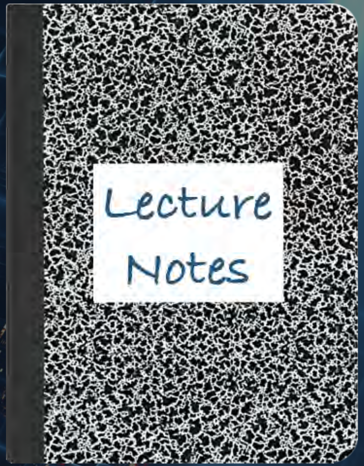


CS 417 – DISTRIBUTED SYSTEMS

Week 5: Part 1

Distributed Mutual Exclusion

Paul Krzyzanowski

Process Synchronization

Techniques to coordinate execution among processes

- One process may have to wait for another
- Shared resource (critical section) may require exclusive access

Mutual exclusion

- Examples
  - Update a fields in database tables
  - Modify a file
  - Modify file contents that are replicated on multiple servers
- Easy to handle if the entire request is atomic
  - Contained in a single message; server can manage mutual exclusion
- Needs to be coordinated if the request comprises multiple messages or spans multiple systems

Centralized Systems

Achieve mutual exclusion via:

- Test & set in hardware
- Semaphores
- Messages (inter-process)
- Condition variables

Distributed Mutual Exclusion

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

Required properties:

Safety: At any instant, only one process may hold the resource

Liveness: The algorithm should make progress; processes should not wait forever for messages that will never arrive

Also desired:

Fairness: Each process gets a fair chance to hold the resource: bounded wait time & in-order processing

Assumptions

Resource identification

- 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")*, *lock("shared_file.txt")*

  …and every process can identify itself uniquely
- We'll just use *request(R)* to request exclusive access to resource *R*

- Process identification
  - Every process has a unique ID (e.g., *address.process_id*)
- Reliable communication
  - Network messages are reliable (may require retransmission of lost/corrupted messages)
- Live processes
  - The processes in the system do not die

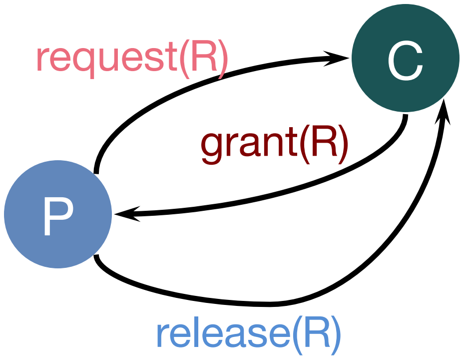
Categories of mutual exclusion algorithms

- 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
  - A 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

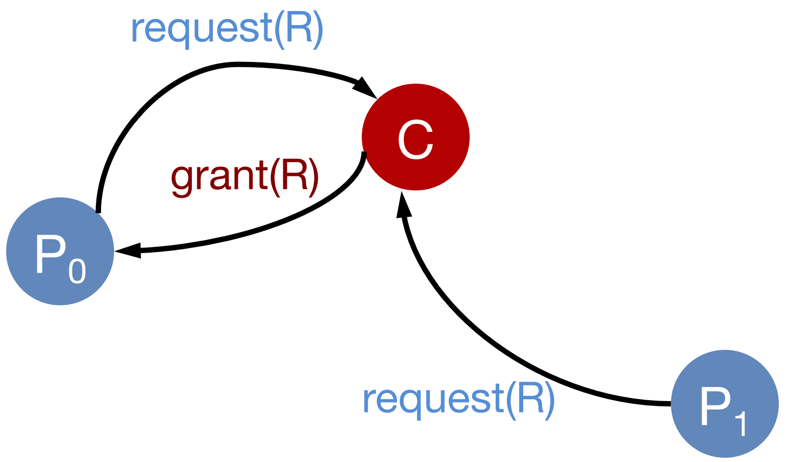
If another process claimed resource:

- Coordinator does not reply until release
- Maintain queue: service requests in FIFO order

R in use by: P_0

<u>R Request Queue</u>

P_1



Centralized algorithm

If another process claimed resource:

