Distributed Algorithms

Mutual exclusion

Etienne Lozes (and Ludovic Henrio)
projet SCALE, I3S
etienne.lozes@univ-cotedazur.fr

Distributed Mutual Exclusion

1 – **Introduction**

2 – Solutions Using Message Passing

3 – Token Passing Algorithms

4 – A Taste of Quorum-Based Algorithms

Why Do We Need Distributed Mutual Exclusion (DME) ?

Atomicity exists only up to a certain level

Atomic instructions define the granularity of the computation
Types of possible interleaving
- Assembly Language Instruction?
- Remote Procedure Call?
- Weak memory model?

Some applications are:

- **Resource sharing**
- Avoiding concurrent update on **shared data**
- Controlling the grain of atomicity
- Medium Access Control in Ethernet

Why Do We Need Distributed Mutual Exclusion (DME) ?

Example: Bank Account Operations

shared n : integer

Process P

Account receives amount nP

Computation: n = n +nP:

```
  P1. Load Reg_P, n
  P2. Add Reg_P, nP
  P3. Store Reg_P, n
```

Process Q

Account receives amount nQ

Computation: n = n +nQ:

```
  Q1. Load Reg_Q, n
  Q2. Add Reg_Q, nQ
  Q3. Store Reg_Q, n
```

Why Do We Need DME? (example cont'd)

Possible Interleaves of Executions of P and Q:

- 2 give the expected result n= n + nP + nQ
  - P1, P2, P3, Q1, Q2, Q3
  - Q1, Q2, Q3, P1, P2, P3
- 5 give erroneous result n = n+nQ
  - P1, Q1, P2, Q2, P3, Q3
  - P1, P2, Q1, Q2, P3, Q3
  - P1, Q1, Q2, P2, P3, Q3
  - Q1, P1, Q2, P2, P3, Q3
  - Q1, Q2, P1, P2, P3, Q3
- 5 give erroneous result n = n + nP
  - Q1, P1, Q2, P2, Q3, P3
  - Q1, Q2, P1, P2, Q3, P3
  - Q1, P1, P2, Q2, Q3, P3
  - P1, Q1, P2, Q2, Q3, P3
  - P1, P2, Q1, Q2, Q3, P3

Exercise

```
int c = 0; // shared counter

void f(){
  for(int i=0;i<100;i++) c = c + 1;
}


void main() {
   f() || f()
}
```

What are all the possible values for c at the end of the program?

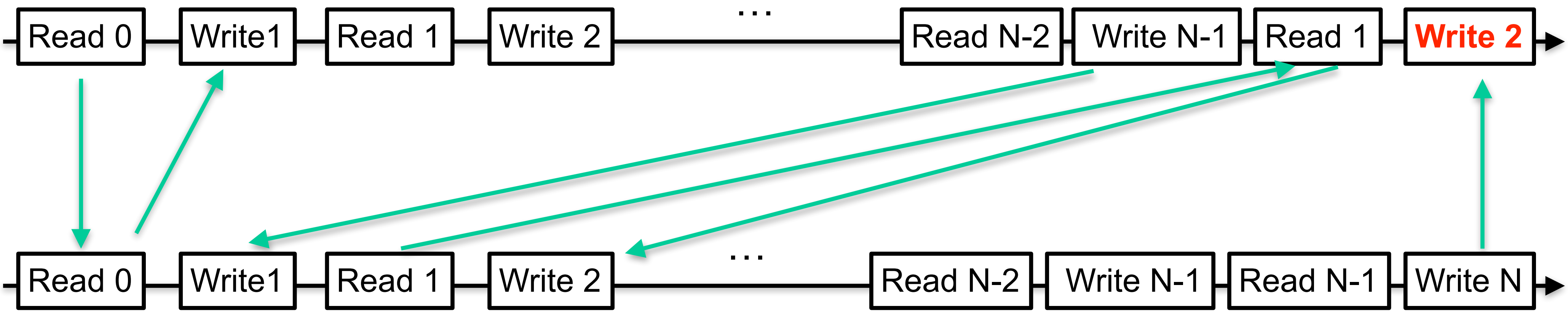
Correction

What are all the possible values for c at the end of the program?

c between 2 and 2N included.

c = 2N is when every read is immediately followed by its write.

c=2 is achieved as follows



Principle of the Mutual Exclusion Problem

Each process, before entering the CS acquires the authorization to do so.

Acquire authorisation

Enter CS

<critical section>

Exit CS

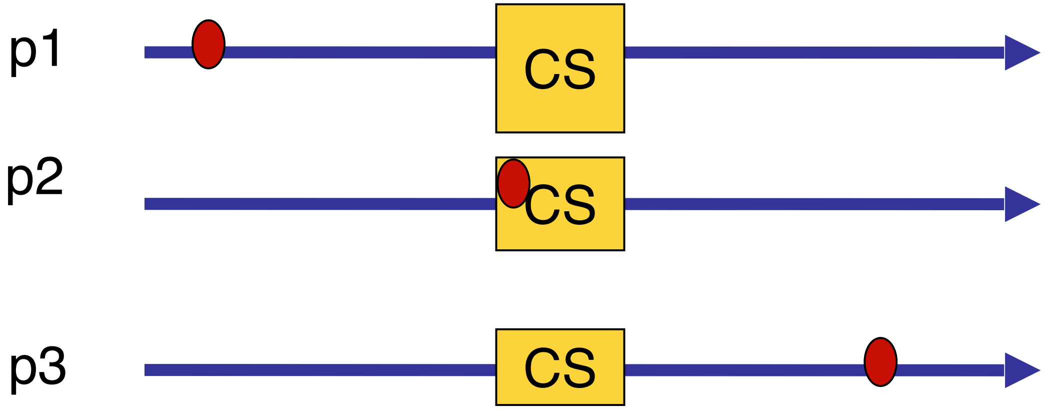
Acquire authorisation

Enter CS

<critical section>

Exit CS

Critical section should eventually terminate



Correctness Conditions

- ME1 : Mutual Exclusion
  - At most one process can remain in CS at any time
  - Safety property
- ME2 : Freedom from deadlock
  - At least one process is eligible to enter CS
  - Liveness property
- ME3 : Fairness
  - Every process trying to enter must eventually succeed
  - Absence of starvation
- A measure of fairness: bounded waiting
  - Specifies an upper bound on the number of times a process waits for its turn to enter SC -> n-fairness (n is the MAXIMUM number of rounds)
  - FIFO fairness when n=0

Mutual exclusion in the shared memory model – a solution for 2 processes

```
int last_interested; // shared variables
bool interested[2];

void ENTER_CS(tid_self){  // tid_self = 0 or 1
  interested[tid_self] = true;
  last_interested = tid_self; // write event WL
  int tid_other = 1 - tid_self;
  while( (last_interested==tid_self) && interested[tid_other] ) ;
      // spin-lock
}


void EXIT_CS(tid_self){
  interested[tid_self] = false;
}
```

Does it satisfy liveness?

Peterson's algorithm (1981)

Principle :

1) I say I am interested in entering CS
2) I say I am the last interested one
3) I wait as long as I read in shared memory that the other is also interested <u>and</u> I am still the last interested one

Peterson satisfies ME2 (liveness)

```
int last_interested; // shared variables
bool interested[2];

void ENTER_CS(tid_self){  // tid_self = 0 or 1
  interested[tid_self] = true;
  last_interested = tid_self;
  int tid_other = 1 - tid_self;
  while( (last_interested==tid_self) && interested[tid_other] ) ;
      // spin-lock
}


void EXIT_CS(tid_self){
  interested[tid_self] = false;
}
```

Does
it satisfy liveness?

YES!

Informal proof :

by absurd: if both cannot enter CS,they are in a state where both see **`last_interested==tid_self.`** Contradiction.

Safety?

```
int last_interested; // shared variables
bool interested[2];

void ENTER_CS(tid_self){  // tid_self = 0 or 1
  interested[tid_self] = true;
  last_interested = tid_self;
  int tid_other = 1 - tid_self;
  while( (last_interested==tid_self) && interested[tid_other] ) ;
      // spin-lock
}


void EXIT_CS(tid_self){
  interested[tid_self] = false;
}
```

Does it satisfy **safety**?

What's wrong here: they did not necessarily negate the condition in the same state. It's all about interleaving and causal dependencies.

Proof attempt:

« there is no state in which both see the negation of last_interested==tid_self && interested[tid_other], i.e. last_interested!=tid_self || ! interested[tid_other]. »

Peterson satisfies ME1 (safety)

Proof :

by absurd: assume both entered CS. So each passed through a state where `last_interested!=tid_self || ! interested[tid_other]`.

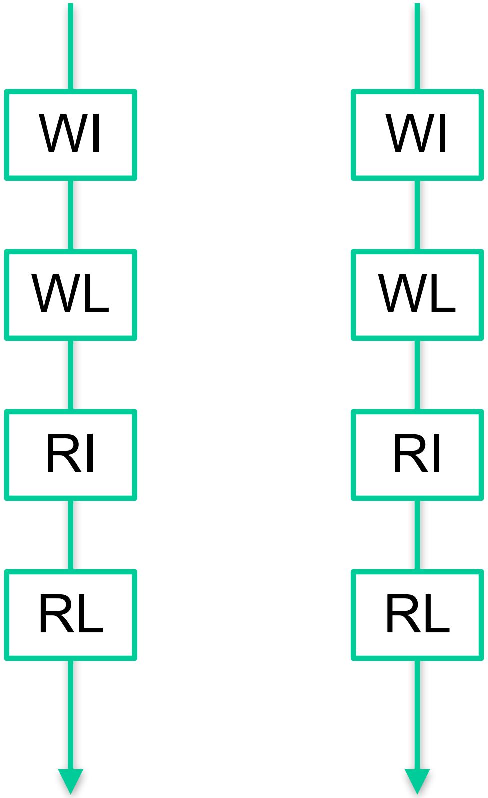
There are 4 events per thread

1. write interested (WI)
2. write last (WL)
3. read interested (RI)
4. read last (RL)

that happen exactly in this order

(well, except 3 and 4 that are not strictly ordered)

Let's first assume that thread 0 passed through a state where `last_interested!=tid_self`



Peterson satisfies ME1 (cont.)

Proof :

by absurd: assume both entered CS. So each passed through a state where `last_interested!=tid_self || ! interested[tid_other]`.

