Distributed Systems

08. Mutual Exclusion & Election Algorithms

Paul Krzyzanowski

Rutgers University

Fall 2016

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

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 the identifier with requests
  - e.g., *lock("printer"), lock("table:employees"), lock("table:employees;row:15")*

- Goal:
  Create an algorithm to allow a process to request and obtain exclusive access to a resource that is available on the network.

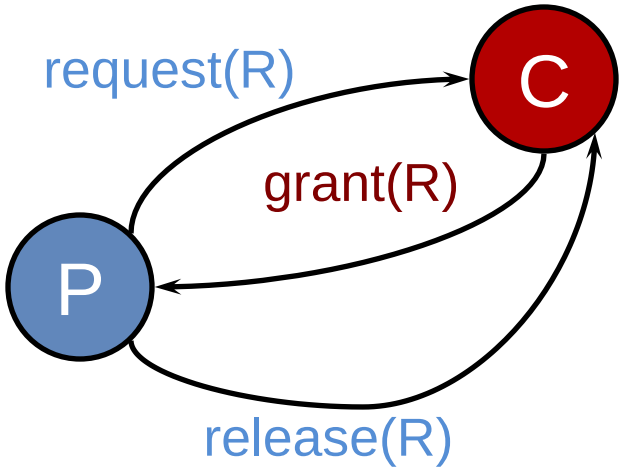
Categories of algorithms

- Centralized
  - A process can access a resource because a central coordinator allowed it to do so
- Token-based
  - A process can access a resource if it is holding a token permitting it to do so
- Contention-based
  - An process can access a resource via distributed agreement

Centralized algorithm

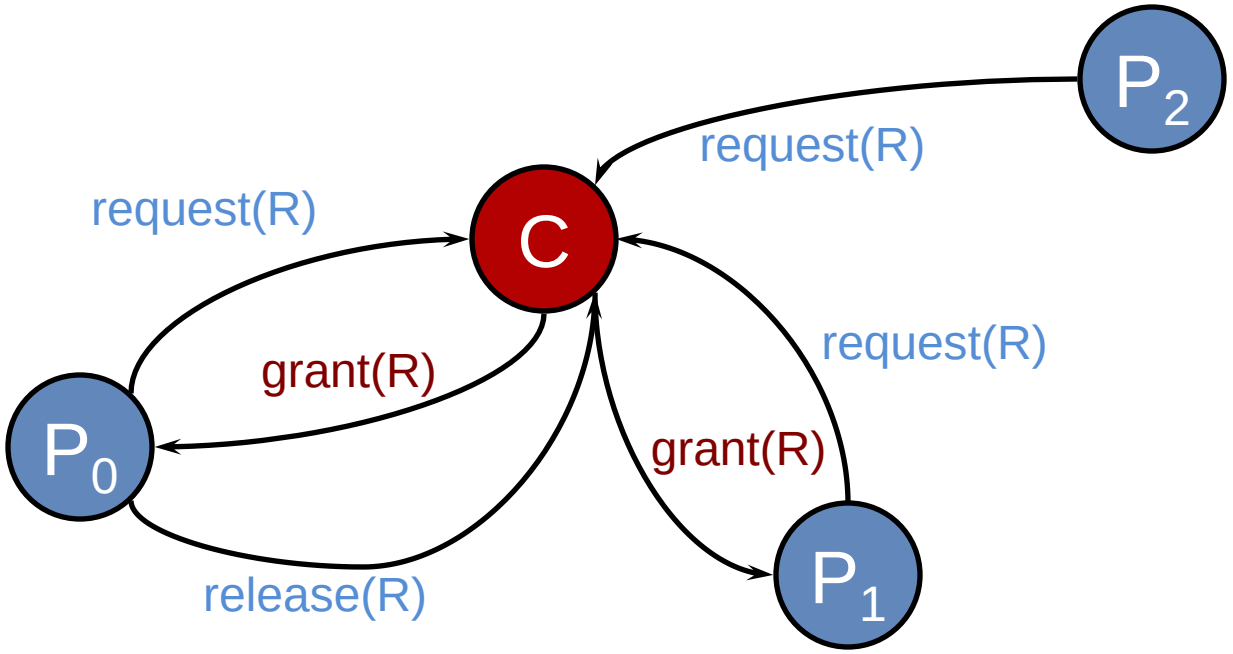
- Mimic single processor system
- One process elected as coordinator