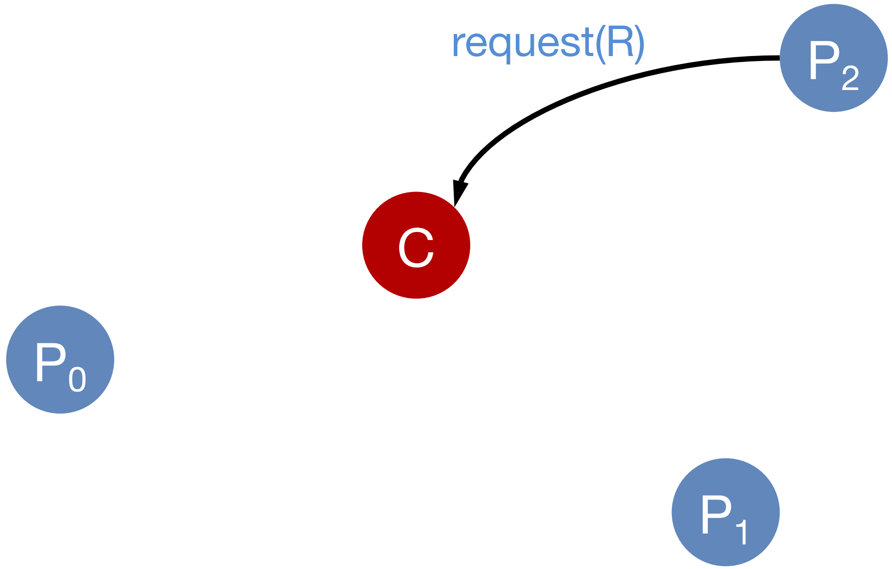
- Coordinator does not reply until release
- Maintain queue: service requests in FIFO order

R in use by: P_0

<u>R Request Queue</u>

P_1

P_2



Centralized algorithm

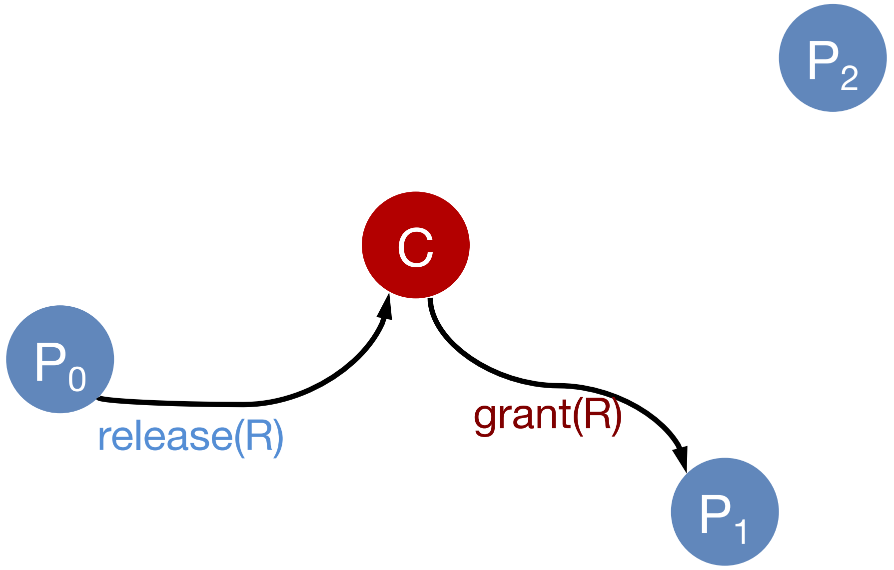
If another process claimed resource:

- Coordinator does not reply until release
- Maintain queue: service requests in FIFO order