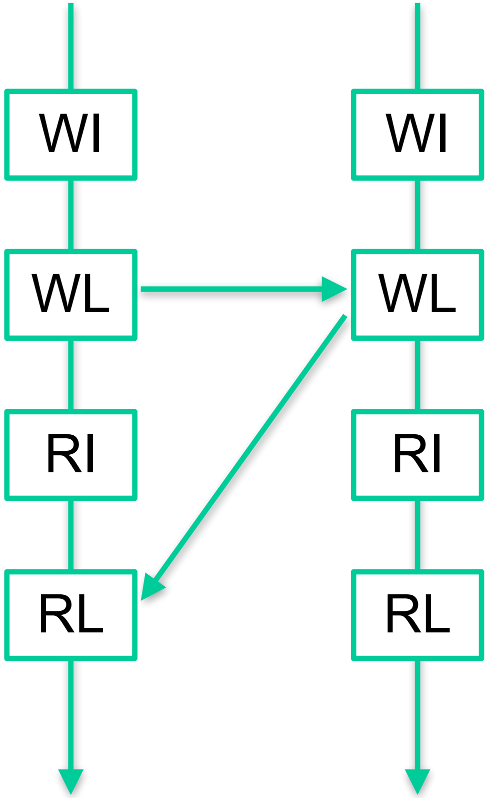
There are 4 events per thread

1. write interested (WI)
2. write last (WL)
3. read interested (RI)
4. read last (RL)

that happen exactly in this order

(well, except 3 and 4 that are not strictly ordered)

Let's first assume that thread 0 passed through a state where `last_interested!=tid_self`



Peterson satisfies ME1 (cont.)

Proof :

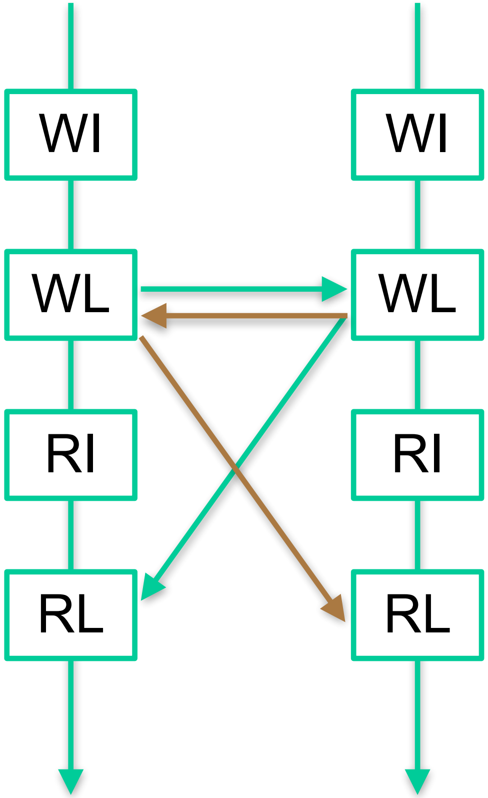
by absurd: assume both entered CS. So each passed through a state where `last_interested!=tid_self || ! interested[tid_other]`.

There are 4 events per thread

1. write interested (WI)
2. write last (WL)
3. read interested (RI)
4. read last (RL)

that happen exactly in this order

(well, except 3 and 4 that are not strictly ordered)



Let's first assume that thread 0 passed through a state where `last_interested!=tid_self`

`If thread 1 also passed through last_interested!=tid_self`

`we get a cycle of « happens before » relation : contradiction.`

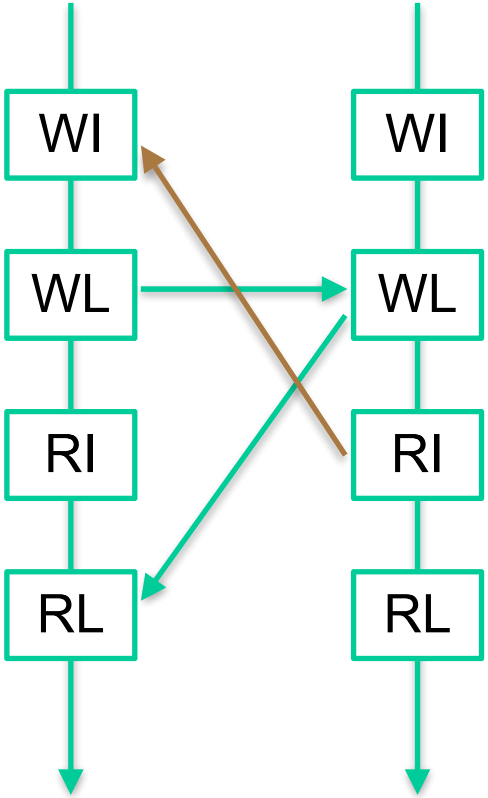
Peterson satisfies ME1 (cont.)

Proof :

by absurd: assume both entered CS. So each passed through a state where `last_interested!=tid_self || ! interested[tid_other]`.

There are 4 events per thread
1. write interested (WI)
2. write last (WL)
3. read interested (RI)
4. read last (RL)

that happen exactly in this order
(well, except 3 and 4 that are not strictly ordered)



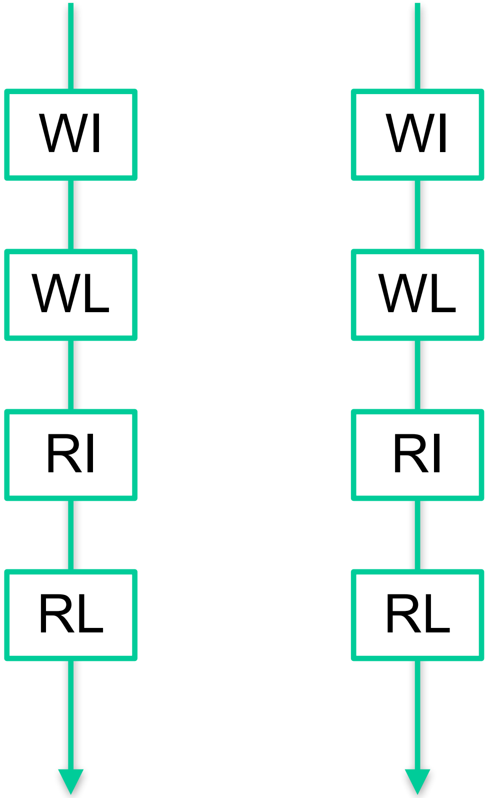
Let's first assume that thread 0 passed through a state where `last_interested!=tid_self`

If thread 1 passed through `! interested[tid_other]` we also get a cycle RI1->WI0->WL0->WL1->RI1.

Peterson satisfies ME1 (cont.)

Proof :
by absurd: assume both entered CS. So each passed through a state where
`last_interested!=tid_self || ! interested[tid_other]`.

There are 4 events per thread
1. write interested (WI)
2. write last (WL)
3. read interested (RI)
4. read last (RL)
that happen exactly in this order
(well, except 3 and 4 that are not strictly ordered)



Finally let's assume both passed through a state where
`! interested[tid_other].`

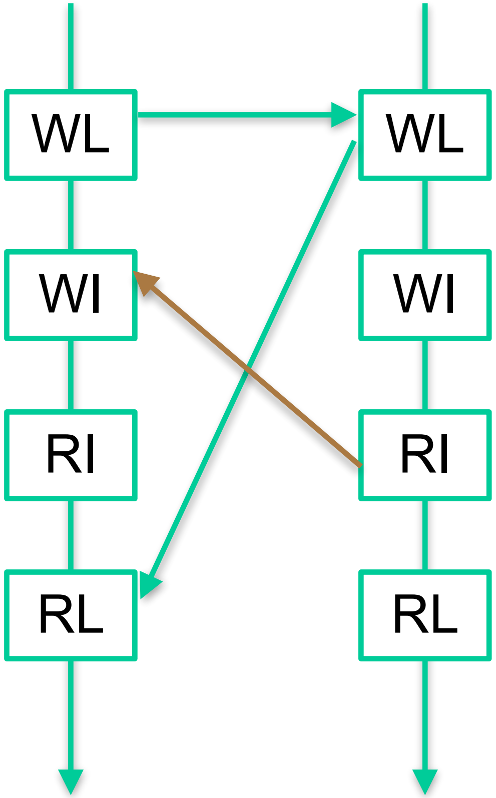
Exercise: put the arrows and end the proof!

What about changing the order of the two writes?

```
int last_interested;
bool interested[2];

void ENTER_CS(tid_self){
  last_interested = tid_self;
  interested[tid_self] = true;
 int tid_other = 1 - tid_self;
  while( (last_interested==tid_self)
          && interested[tid_other] ) ;
      // spin-lock
}
```

Does it still satisfy safety?

This corresponds to what we saw two slides ago. But now, no cycle!



Conclusion: this variant of Peterson's algorithm does not ensure safety !!!

Weak memory models

« Beware of bugs in the above code; I have only proved it correct, not tried it. »

Donald Knuth
in Notes on the van Emde Boas construction of priority deques: An instructive use of recursion

Weak memory models : any reordering of read/write instructions can occur provided they do not change the meaning of the code, **if considered single-threaded**.

Why? Because cache coherence is expensive! These reorderings aim at reducing synchronizations among cores. You can force synchronizations using barriers (fences).

Distributed Mutual Exclusion

1 – Introduction

2 – Solutions using Message Passing

3 – Token Passing Algorithms

4 – A Taste of Quorum-Based Algorithms

Problem formulation

- Assumptions
  - n processes (n>1), numbered 0 ... n-1, noted Pi communicating by sending / receiving messages
  - topology: completely connected graph
  - each Pi periodically wants:
    1. enter the Critical Section (CS)
    2. execute the CS code
    3. eventually exit the CS code
- Devise a protocol that satisfies:

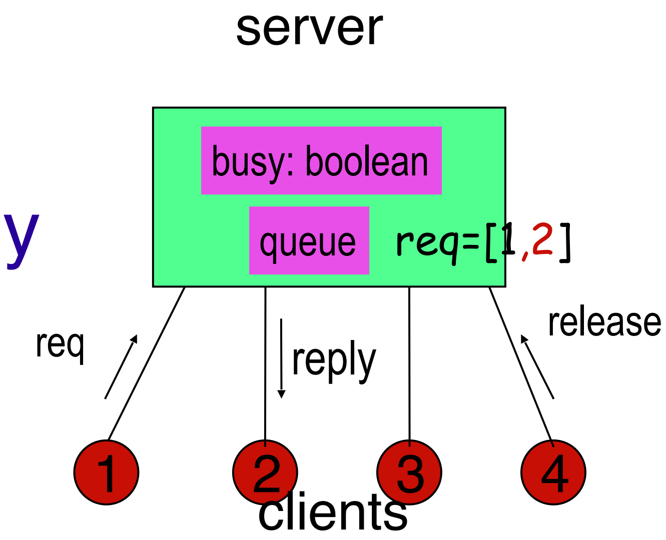
ME1 : Mutual Exclusion

ME2 : Freedom from deadlock

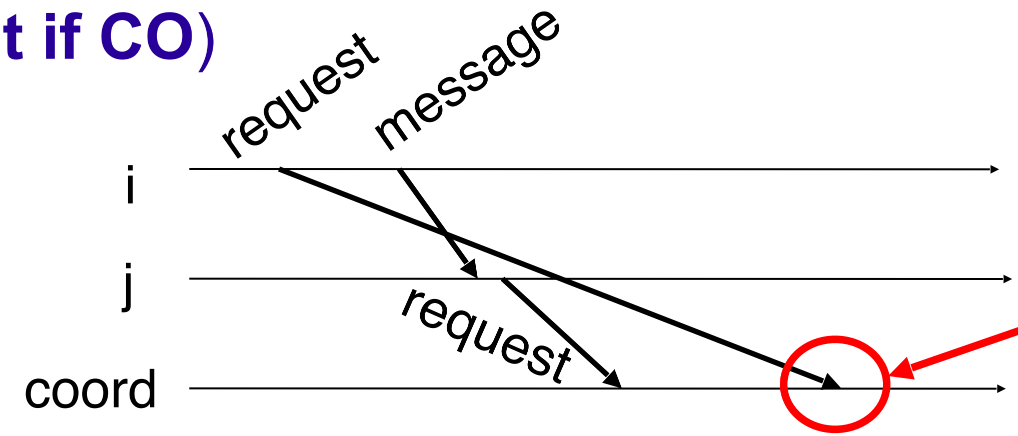
ME3 : Progress (of each process) → Fairness

Centralized solution

- Use a coordinator process
  - External process
  - One of the Pi-s
- Queue requests and authorize one by one
- Problems:
  - Major: Single point of failure, contention
  - Minor: Unable to achieve FIFO fairness (**except if CO**)



Example:



How to anticipate this late arrival?