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



Centralized algorithm

- If another process claimed resource:
  - Coordinator does not reply until release
  - Maintain queue
    - Service requests in FIFO order



Centralized algorithm

Benefits

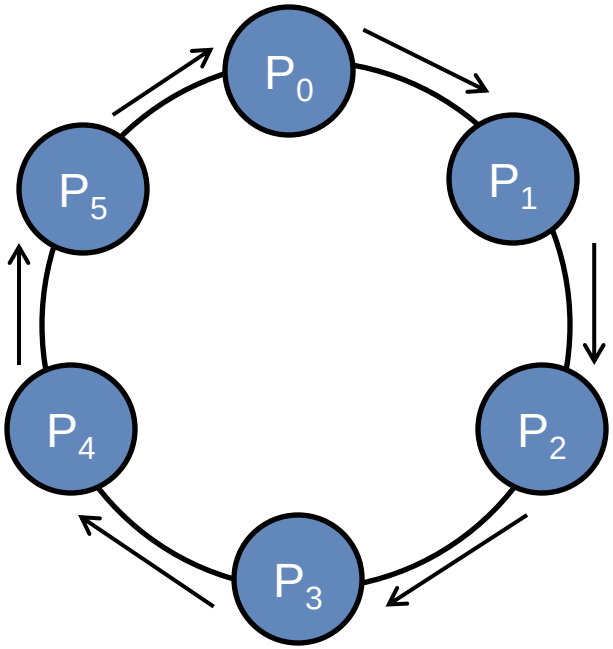
- Fair: All requests processed in order
- Easy to implement, understand, verify

Problems

- Process cannot distinguish being blocked from a dead coordinator
- Centralized server can be a bottleneck

