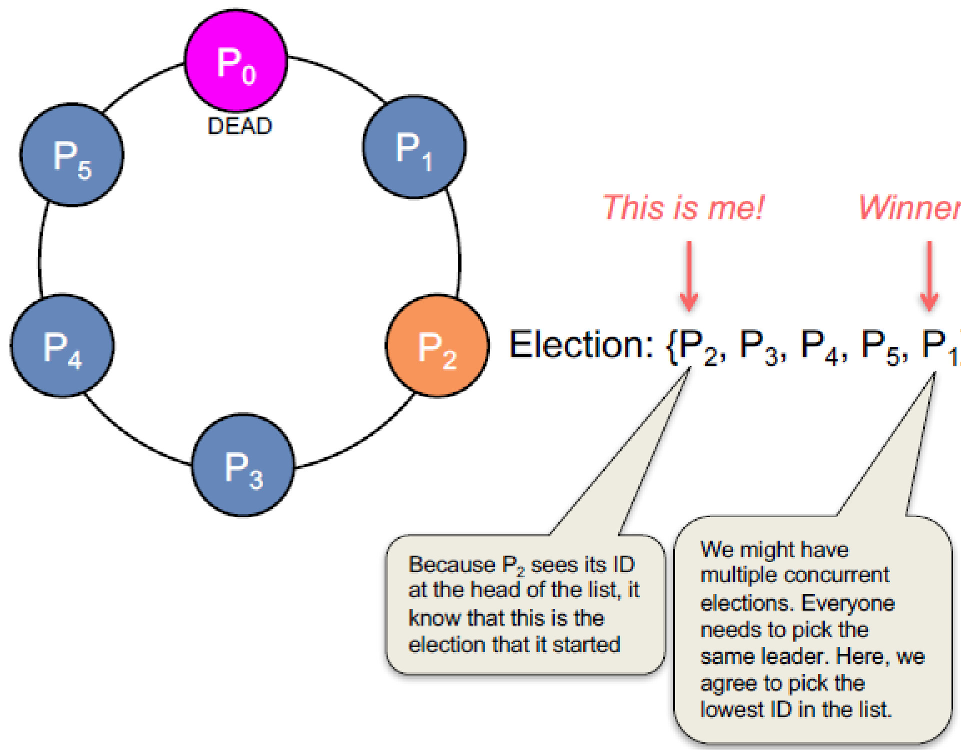
Ring algorithm （环选举算法）



Ring algorithm （环选举算法）

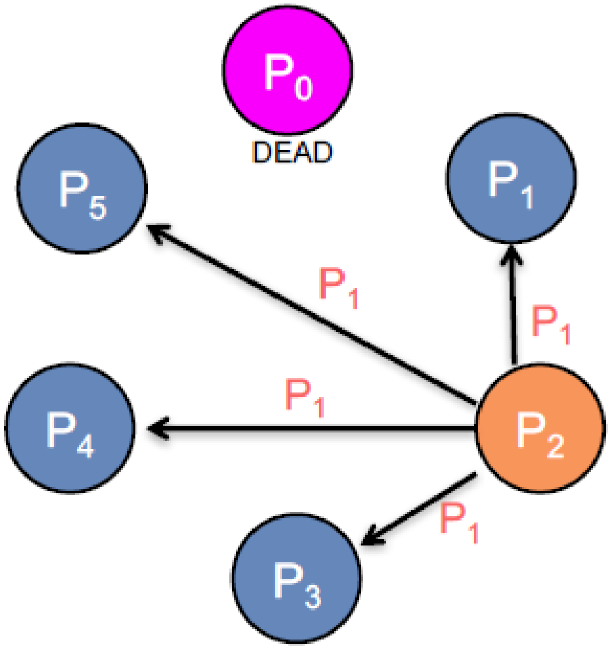
P_2 receives the election message that it initiated

P_2 now picks a leader (e.g., lowest or highest ID)



Ring algorithm （环选举算法）

P_2 announces the new coordinator to the group



Chang & Roberts 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 – *higher ID will always win over a lower one*

If PID(message) < PID(process)

replace PID in message with PID(process)

forward the new message – *we have a higher ID number; use it*

If PID(message) < PID(process) AND process is *participant*

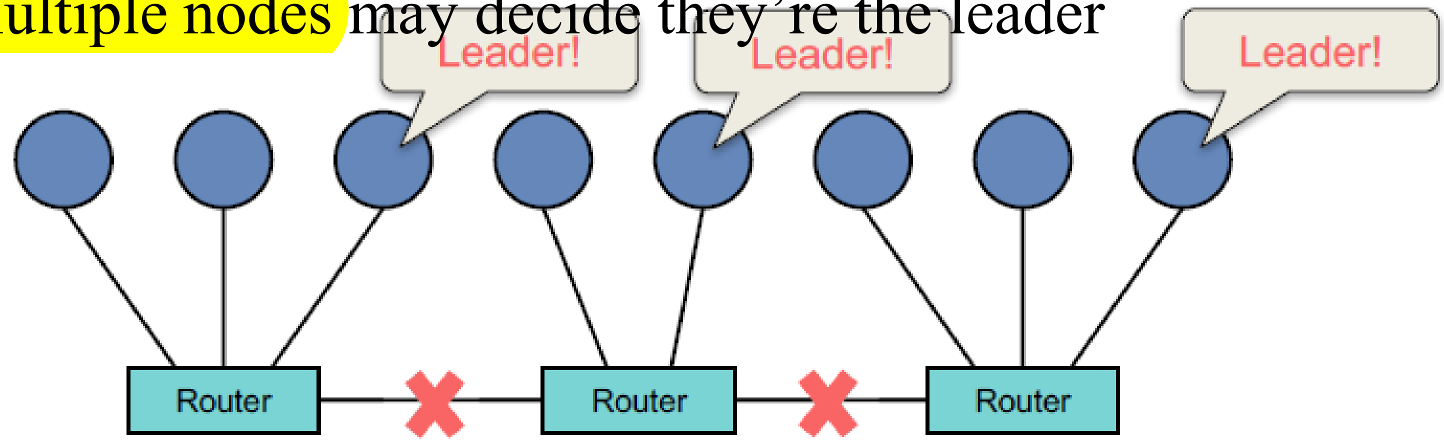
discard the message – *we're already circulating our ID*

If PID(message) == PID(process)

the process is now the leader – *message circulated: announce winner*

Network Partitioning: Split Brain （脑裂）

- 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