Distributed solution : naïve approach

Before entering critical section:
1) broadcast a **REQUEST** message to all others
2) wait for **ACK** messages from all others
3) when done, enter critical section

When leaving critical section
1) broadcast a **RELEASE** message to all others

Why does not it work?

If two processes broadcast REQUEST concurrently, they confuse everybody.

What if a timestamp is given when sending REQUEST?

Lamport's Solution

- Assumptions:
  - Each communication channel is FIFO
  - **Each process maintains a queue Q of known requests**
- Algorithm described by 5 rules

LA1. To request entry, send a time-stamped message to **every** other process and **enqueue to local Q (of sender)**

LA2. Upon reception place request in Q and send time-stamped ACK but **once out of CS** (possibly immediately if already out)

LA3. Enter CS when:

1. request first in Q (chronological order)
2. AND all ACK received from others

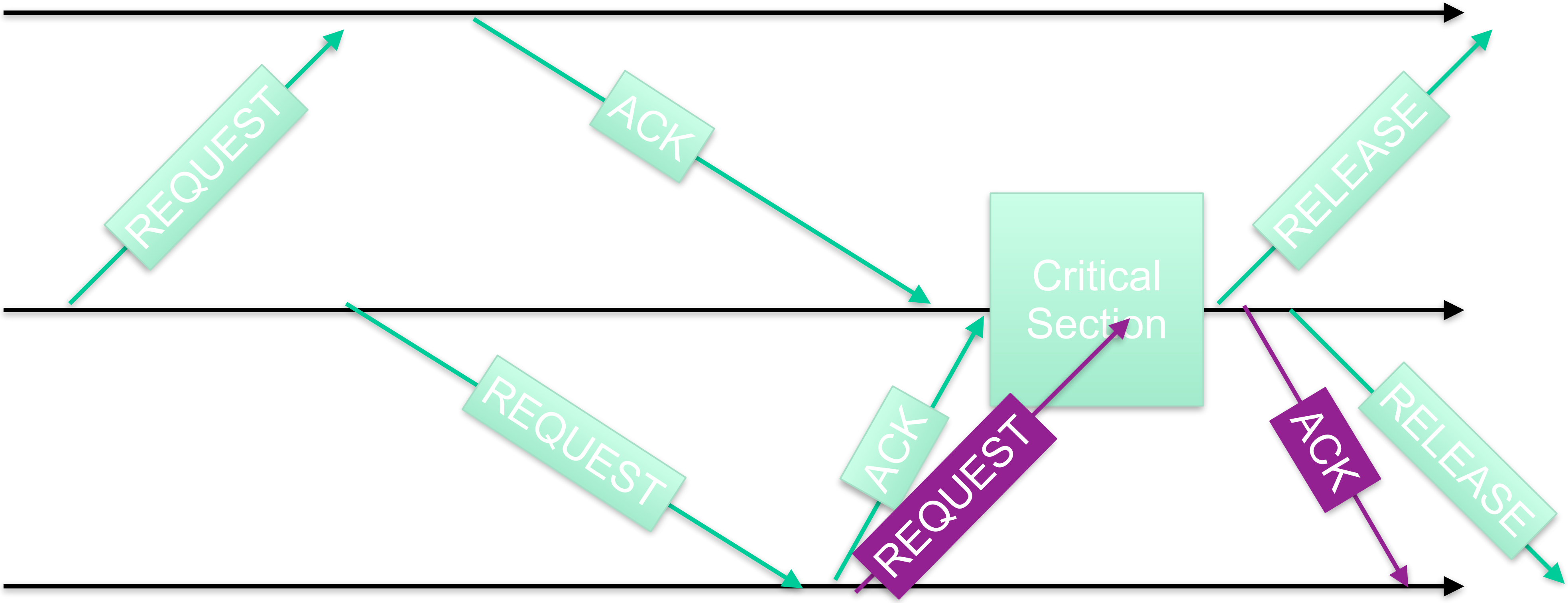
LA4. To exit CS, a process must:

1. delete request from Q
2. send time-stamped release message to others

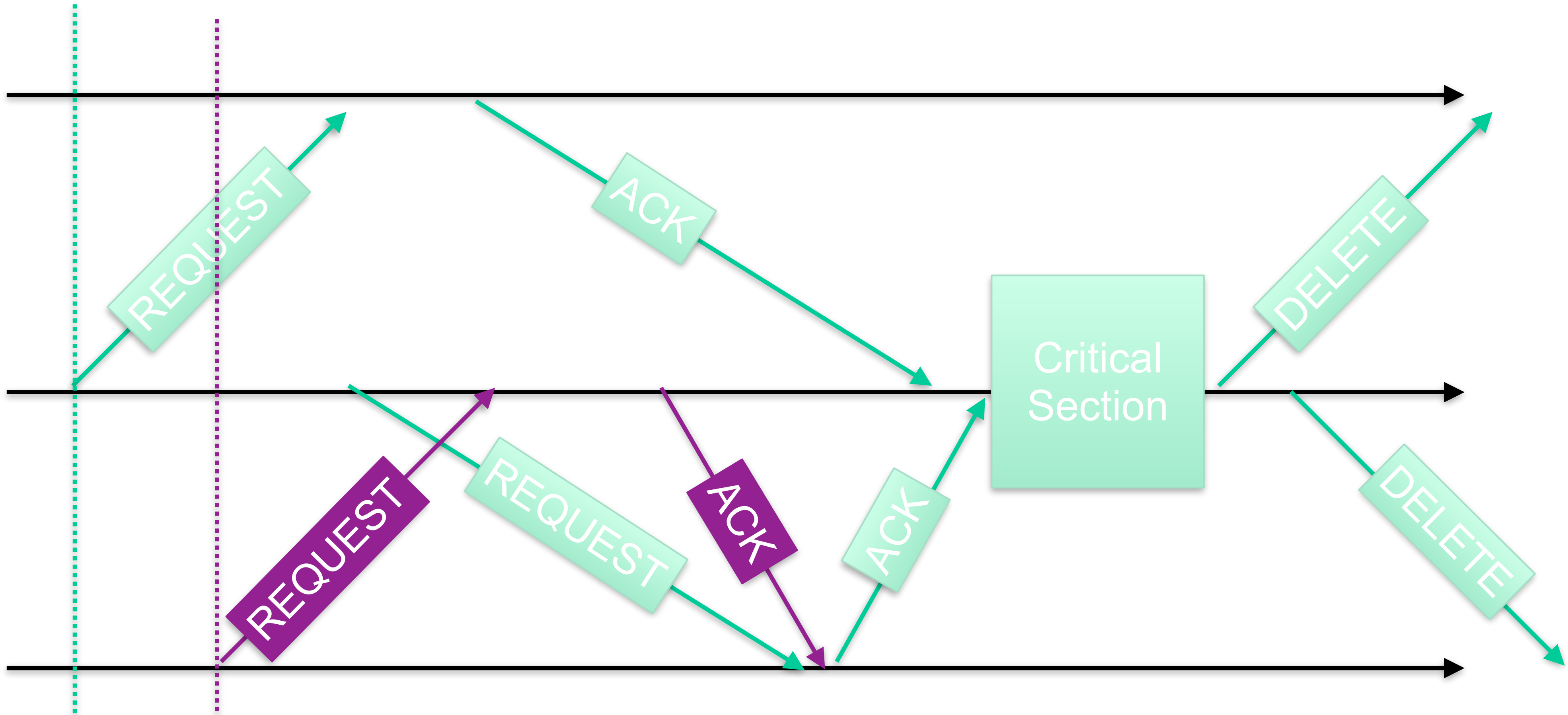
LA5. When receiving a release msg, remove request from Q