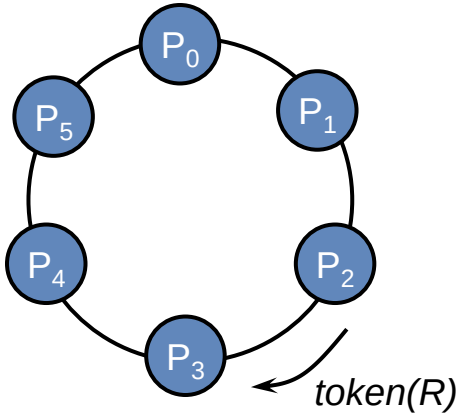
Token Ring algorithm

- 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



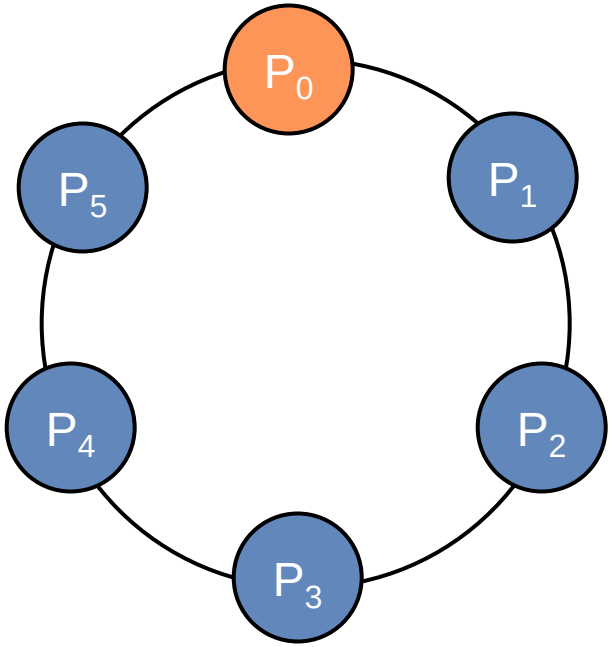
Token Ring algorithm

- 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

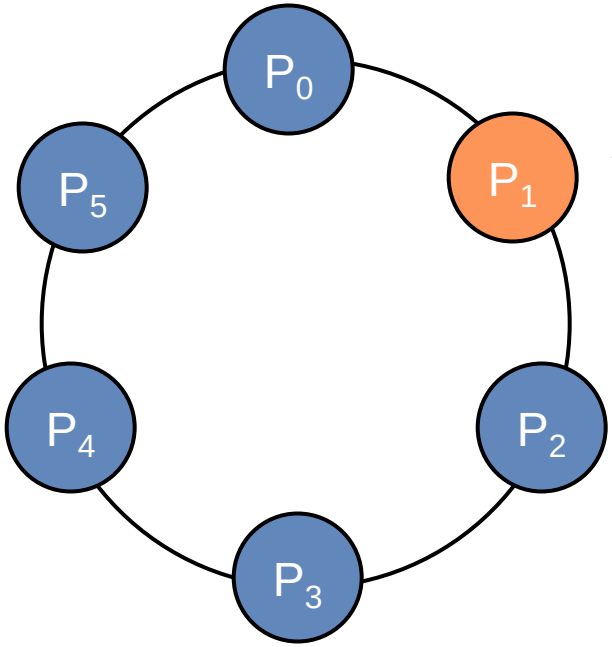


Token Ring algorithm

Your turn to access resource R

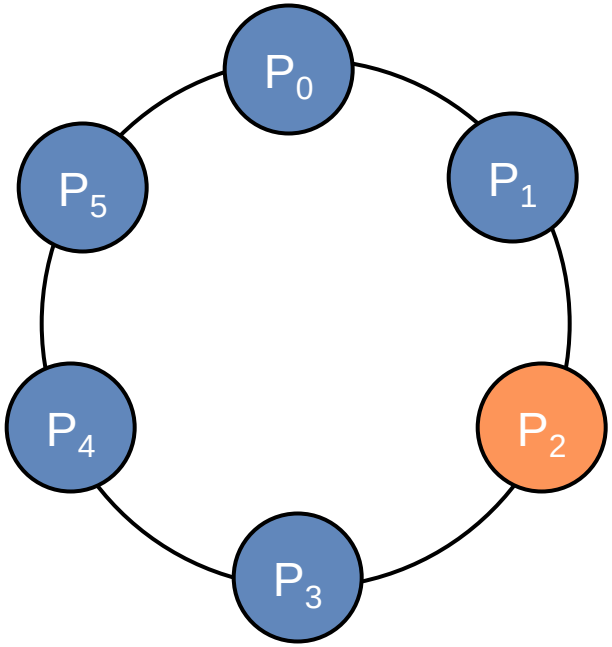


Token Ring algorithm



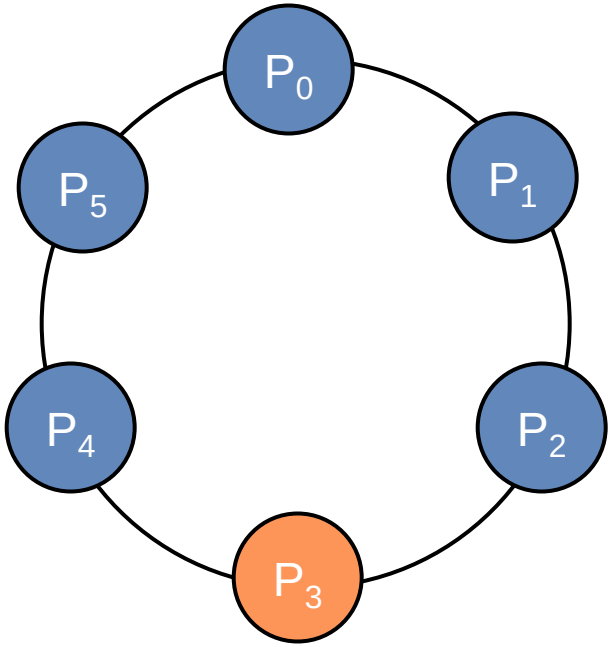
Your turn to access resource R

Token Ring algorithm



Your turn to access resource R

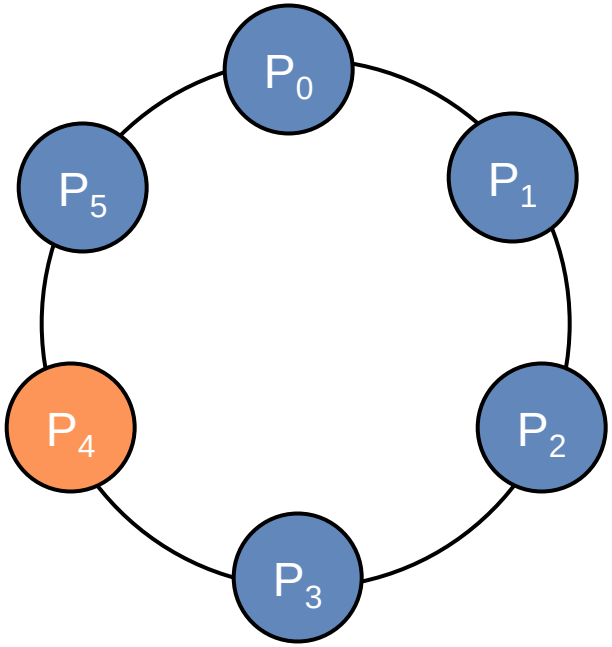
Token Ring algorithm



Your turn to access resource R

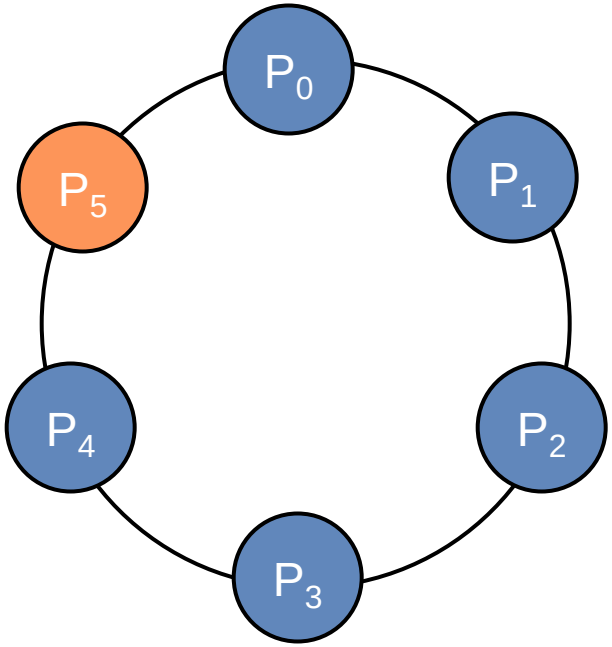
Token Ring algorithm

Your turn to access resource R



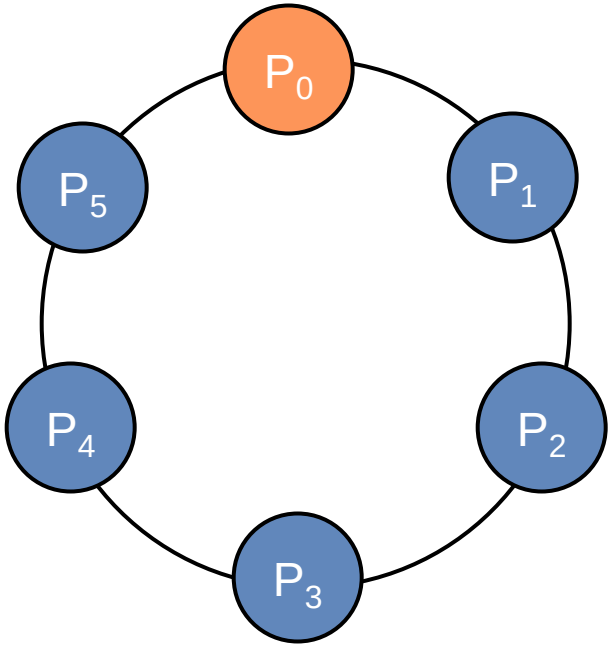
