




# Distributed Information Processing

2<sup>nd</sup> Lecture

Eom, Hyeonsang (엄현상)  
Department of Computer Science  
& Engineering  
Seoul National University



# Outline

- Clock
  - Logical Clocks & Ordering of Events
  - Vector Clocks
- Q&A

# Partial Ordering of Events

## ■ Assumptions

- System Consists of  $N$  Processes:

$$p_i \quad (i = 1, \dots, N)$$

- System Is Composed of a Collection of Events

- Examples of events

- Execution of a subprogram
- Execution of a single machine instruction
- Sending or receiving a message

$$history(p_i) = h_i = \langle e_i^k \mid k = 1, \dots \rangle \quad (i = 1, \dots, N)$$

# Partial Ordering of Events [Lamport78]

## ■ “Happened Before” Relation ( $\rightarrow$ )

(1)  $k < l \Rightarrow$  (if)  $e_i^k \rightarrow e_i^l \quad (i = 1, \dots, N)$

(2)  $e_i = \text{send}(m), e_j = \text{receive}(m)$  for Message  $m$   
 $\Rightarrow e_i \rightarrow e_j$

(3)  $e \rightarrow e' \wedge e' \rightarrow e'' \Rightarrow e \rightarrow e''$  Transitive

\*  $e \not\rightarrow e$  Irreflexive

## □ Concurrent Relation ( $\parallel$ )

$$e \not\rightarrow e^* \wedge e^* \not\rightarrow e \Leftrightarrow e \parallel e^*$$

# Logical Clocks [Lamport78]

## ■ Ways of Assigning Numbers to Events

$LC_i(e)$ , where  $e$  is an event in process  $p_i$

## ■ Clock (Correctness) Condition

$$e \rightarrow e^* \Rightarrow LC(e) < LC(e^*)$$

□ Satisfied if the following two conditions hold:

$$(1) k < l \Rightarrow LC(e_i^k) < LC(e_i^l) \quad (i = 1, \dots, N)$$

$$(2) e_i = \text{send}(m), e_j = \text{receive}(m) \text{ for Message } m$$

$$\Rightarrow LC(e_i) < LC(e_j)$$

# Logical Clocks (Cont'd) [Lamport78]

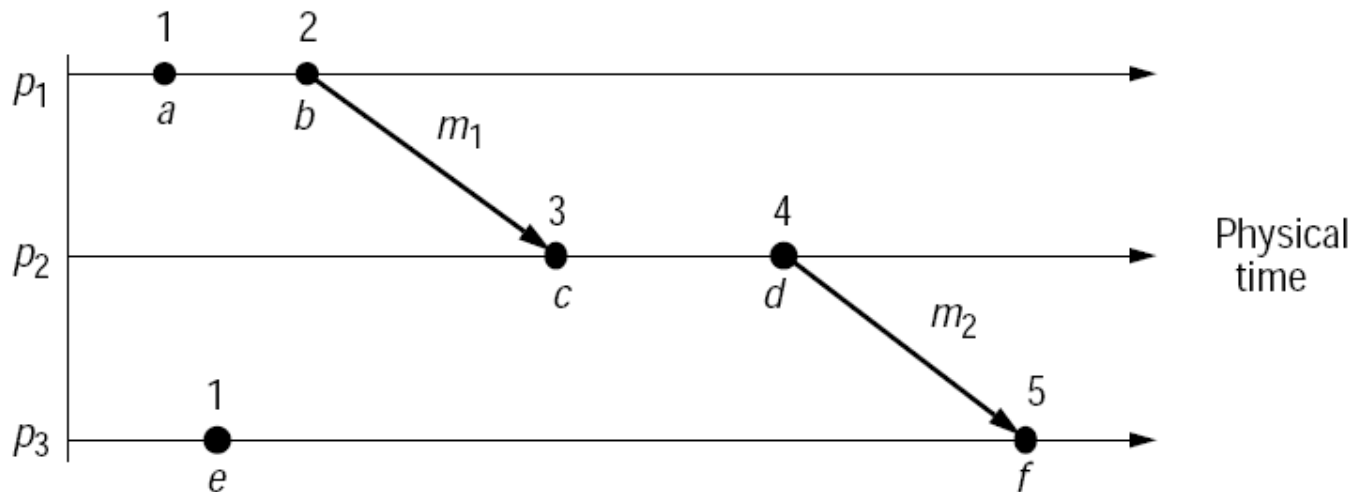
## Implementation Rules

(1)  $LC(e_i^{k+1}) = LC(e_i^k) + 1$  for Successive Events  $e_i^k, e_i^{k+1}$

( $i = 1, \dots, N$ )