Run an example with 3 processes and different interleavings

Example 1

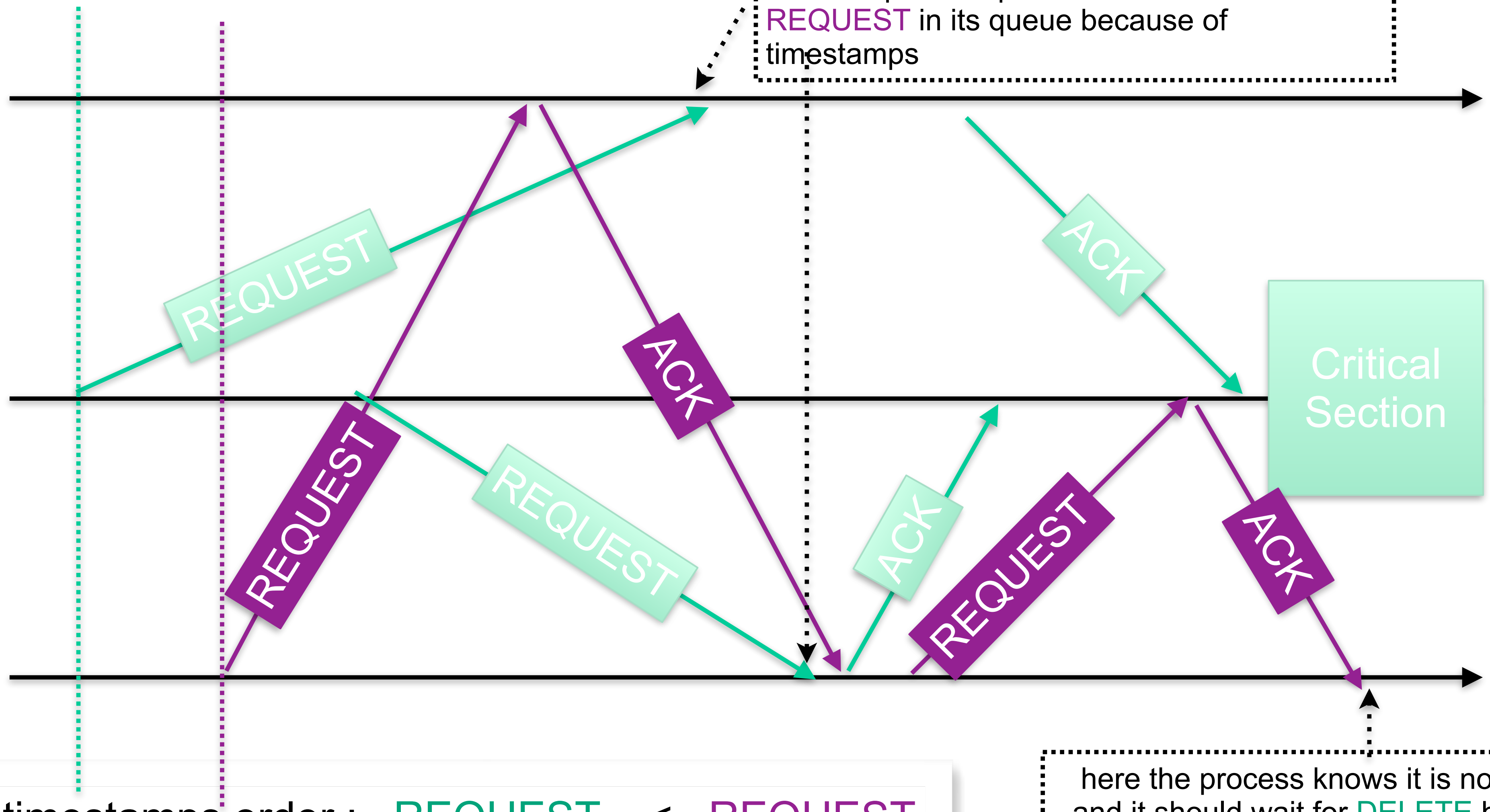

REQUEST
ACK
REQUEST
Critical Section
ACK
REQUEST
RELEASE
ACK
RELEASE

Example 2



timestamps order : REQUEST < REQUEST

Example 3



here the process places REQUEST before REQUEST in its queue because of timestamps

here the process knows it is not first and it should wait for DELETE before entering the critical section

timestamps order : REQUEST < REQUEST

Analysis of Lamport's Solution

Can you show that it satisfies all the properties (i.e. ME1, ME2, ME3) of a correct solution?

Observation. *when all ACKs have been received any request on the way has a greater ts.*

=> "coherent" view of the queue

WHY?

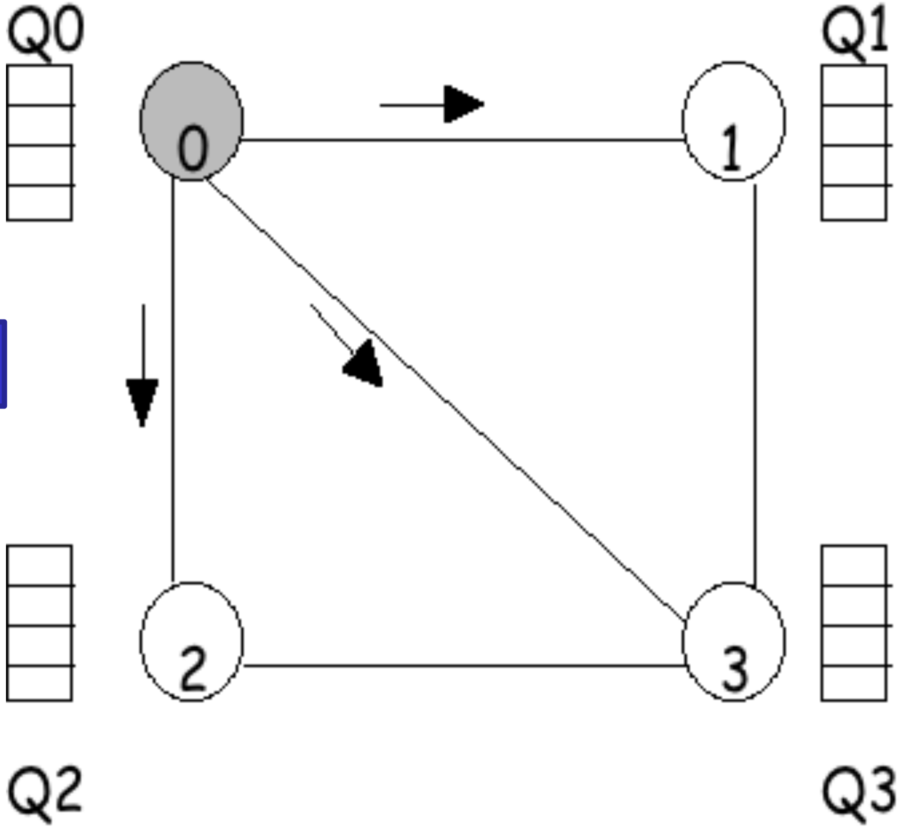
Proof of ME1. *At most one process can be in its CS at any time.*

Suppose not, and both j,k enter their CS. This implies

- j in CS ⇒ Qj.ts < Qk.ts
- k in CS ⇒ Qk.ts < Qj.ts

WHY?

Impossible.



Analysis of Lamport's Solution (2)

Proof of ME2. *(No deadlock)*

The waiting chain is acyclic.

i waits for j

⇒ i is behind j in all queues

(or j is in its CS)

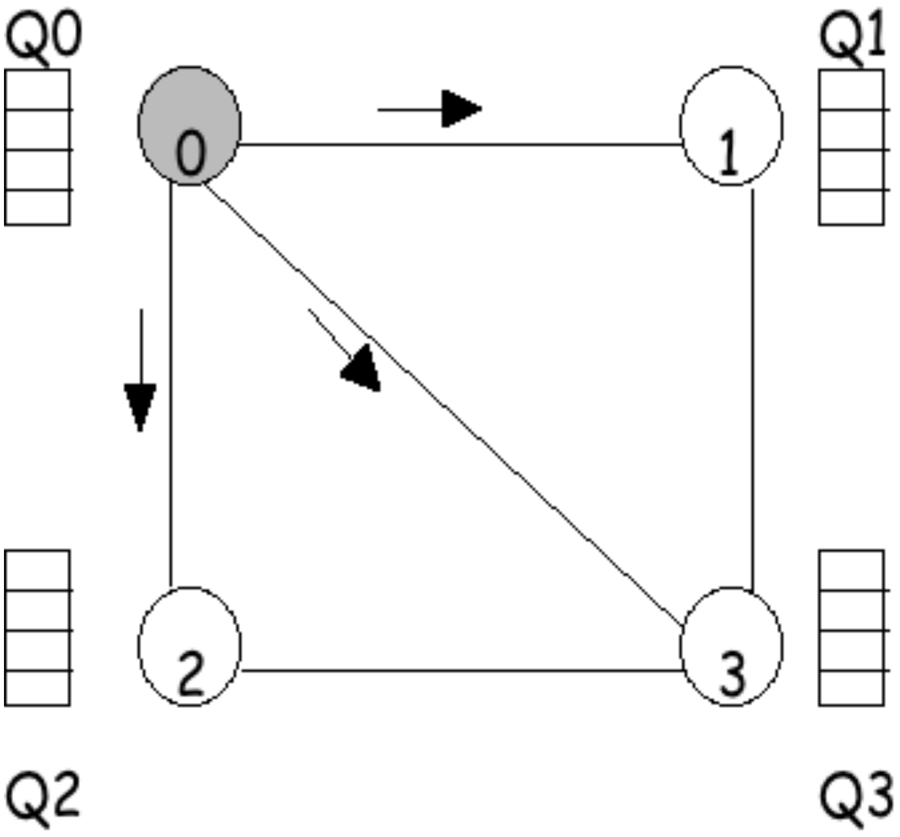
⇒ j does not wait for i

Proof of ME3. *(progress)*

New requests join the end of the queues,

WHY? ALWAYS?

so new requests do not pass the old ones



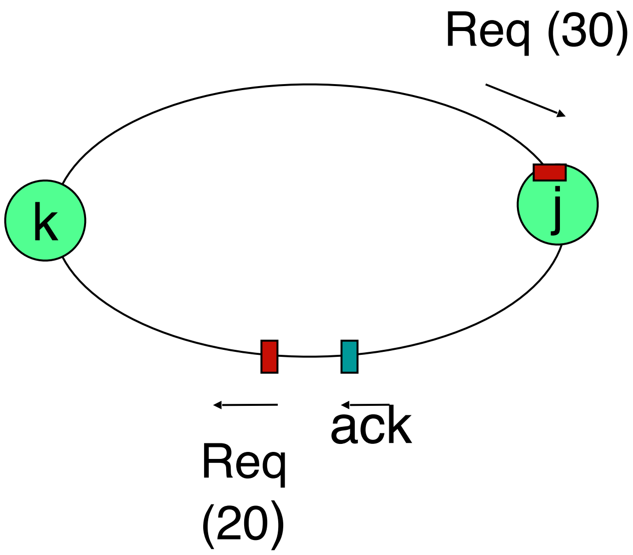
What is causal ordering?

Analysis of Lamport's Solution (3)

Proof of FIFO fairness.

timestamp (j) < timestamp (k)

⇒ j enters its CS before k does so



Suppose not. So, k enters its CS before j. So k did not receive j's request. But k received the ack from j for its own req.

This is impossible **if the channels are FIFO**

.

Message complexity = 3(N-1) (per trip to CS)

(N-1 requests + N-1 ack + N-1 release)

Ricart & Agrawala's Solution

What is new?

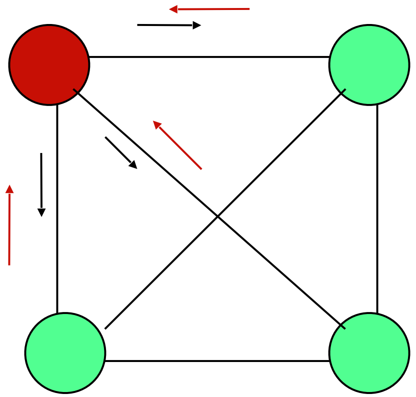
1. Broadcast a timestamped ***request*** to all.

2. Upon receiving a request, send ***ack*** if

-You do not want to enter your CS, or

-You are trying to enter your CS, but your timestamp is higher than that of the sender.

(If you are already in CS or have a smaller timestamp, then reply nothing but remember the request as *pending*)



3. **Enter CS**, when you receive ***ack*** *from all*.

4. Upon **exit from CS**, send ***ack*** to each pending request before making a new request.

(No release message is necessary)

Run an example with 3 processes and different interleavings

Your turn!

Analysis of Ricart & Agrawala's Solution

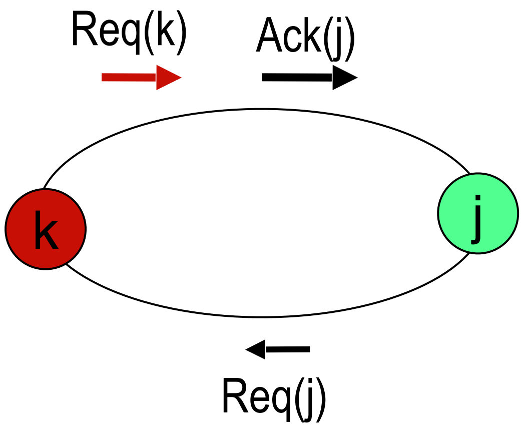
Exercise

ME1. Prove that at most one process can be in CS.