Token Ring algorithm

Your turn to access resource R

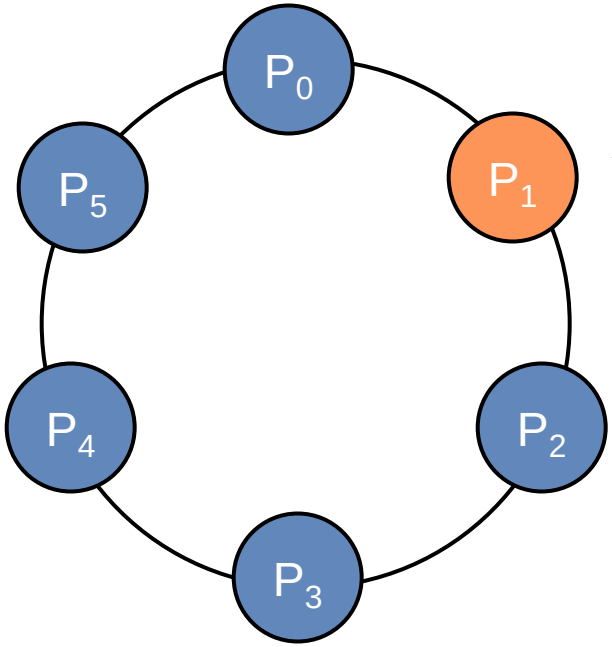


Token Ring algorithm

Your turn to access resource R



Token Ring algorithm



Your turn to access resouce R

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
- 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's 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's Mutual Exclusion

Lamport time

Request a critical section:

- Process P_i sends *request(i, T_i)* to all nodes
  - … and places request on its own queue
- When a process P_j receives a request:
  - It returns a timestamped *ack*
  - Places the request on its request queue

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

Enter a critical section (accessing resource):

- P_i has received acks from everyone