R in use by: P_1

<u>R Request Queue</u>

P_2



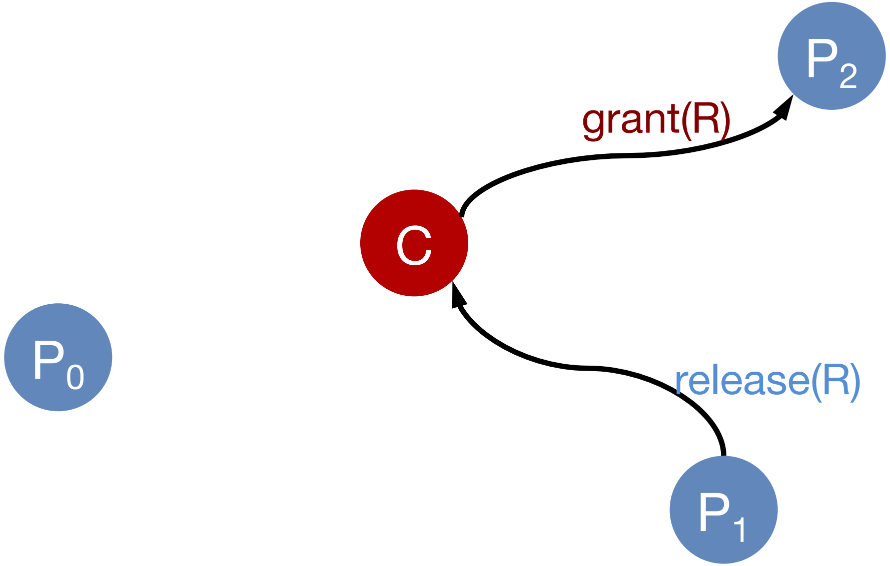
Centralized algorithm

If another process claimed resource:

- Coordinator does not reply until release
- Maintain queue: service requests in FIFO order

R in use by: P_2

R Request Queue



Centralized algorithm

Benefits

- Fair: All requests processed in order
- Easy to implement, understand, verify
- Processes do not need to know group members – just the coordinator
- Efficiency: 2 messages to enter, 1 message to exit