ME2. Prove that deadlock is not possible.

ME3. Prove that FIFO fairness holds **even if channels are not FIFO (note: this is the same fairness as in Lamport's solution)**

$TS(j) < TS(k)$



Message complexity = 2(N-1)
(N-1 requests + N-1 acks - no release message)

Exercise

- A Generalized version of the mutual exclusion problem in which up to L processes (L ≥1) are allowed to be in their critical sections simultaneously is known as the **L-exclusion** problem.
- Precisely, if fewer than L processes are in the CS at any time and one more process wants to enter it, it must be allowed to do so.

- **Modify R.-A. algorithm to solve the L-exclusion problem.**

Distributed Mutual Exclusion

1 – Introduction

2 – Solutions Using Message Passing

3 – Token Passing Algorithms

4 – A Taste of Quorum-Based Algorithms

Token Ring Approach

Processes are organized in a logical ring: pi has a communication channel to p(i+1) mod (n).

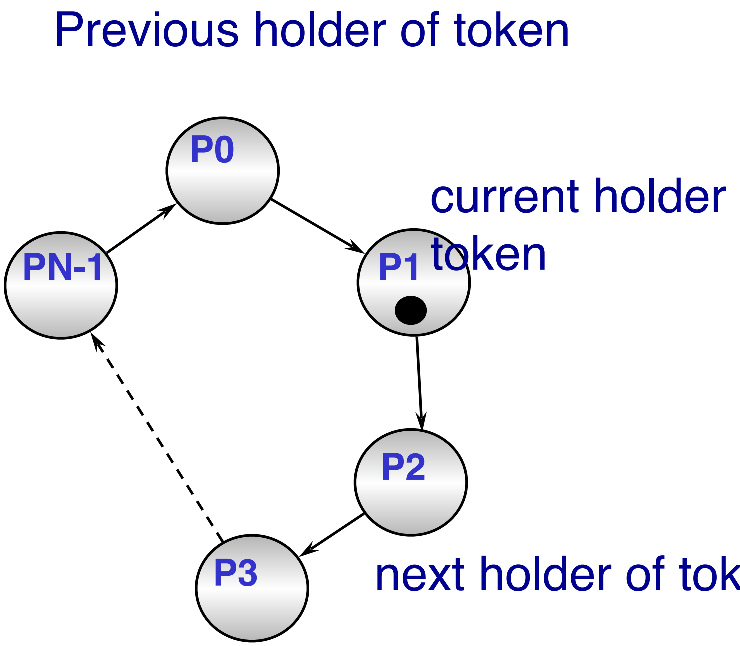
Operations:

Only the process holding the token can enter the CS.

To enter the critical section, wait passively for the token. When in CS, hold on to the token.

To exit the CS, the process sends the token onto its neighbor.

If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.



Previous holder of token

current holder token

next holder of tok

The basic ring approach

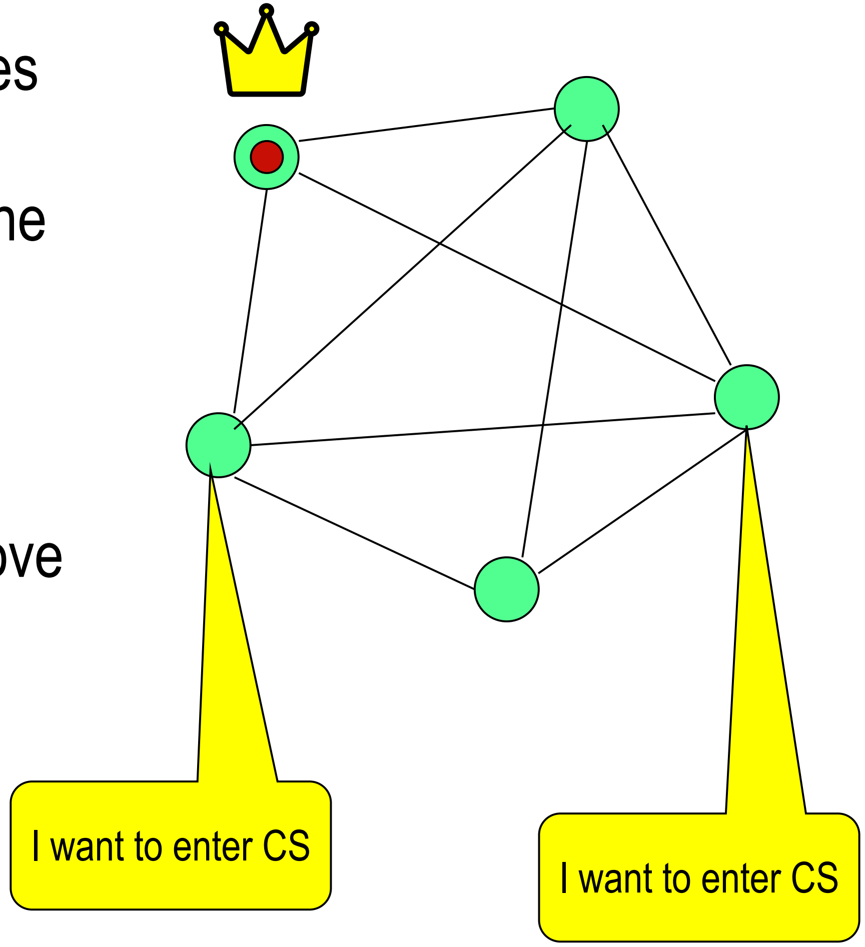
- **Features:**
  - **Safety & liveness are guaranteed, but ordering is not.**
  - **Bandwidth: 1 message per exit**
  - **(N-1) -fairness**
  - **Delay between one process's exit from the CS and the next process's entry is between 1 and N-1 message transmissions.**

Completely connected networks

Completely connected network of processes

There is **one token** (👑) in the network. The holder of the token has the permission to enter CS.

Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.



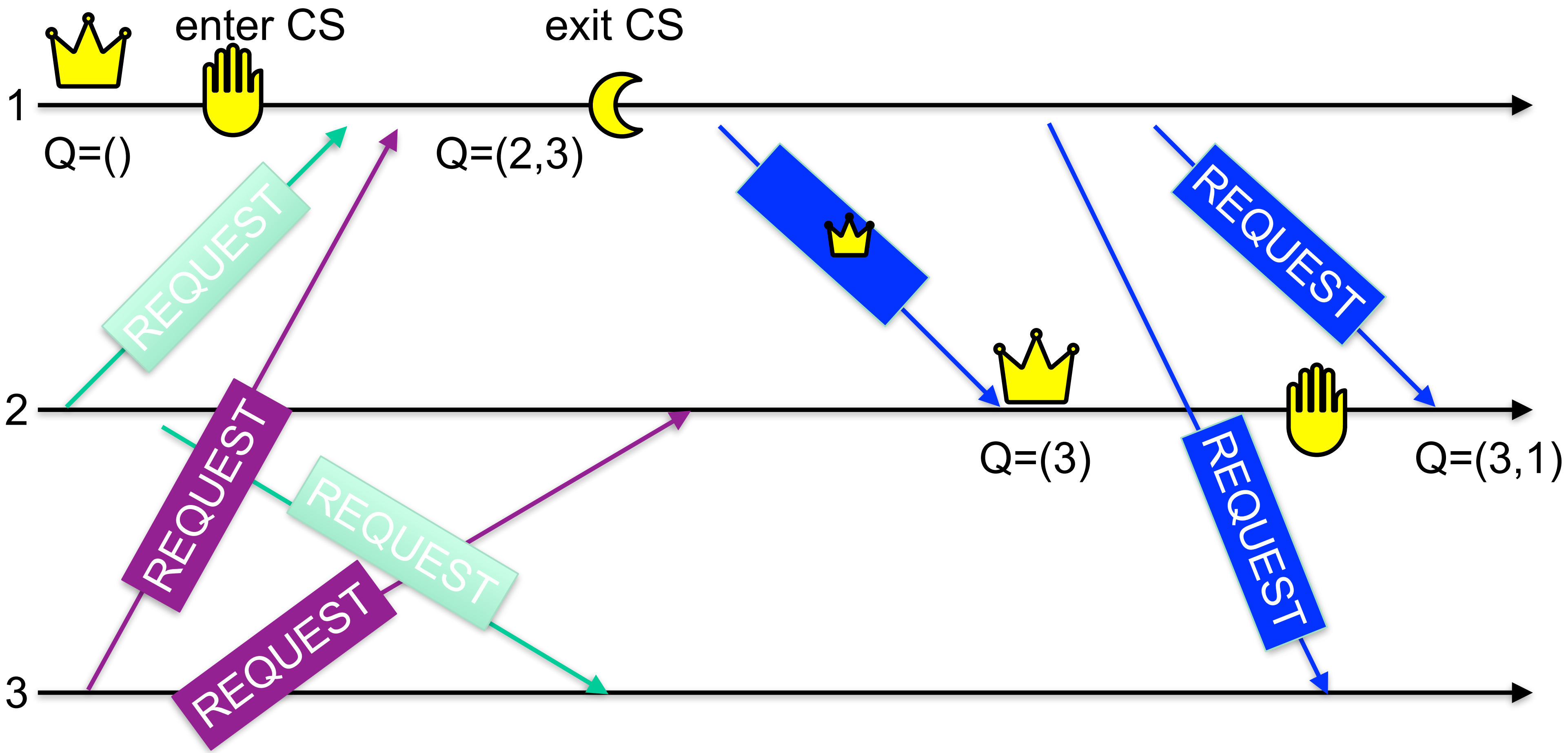
A naïve algorithm

The king maintains **a queue of pending requests.**
If Pi receives a request from Pj while it holds the token, it adds Pj to the queue.

When the king exits CS, it sends the token to the first process in the queue, together with the queue:
the token is the queue.

Since the king changes, it is not known in advance, so any process **broadcasts its request** for entering CS.

Example



The queue moves from 1 to 2

Exercise

Give two scenarios where the algorithm goes wrong as follows:

- a process request remains unsatisfied (starvation)
- a process receives the token even if he did not ask it (!)

Your turn!