- P_i's request has the earliest timestamp in its queue

Release a critical section:

- Process P_i removes its request from its queue
- sends *release(i, T_i)* 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

Lamport's 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

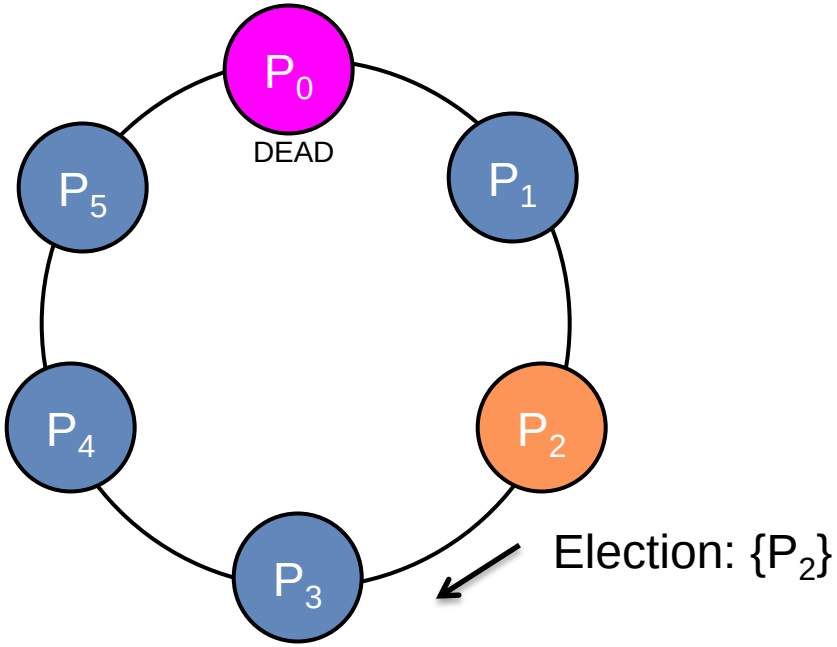
- Ring arrangement of processes
- If any process detects failure of coordinator
  - Construct election message with process ID and send to next process
  - If successor is down, skip over
  - Repeat until a running process is located
- Upon receiving an election message
  - Process forwards the message,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

Ring algorithm

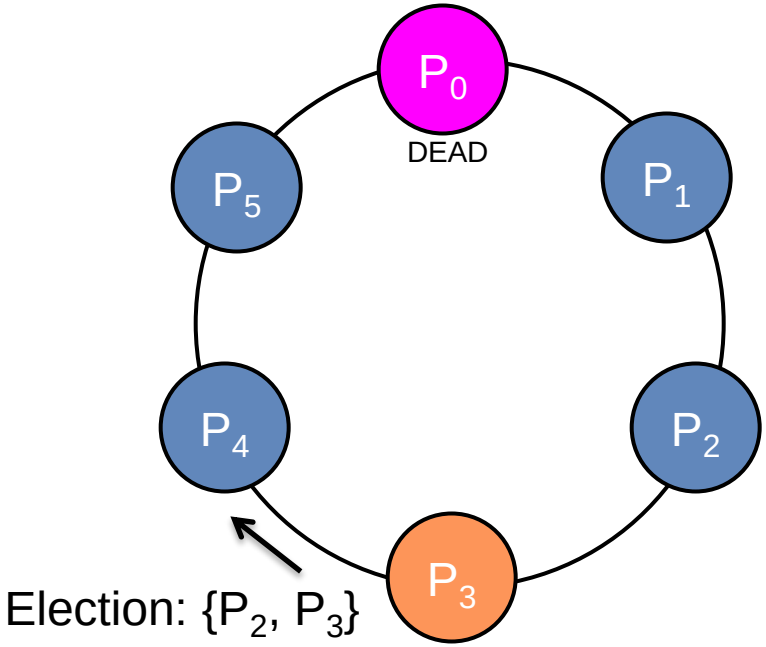
Eventually message returns to originator