(2)  $e_i = \text{send}(m), e_j = \text{receive}(m)$  for Message  $m$

$\Rightarrow t = LC(e_i) \in m, LC(e_j) = \max\{LC(e_j^{l-1}), t\} + 1$



# Ordering the Events Totally

## [Lamport78]

Arbitrary

### ■ Total Ordering ( $\Rightarrow_{t.o.}$ )

$$e \Rightarrow_{t.o.} e^* \text{ iff } LC_i(e) < LC_j(e^*) \vee \{LC_i(e) = LC_j(e^*) \wedge i < j\}$$

$$e \rightarrow e^* \Rightarrow e \Rightarrow_{t.o.} e^*$$

### □ Use Case: Mutual Exclusion Problem

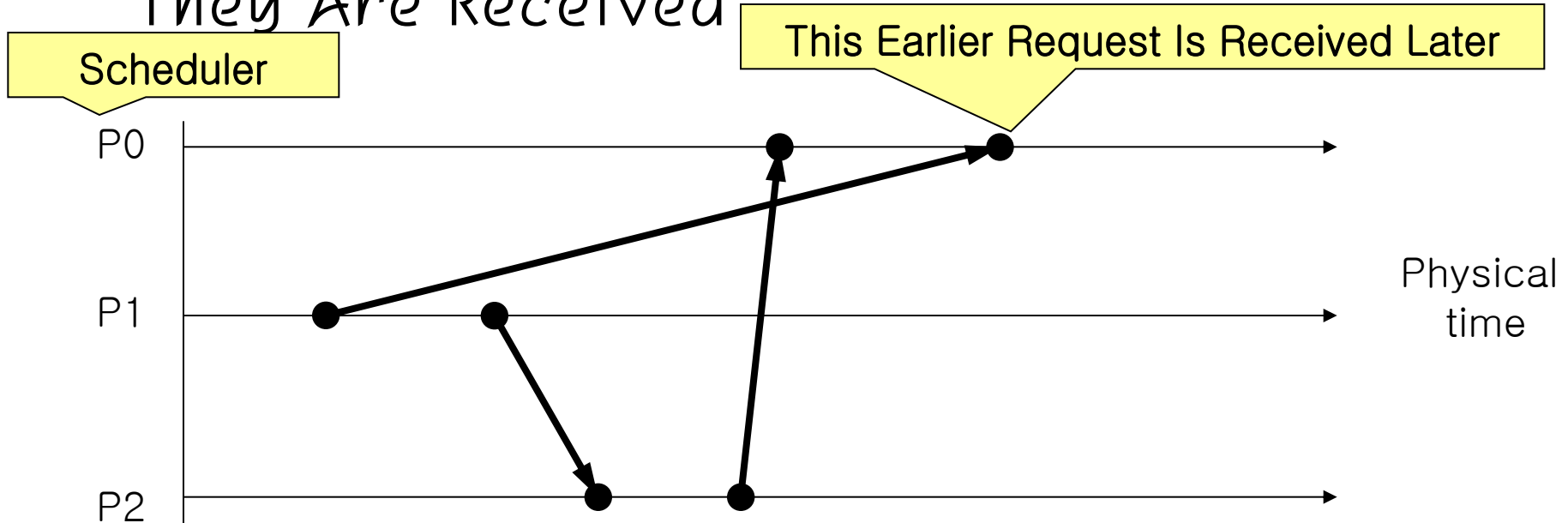
#### ■ Processes sharing a single resource

- (1) A process using it must release it before the following use
- (2) The processes must use it in their request order
- (3) If it is eventually released, every request is eventually granted

# Ordering the Events Totally (Cont'd) [Lamport78]

## ■ Nontrivial Scheduling Problem

- The Requests Cannot Be Granted in the Order They Are Received





# Ordering the Events Totally

(Cont'd)

■ Nontrivial

- (1) A process using it must release it before the following use
- (2) The processes must use it in their request order
- (3) If it is eventually released, every request is eventually granted

## □ Assumptions for an Algorithm

- In-order message delivery
- Each process' request queue
  - Initially containing a request with  $T_0:P_0$