Suzuki-Kasami Algorithm

Process i broadcasts **(i, num)**

Sequence number of the request

Each process maintains

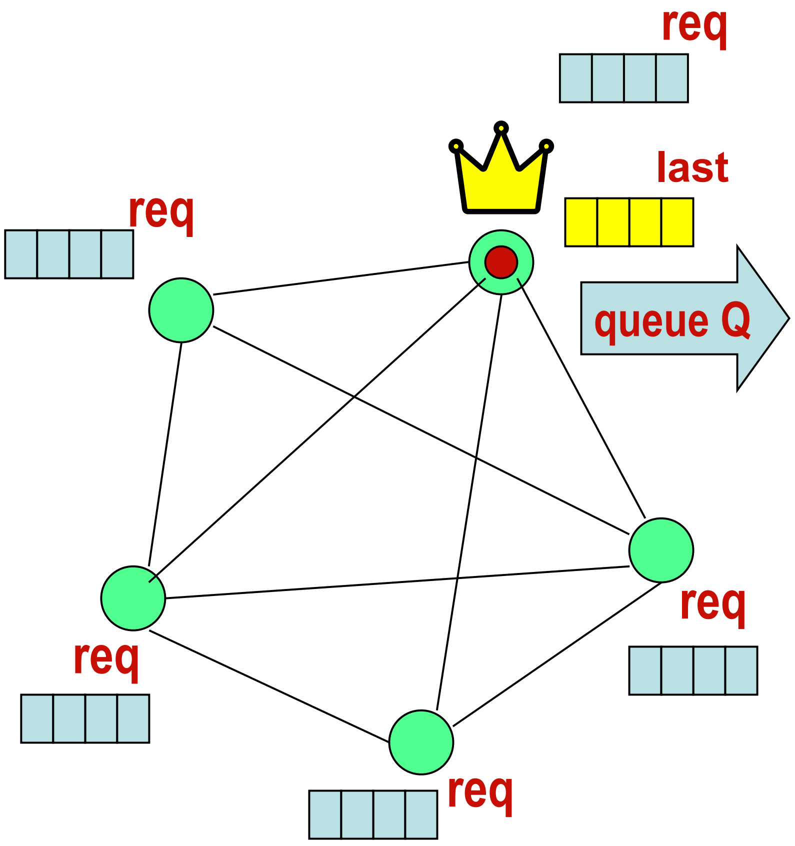
-an array **req**: **req[j]** denotes the sequence nb of the ***latest request*** from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

-an array **last: last[j]** denotes the sequence number of *the latest visit* to CS for process j.

- **a queue Q** of waiting processes



req: array[0..n-1] of integer

last: array [0..n-1] of integer

Suzuki-Kasami Algorithm (2)

When a process **i** receives a request **(k, num)** from process **k,** it sets **req[k] to max(req[k], num).**

Qu: why???

The holder of the token

--Completes its CS

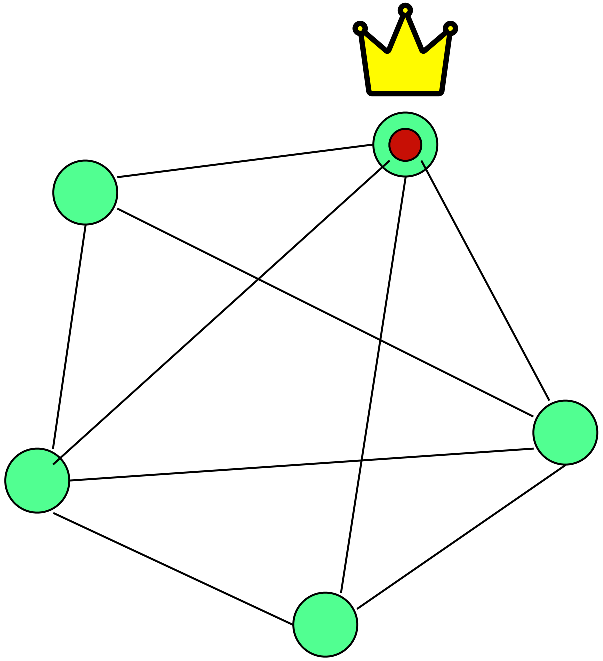
--Sets **last[i]:=** its own **num**

--Updates **Q** by adding all processes **k** such that **1+ last[k] = req[k] and k not in Q**

(*This guarantees the freshness of the request)*

--Sends the token to the ***head of Q***, along with the array **last** and the ***tail of Q***

In fact, **token ≡ (Q, last)**



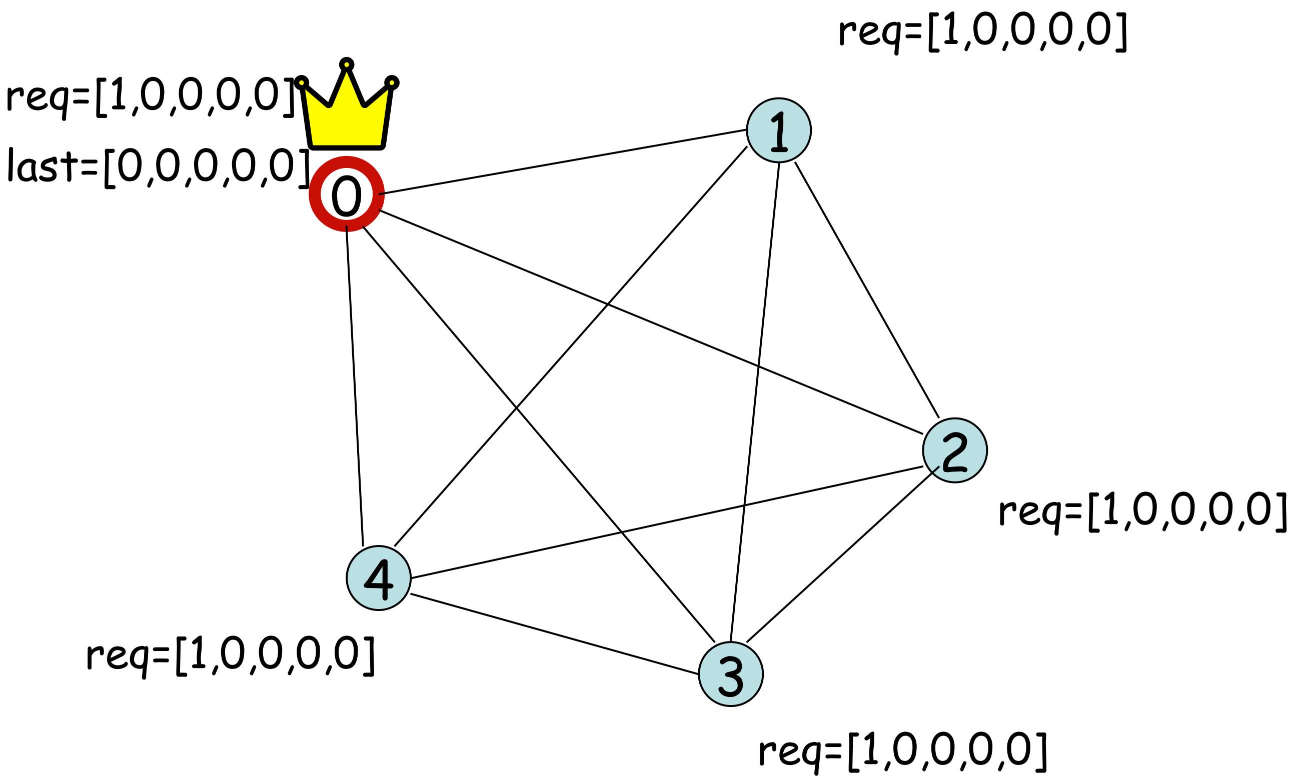
Req: array[0..n-1] of integer

Last: Array [0..n-1] of integer

Suzuki-Kasami Algorithm (3)

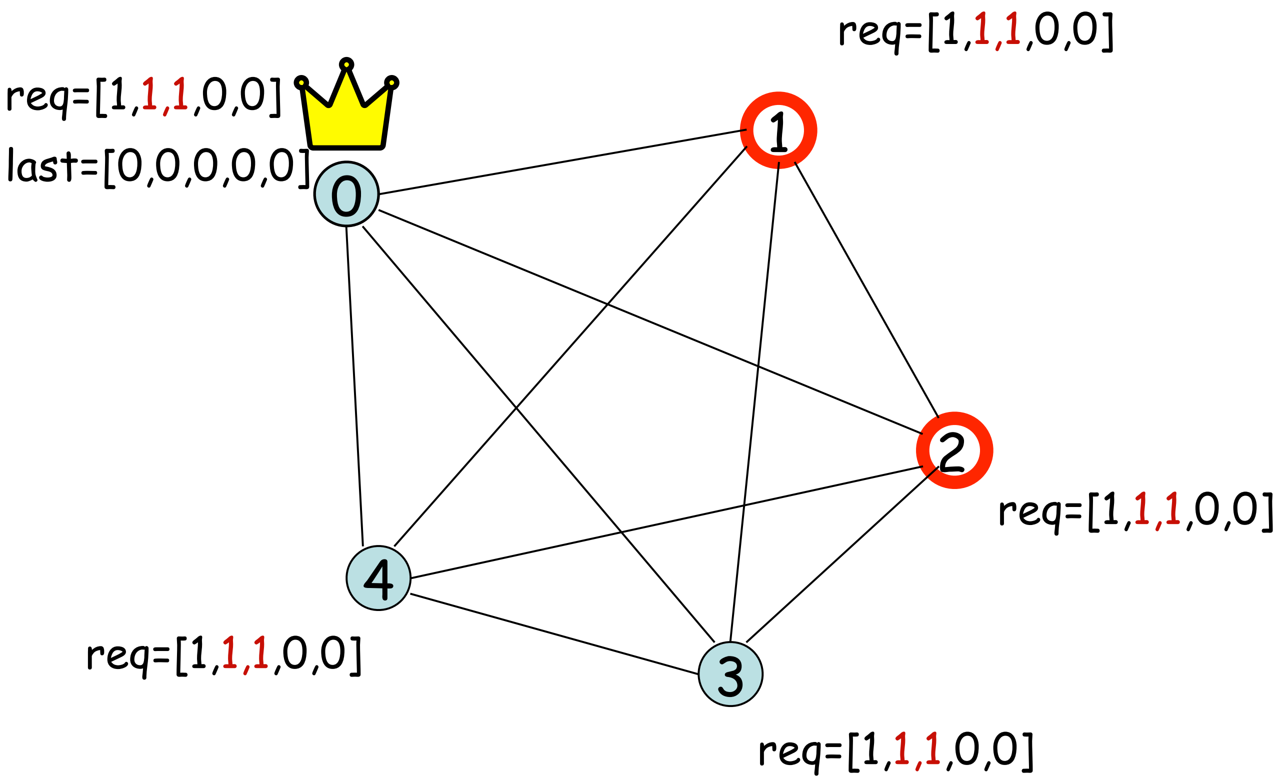
```
{Program of process j}
Initially, $\forall i$: req[i] = last[i] = 0
* Entry protocol *
        req[j] := req[j] + 1
        Send (j, req[j]) to all
        Wait until token (Q, last) arrives
        Critical Section
* Exit protocol *
        last[j] := req[j]
        $\forall k \neq j$: $k \notin Q \wedge req[k] = last[k] + 1$ → append k to Q;
        if Q is not empty → send (tail-of-Q, last) to head-of-Q fi

* Upon receiving a request (k, num) *
        req[k] := max(req[k], num)
```