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

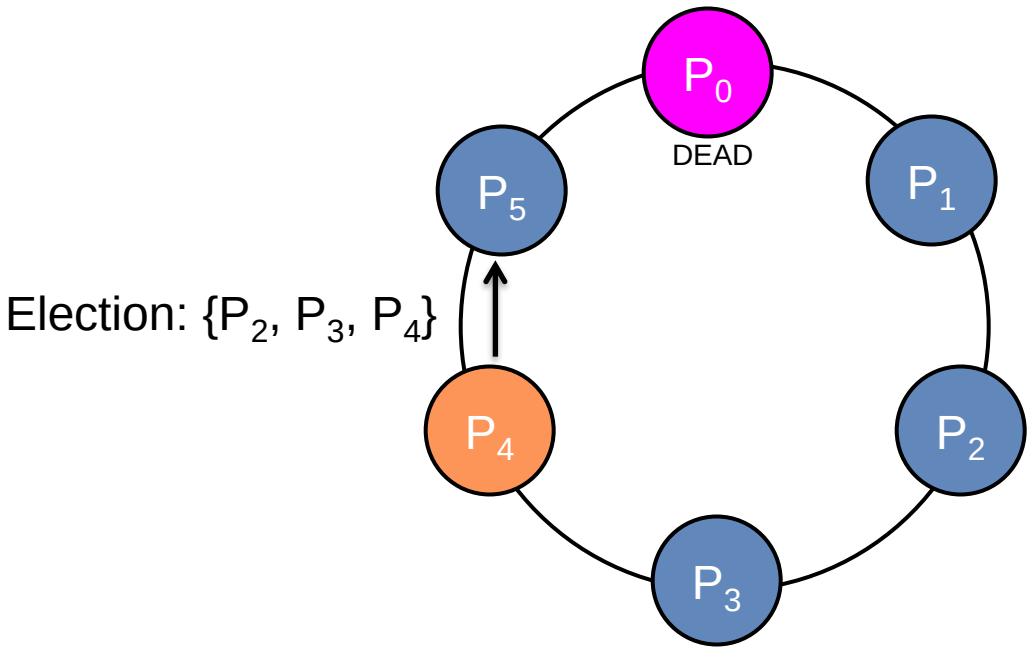
Ring algorithm



Ring algorithm



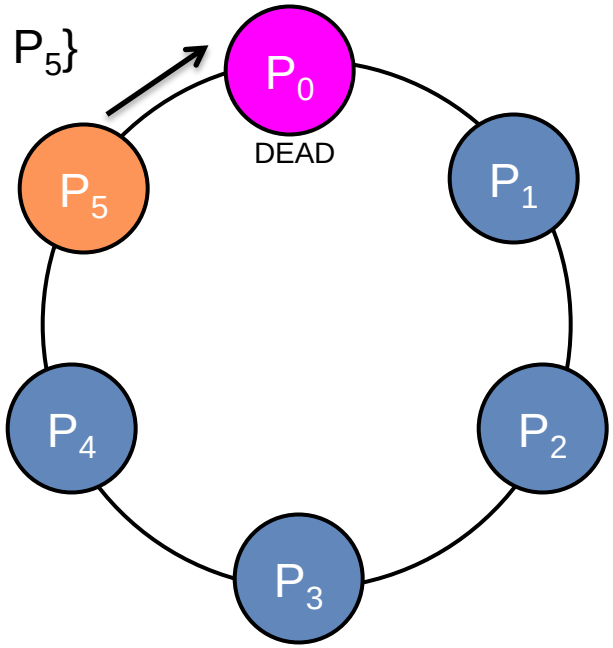
Ring algorithm



Ring algorithm

Election: {P_2, P_3, P_4, P_5}

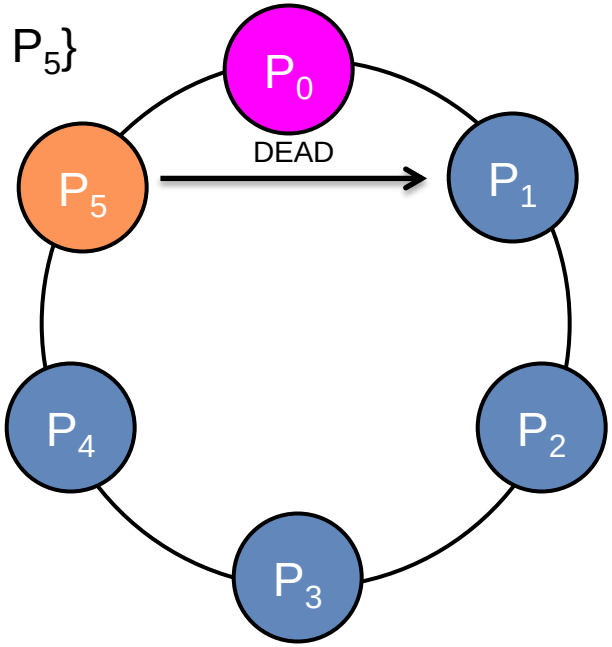
Fails: P_0 is dead



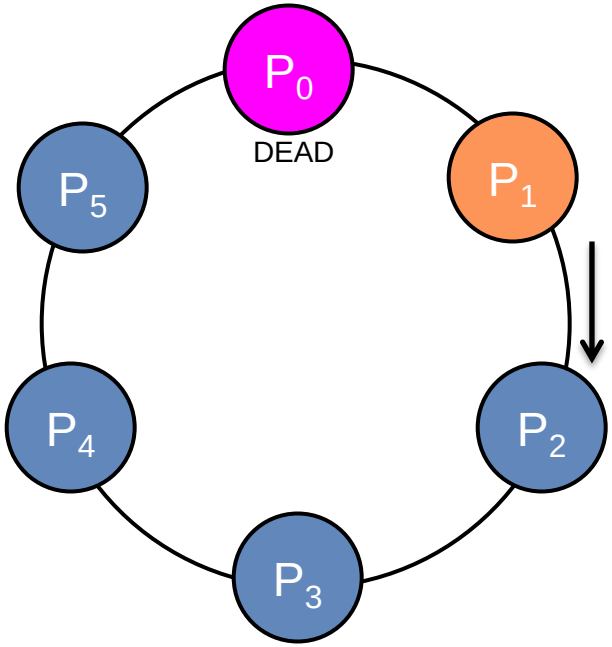
Ring algorithm

Election: {P_2, P_3, P_4, P_5}

Skip to P_1



Ring algorithm

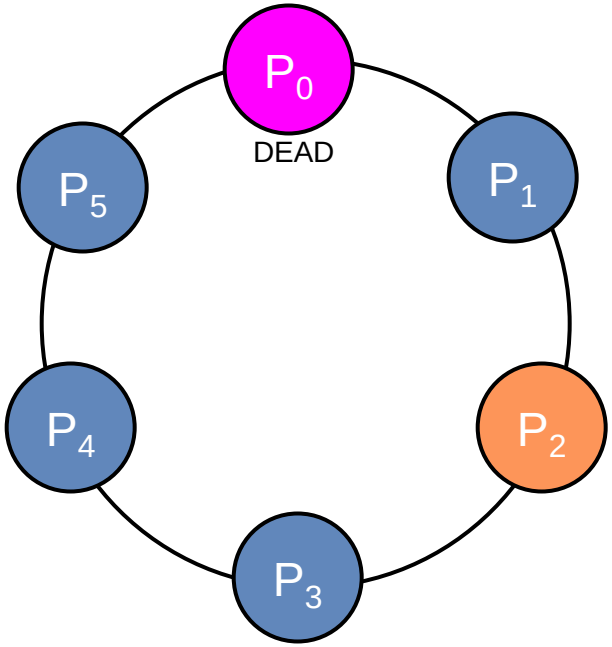