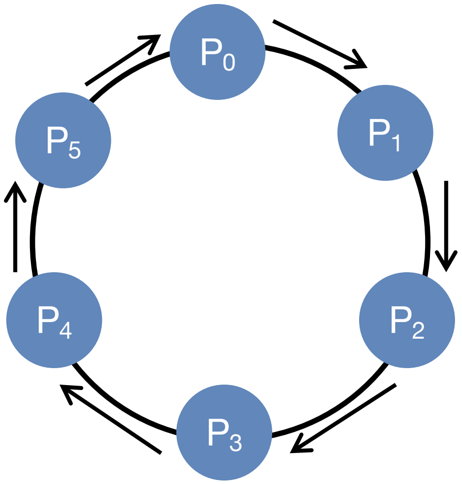
Problems

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

Token Ring algorithm

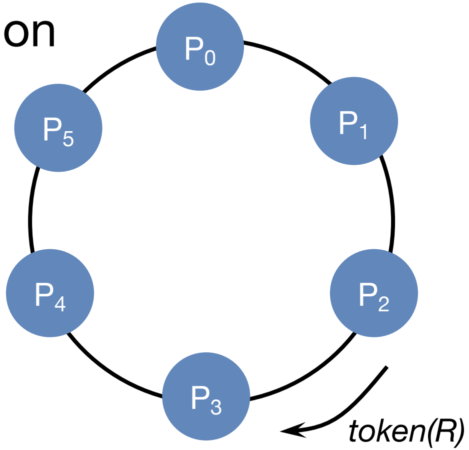
Assume known group of processes

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

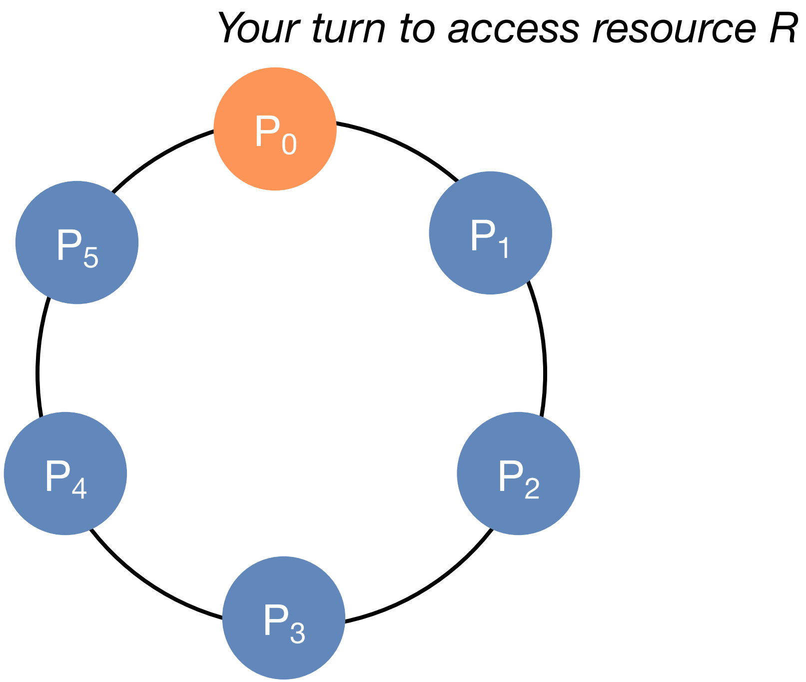


Token Ring algorithm

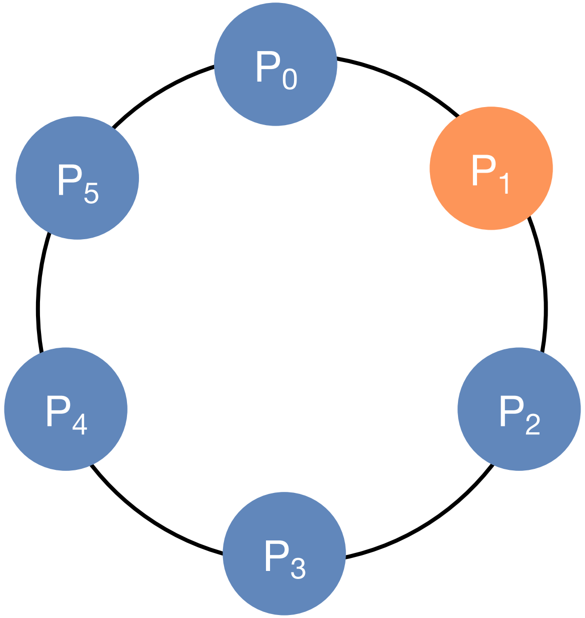
- Initialization
  - Process 0 creates a token for resource R
- Token circulates around ring from P_i to $P_{(i+1)}$mod N
- When process acquires token
  - Checks to see if it needs to enter critical section
  - If no, send ring to neighbor
  - If yes, access resource
    - Hold token until done