Example of Suzuki-Kasami Algorithm Execution



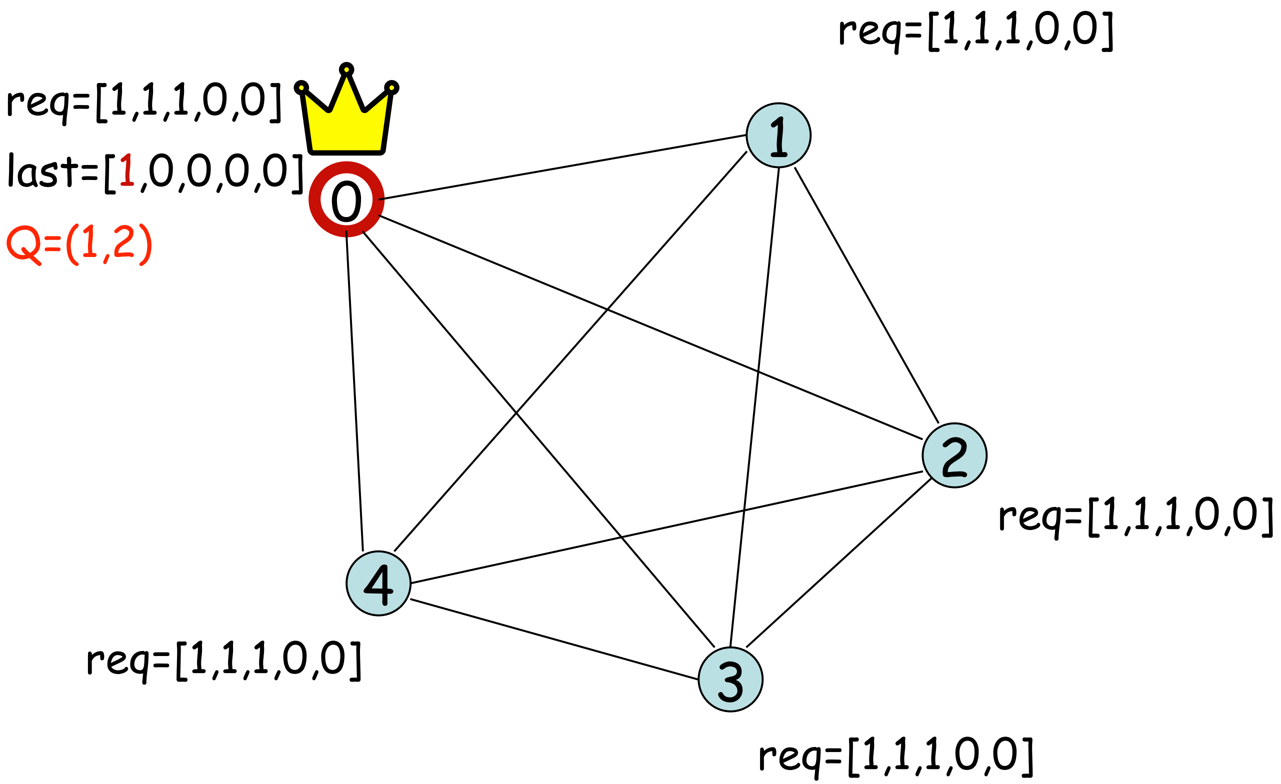
initial state: process 0 has sent a request to all, and grabbed the token