Election: {P_2, P_3, P_4, P_5, P_1}

Ring algorithm

P_2 receives the election message that it initiated

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



This is me! *Winner!*

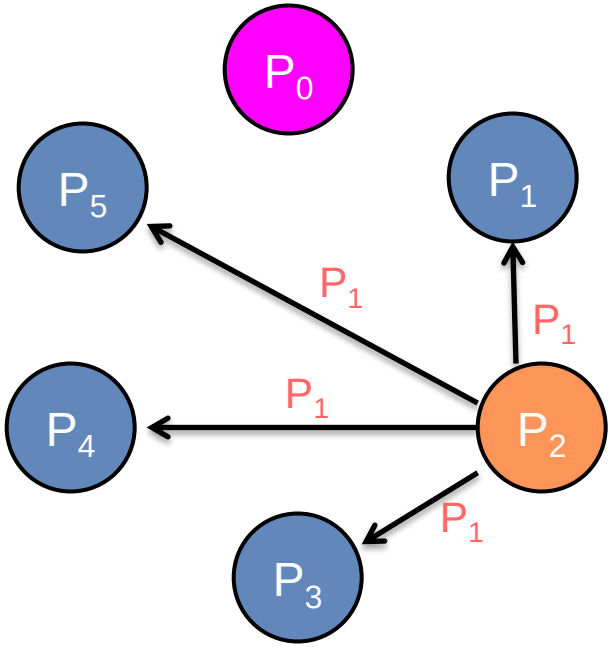
Election: {P_2, P_3, P_4, P_5, P_1}

Because P_2 sees its ID at the head of the list, it know that this is the election that it started

We might have multiple concurrent elections. Everyone needs to pick the same leader. Here, we agree to pick the lowest ID in the list.