Smallest Time Value:  
Scheduler ID

## □ Algorithm

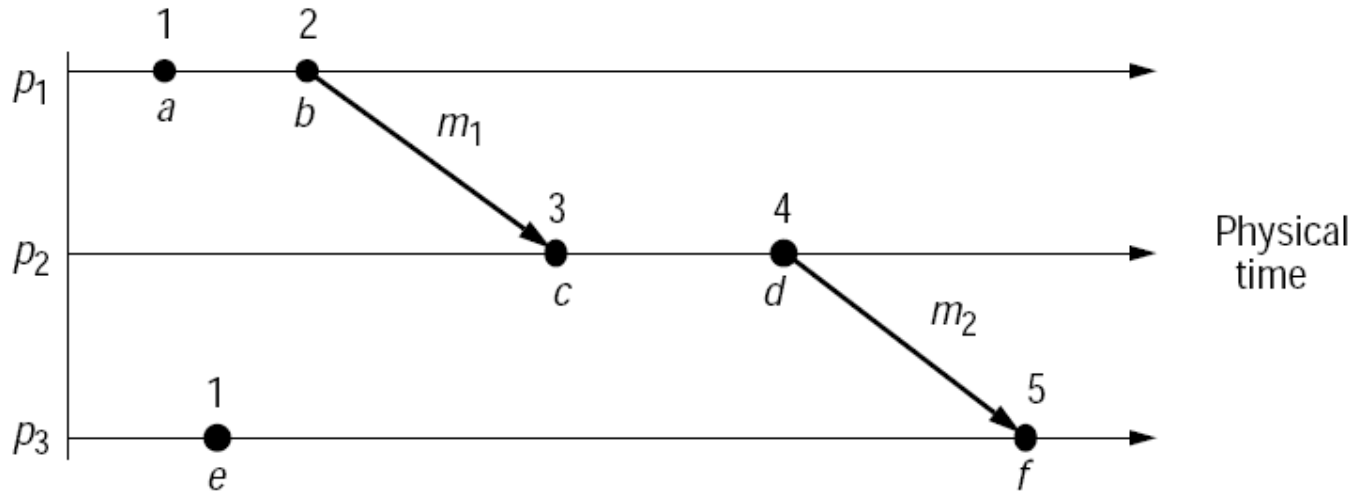
Send Time

- $P_i$  sends a  $T_m:P_i$  Req to all others and puts it in the queue
  - Each receiver puts it in the queue and sends a timestamped Ack to the requestor
- When releasing the resource,  $P_i$  removes the Req and sends a timestamped Rel to all others
  - Each receiver removes  $P_i$ 's Req from its queue
- The  $T_m:P_i$  Req is granted if it is the first in  $\Rightarrow$ , and  $P_i$  has received a message from all others timestamped later than  $T_m$

# Vector Clocks

## ■ Shortcomings of Lamport's Logical Clock

$$LC_i(e) < LC_j(e^*) \not\Rightarrow e \rightarrow e^*$$



# Vector Clocks (Cont'd)

## Implementation Rules

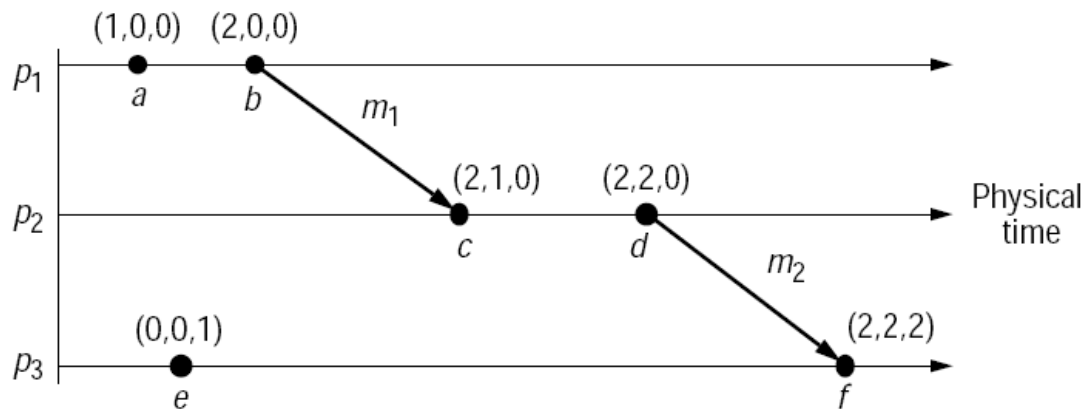
(1)  $VC_i[j] = 0$  ( $i, j = 1, \dots, N$ )

(2)  $VC_i(e_i^{k+1})[i] = VC_i(e_i^k)[i] + 1$  for Successive Events  $e_i^k, e_i^{k+1}$   
 ( $i = 1, \dots, N$ )

(3)  $e_i = send(m), e_j^l = receive(m)$  for Message  $m$

Increment the j-th Elt

$$\Rightarrow t = VC_i(e_i) \in m, VC_j(e_j^l) = \max\{VC_j(e_j^{l-1}), t\} +_j 1$$



# Vector Clocks (Cont'd)

## ■ Vector-Timestamp Comparisons

$$VC = VC^* \text{ iff } VC[j] = VC^*[j] \ (j = 1, \dots, N)$$

$$VC \leq VC^* \text{ iff } VC[j] \leq VC^*[j] \ (j = 1, \dots, N)$$

$$VC < VC^* \text{ iff } VC \leq VC^* \wedge VC \neq VC^*$$

## ■ Properties

$$e \rightarrow e^* \text{ iff } VC(e) < VC(e^*)$$

Strong  
Clock  
Condition

### □ Hints for the Proof

$$e \parallel e^* \Rightarrow \neg\{VC(e) \leq VC(e^*) \vee VC(e) \geq VC(e^*)\}$$