Token Ring algorithm

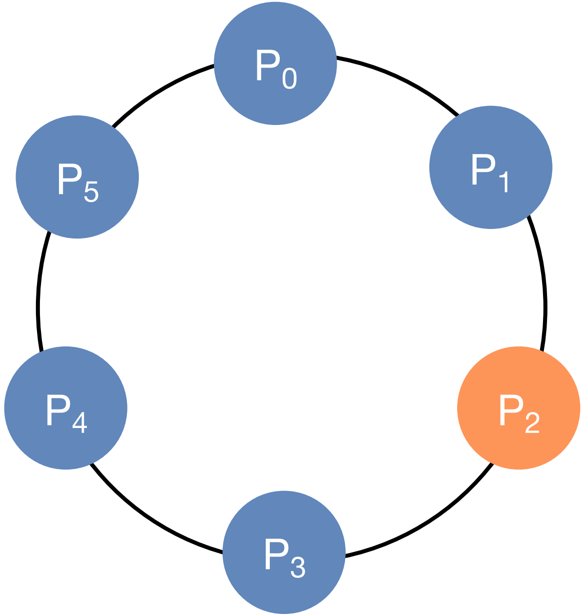


Token Ring algorithm



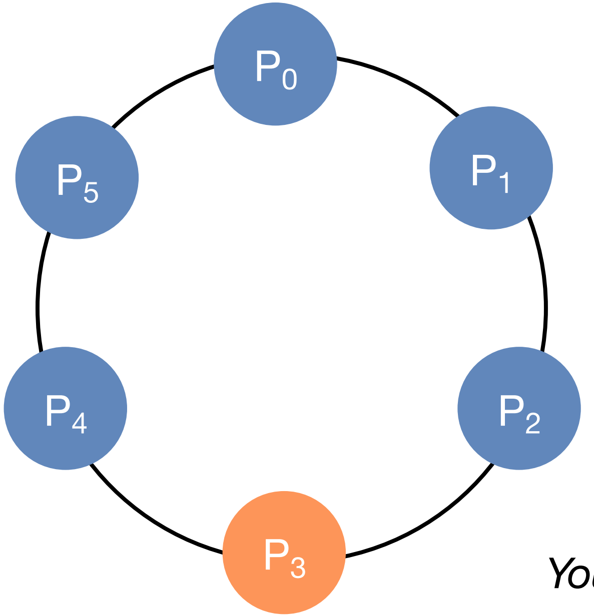
Your turn to access resource R

Token Ring algorithm



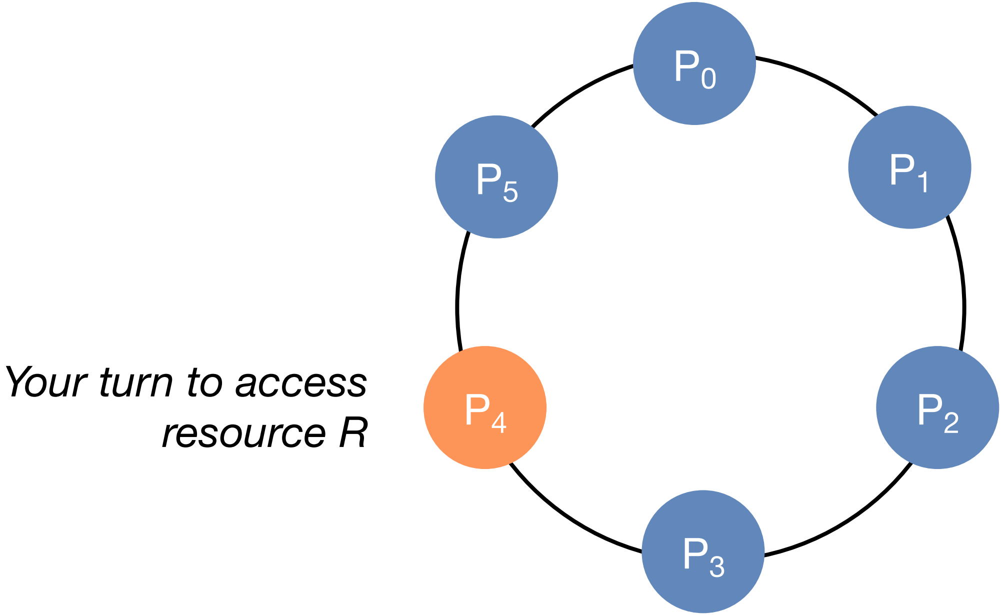
Your turn to access resource R

Token Ring algorithm

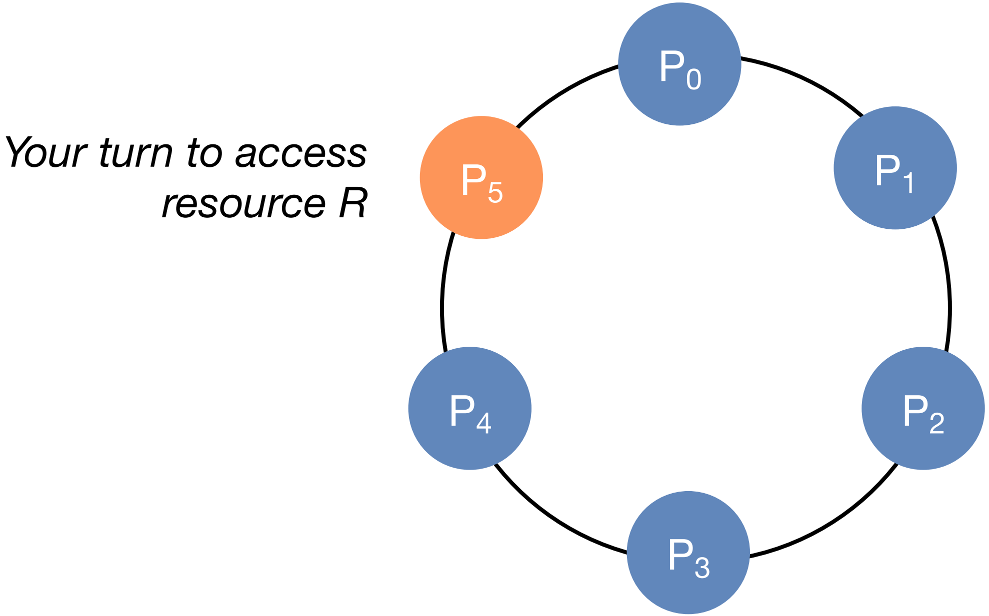


Your turn to access resource R

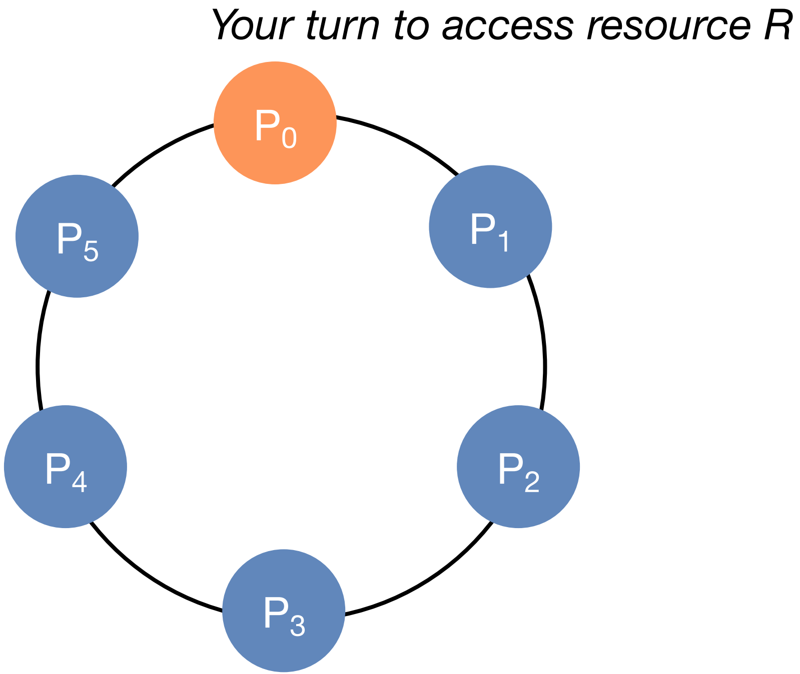
Token Ring algorithm



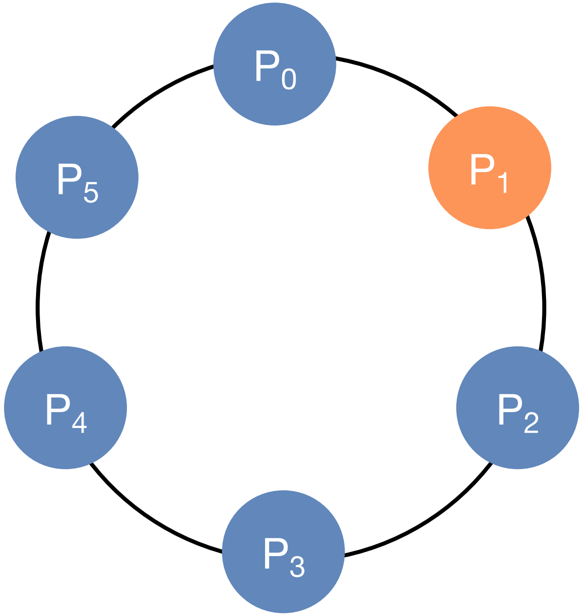
Token Ring algorithm



Token Ring algorithm



Token Ring algorithm



Your turn to access resource R

Token Ring algorithm summary

- **Safety**: Only one process at a time has token
  - Mutual exclusion guaranteed
- **Liveness**: Order well-defined (but not necessarily first-come, first-served)
  - Starvation cannot occur
  - Lack of FCFS ordering may be undesirable sometimes
- **Delay**:
  - Request = *0…N-1* messages
  - Release = 1 message

Token Ring algorithm summary

- Downsides/Problems
  - Constant activity
  - 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?