Example of Suzuki-Kasami Algorithm Execution



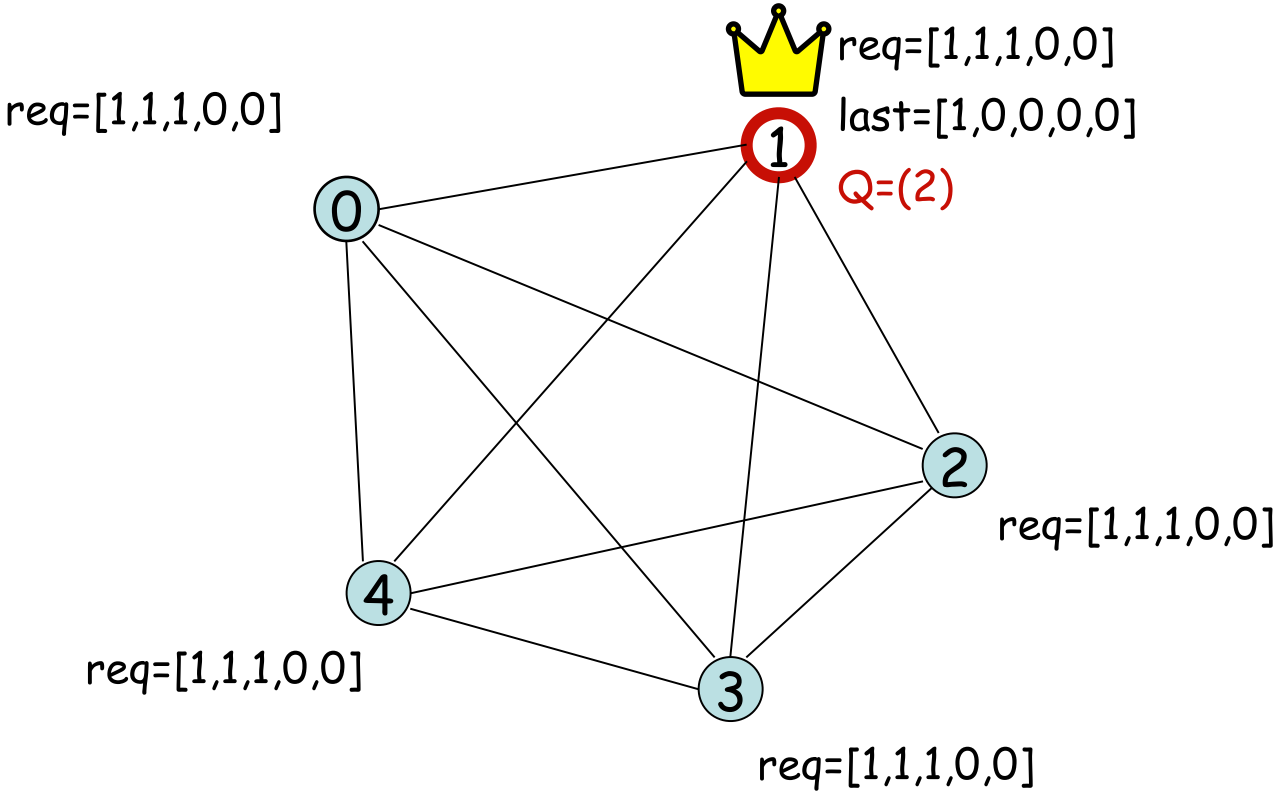
1 & 2 send requests to enter CS

Example of Suzuki-Kasami Algorithm Execution



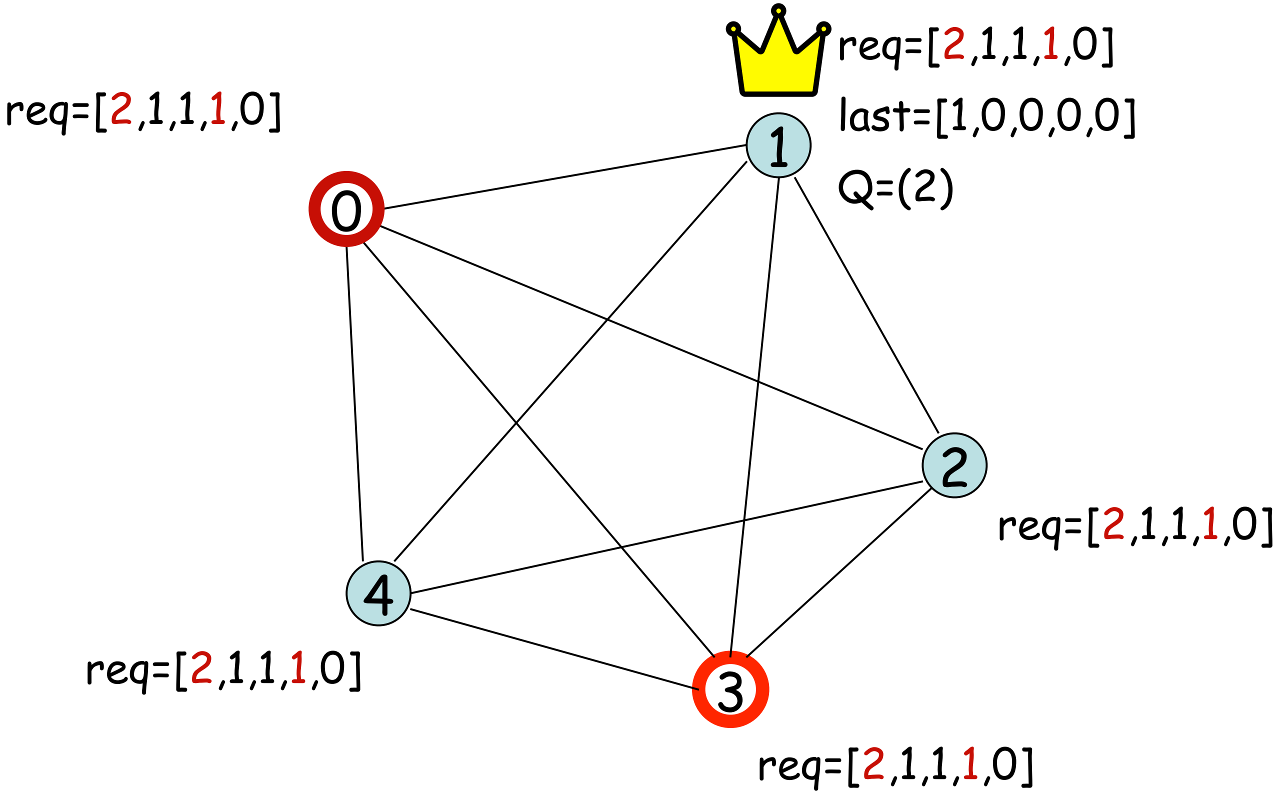
0 prepares to exit CS

Example of Suzuki-Kasami Algorithm Execution



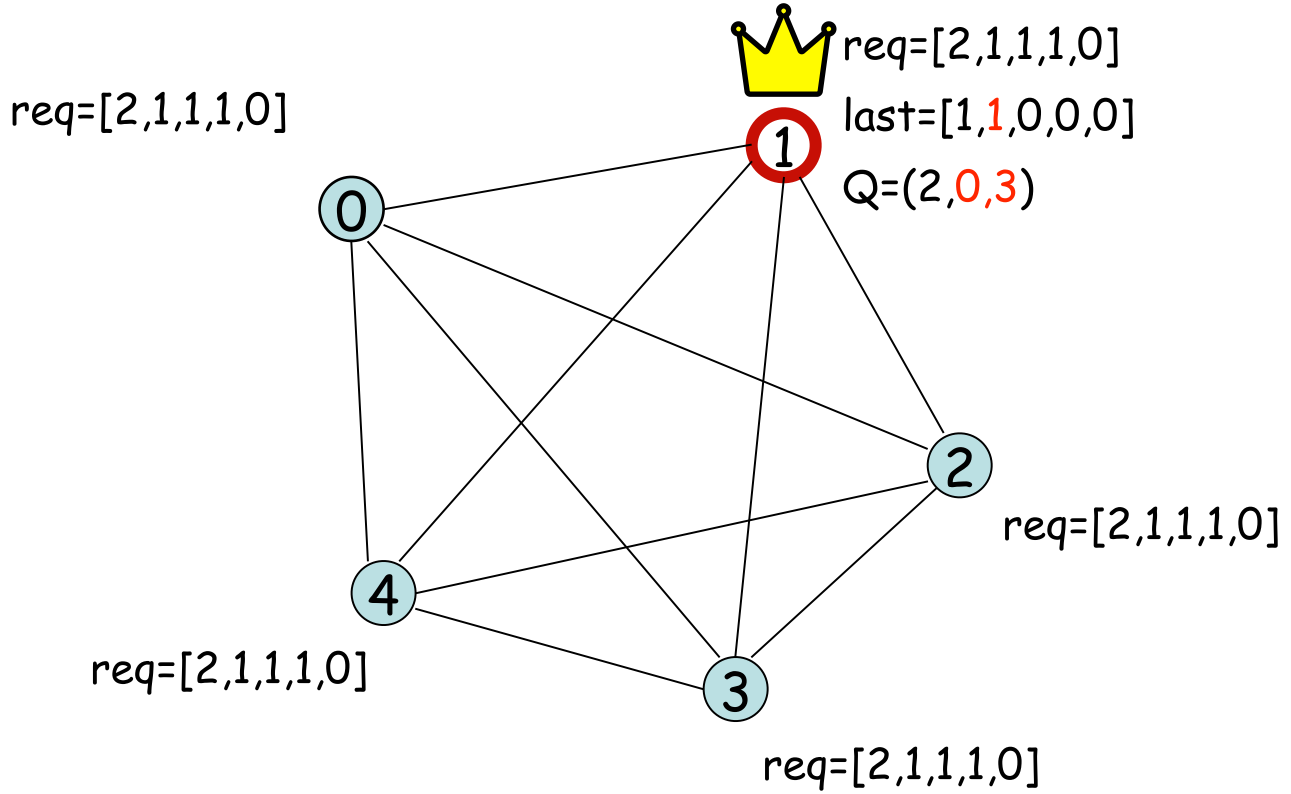
1 receives the token (Q and last) from 0

Example of Suzuki-Kasami Algorithm Execution



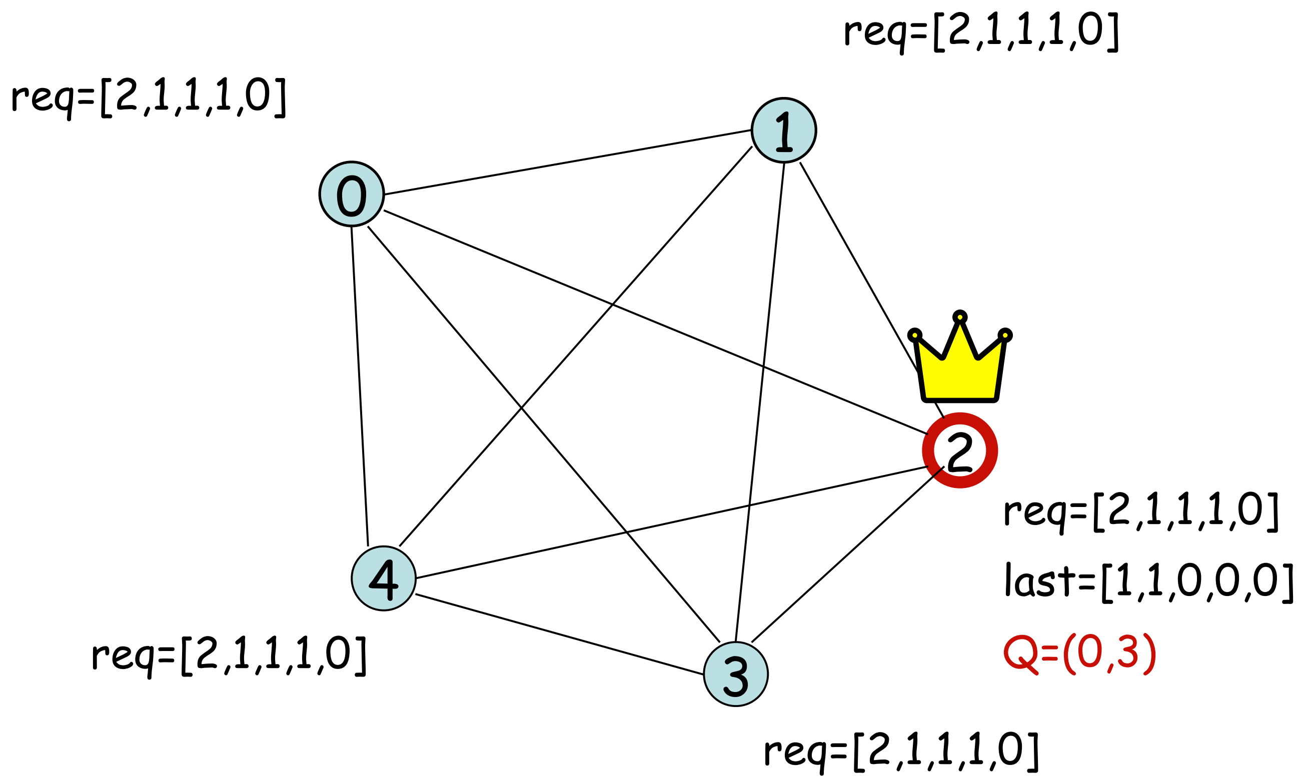
0 and 3 send requests

Example of Suzuki-Kasami Algorithm Execution



1 exists critical section and prepares to pass the token

Example of Suzuki-Kasami Algorithm Execution



2 receives the token from 1

Summary and advantages

Token-based + queue :

- Satisfies ME1 to ME3

WHY?-> Homework

- Less messages: N by CS

WHY?

- Question: is this algorithm fair? All messages received during the CS are enqueued at the same position, cannot we do better?
- Note: index can be bound
- Note 2: A similar algorithm was published by Ricart and Agrawala at the same period

Distributed Mutual Exclusion

1 – Introduction

2 – Solutions Using Message Passing

3 – Token Passing Algorithms

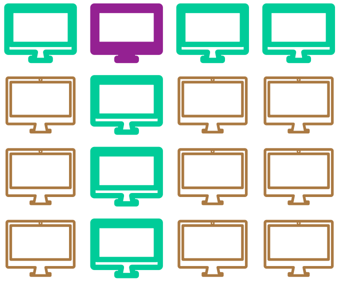
4 – A Taste of Quorum-Based Algorithms

Quorum based algorithms

- Some algorithms have a **sublinear** **O(sqrt N)** message complexity.
- Each process is required to obtain permission from only a **subset** of peers
- To end this course: a gentle taste of these algorithms and the problems they have to face

A quorum-based algorithm for grids

N processes are placed on a two-dimensional grid



they can only communicate with processes of either the same row or the same column

The REQUEST->ACK->RELEASE principle

1) A process broadcasts a request to its row and column. Therefore O(sqrt N) messages
2) It waits for an ack from everybody on the row and the column before entering CS
3) It broadcasts a release when exiting CS

Rules for sending an ack

Each process maintains its own queue of pending requests.

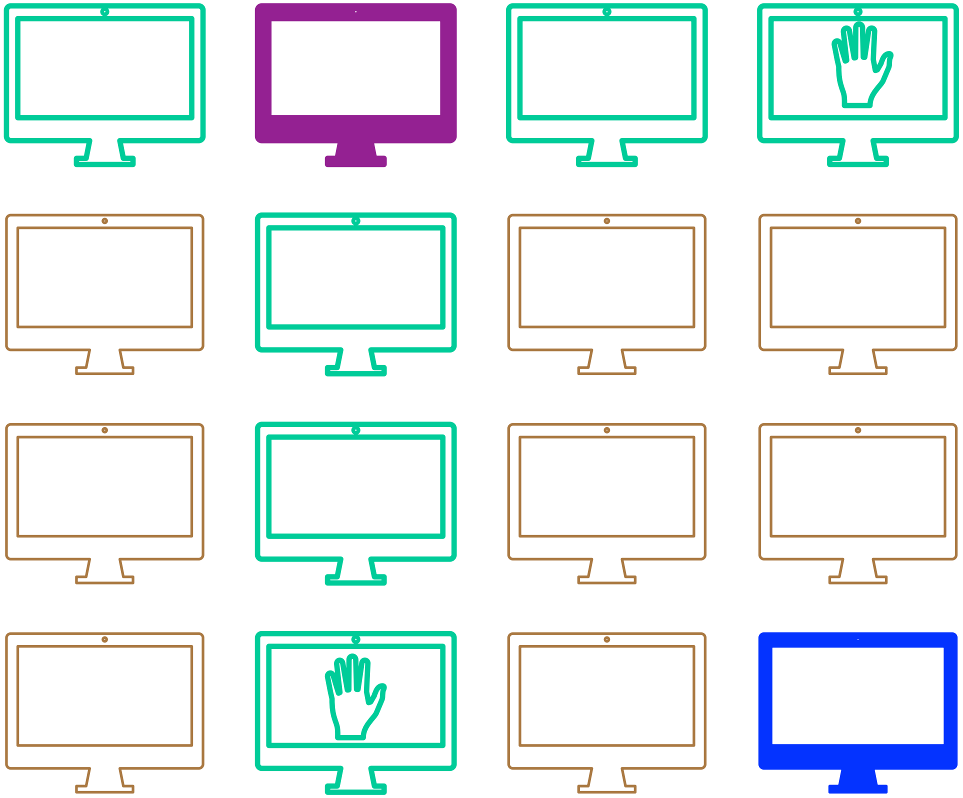
When Pi receives a REQUEST from Pj:

1) if the queue is empty, it sends an ACK to Pj
2) in any case, Pj enqueues Pi

When Pi receives a RELEASE from Pj:

1) it dequeues Pj
2) if the queue is not empty, it sends an ACK to the process Pk at the head of the queue

Example



1) 🖥 broadcasts REQUEST to all the 🖥

2) each 🖥 answers ACK to 🖥 who then enters CS

3) 🖥 broadcasts REQUEST to its row and column

4) all but the two ✋ answer ACK

5) 🖥 broadcasts RELEASE to all the 🖥

6) the two ✋ answer ACK to 🖥, who then enters CS

Homework

This algorithm satisfies safety WHY?

This algorithm <u>does not</u> satisfy liveness WHY?

BONUS (technical) : read about Maekawa's algorithm to learn how to recover liveness.

Conclusion

- What you should have learnt:
  - design distributed algorithms
  - write a few classical ones
  - analyse and reason upon an algorithm
    More or less formal approaches (diagrams vs formal reasoning)

- A word on more systematic formal approaches
  - Model checking
  - Framework for reasoning on algorithms, e.g. TLA+

Homeworks

1) Prove that Suzuki-Kasami algorithm verifies the three properties of mutual exclusion

Note: have a look at the similar proofs in the course

2) Explain why the quorum-based algorithm presented in this lecture satisfies safety, but does not satisfy liveness.