Ring algorithm

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

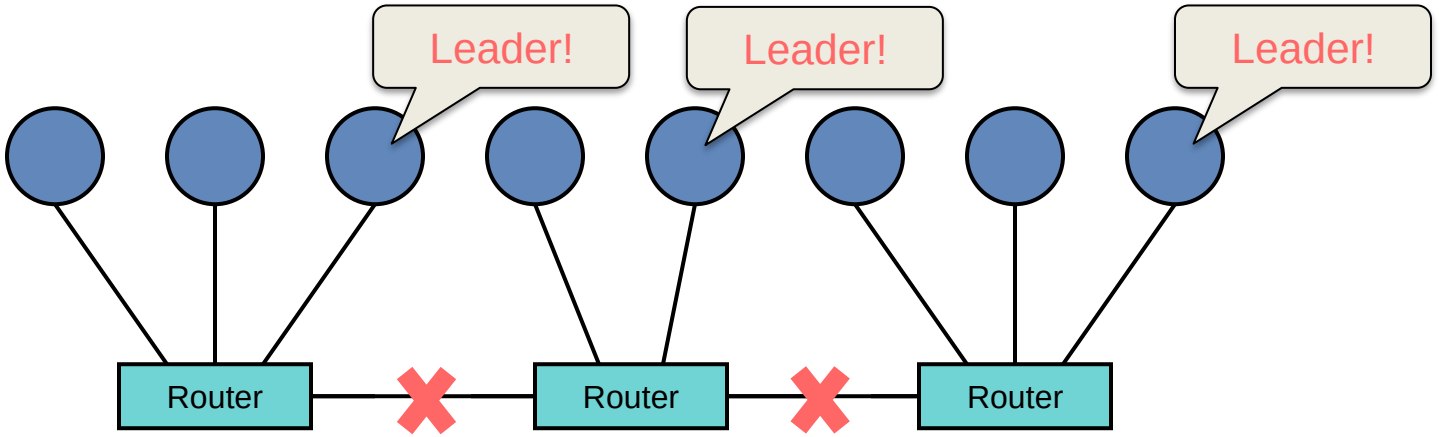
  If PID(message) < PID(process)
  replace PID in message with PID(process)
  forward the new message

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

  If PID(message) == PID(process)
  the process is now the leader

Network Partitioning: 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
- We will visit this problem later!

The End