Lamport's Mutual Exclusion

Distributed algorithm using reliable multicast and logical clocks

- Messages are sent reliably and in single-source 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
- Each process maintains request queue
  - Queue contains mutual exclusion requests
  - Queues are sorted by message timestamps

1. Request a Resource

Request a critical section:

Lamport time

- Process P_i sends ***Request(R, i, T_i)*** to all nodes
  It also places the same request onto its own queue
- When a process P_j receives a request:
  - It returns a timestamped ***Reply(T_j)***
  - Places the request on its request queue

Every process will have an identical queue

- *Same contents in the same order*

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

Sample request queue for R
Identical at each process

2. Use the Resource

Enter a critical section (accessing resource):

- P_i has received ***Reply*** messages from every process Pj where $T_j > T_i$
- P_i's request has the earliest timestamp in its queue

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

Sample request queue for R
Identical at each process

If your request is at the head of the queue
AND you received Replies for that request
… then you can access the critical section

3. Release the resource

<u>Release a critical section</u>:

- 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

Assessment: Lamport's Mutual Exclusion

- **Safety**: Replicated queues – same process on top
- **Liveness**: Sorted queue & Lamport timestamps ensure earlier processes go first
- **Delay/Bandwidth**:
  - *Request* = *2(N-1)* messages: *(N-1) Request* msgs + *(N-1) Reply* msgs
  - *Release* = *(N-1) Release* msgs
- **Problems**
  - *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

Another contention-based distributed algorithm
using reliable multicast and logical clocks

When a process wants to enter critical section:

1. Compose a ***Request(R, i, T_i)*** message containing:
   - *R*: Name of resource
   - *i*: Process Identifier (machine ID, process ID)
   - T_i: Timestamp (totally-ordered Lamport)
2. Reliably multicast request to all processes in group
3. Wait until everyone gives permission (sends a ***Reply***)
4. Enter critical section / use resource

Ricart & Agrawala algorithm

When process receives a *request*:

- If receiver not interested: send ***Reply*** 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 timestamp wins**
  - If receiver is the **loser**: send ***Reply***
  - If receiver is the **winner**: do not reply – queue the request

- When **done** with critical section
  - Send ***Reply*** to all queued requests

Assessment: Ricart & Agrawala Mutual Exclusion

- **Safety**: Two competing processes will not send a REPLY to each other
  - Timestamps in the requests are unique – one will be earlier than the other
- **Liveness**: Ordered by Lamport timestamp if there is contention
- **Delay/Bandwidth**:
  - *Request* = *2(N-1)* messages: *(N-1) Request* msgs + *(N-1) Reply* msgs
  - *Release* = 0 … *(N-1) Reply* msgs to queued requests
- **Problems**
  - *N* points of failure
  - A lot of messaging traffic: requests & releases are sent to the entire group

Lamport vs. Ricart & Agrawala

Lamport

- Everyone replies … always – no hold-back
- *3(N-1)* messages
  - Request – Reply – 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 a Reply until you are done with the critical section
- *2(N-1)* messages
  - Request – ACK
- Process decides to go if it gets ACKs from everyone

Other distributed mutex algorithms

- Suzuki-Kasami
  - Adds a token to Ricart & Agrawala
  - Improves performance to (*N-1*) requests and 1 reply
- Maekawa
  - Partitions the group – each subgroup has at least one process in common with another subgroup
  - Performance improved to $3\sqrt{N}$ … $6\sqrt{N}$ messsages
- Many more…

The End