(Note That  $VC_i[j] \leq VC_j[j]$ )

# Vector Clocks

## ■ Properties

$e_i \rightarrow e_j$  iff  $VC_i[i] \leq VC_j[i]$ , where  $i \neq j$

Simple  
Clock  
Condition

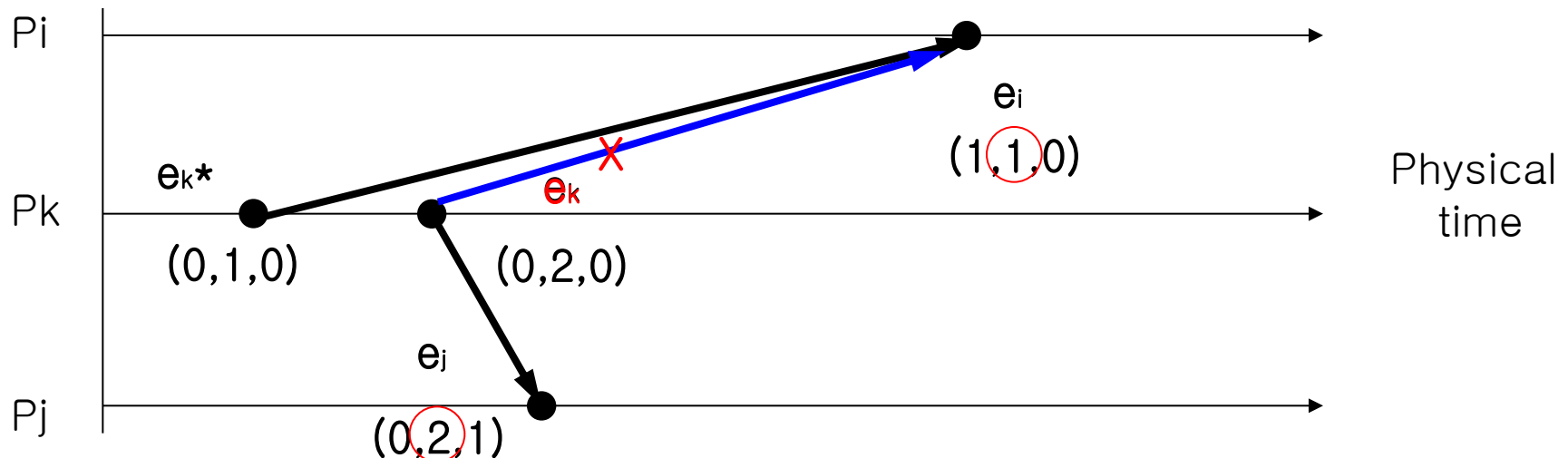
□ Hint for the Proof

(Note That  $VC_i[j] \leq VC_j[j]$ )

$\exists k (\neq j)$  s.t.  $VC_i[k] < VC_j[k]$

Weak Gap-  
Detection

$\Rightarrow \exists e_k$  s.t.  $(e_k \nrightarrow e_i) \wedge (e_k \rightarrow e_j)$



# Vector Clocks (Cont'd)

## ■ Properties

$$\exists k (\neq j) \text{ s.t. } VC_i(e_i)[k] < VC_j(e_j)[k]$$

$$\Rightarrow \exists e_k (\neq e_i) \text{ s.t. } (e_k \nrightarrow e_i) \wedge (e_k \rightarrow e_j)$$

Weak Gap  
Detection

*Suppose That  $i = k$ ,*

$$\exists k (\neq j) \text{ s.t. } VC_k(e_k^*)[k] < VC_j(e_j)[k]$$

$$\Rightarrow \exists e_k (\neq e_k^*) \text{ s.t. } (e_k \nrightarrow e_k^*) \wedge (e_k \rightarrow e_j)$$

$$\text{iff } \exists e_k (\neq e_k^*) \text{ s.t. } e_k^* \rightarrow e_k \rightarrow e_j$$

$$\exists e_k (\neq e_k^* \wedge k \neq j) \text{ s.t. } e_k^* \rightarrow e_k \rightarrow e_j$$

$$\Rightarrow VC_k(e_k^*)[k] < VC_j(e_j)[k]$$

# Vector Clocks (Cont'd)

## ■ Properties

$$\forall (k \neq j), VC_k(e_k^*)[k] \geq VC_j(e_j)[k]$$

Contraposition

$$\Rightarrow \neg \{ \exists e_k (\neq e_k^*) \} \text{ s.t. } e_k^* \rightarrow e_k \rightarrow e_j$$

- If the Monitoring Process Keeps the Value of the  $k$ -th EIT of VC from  $P_k$ , It Can Decide Whether There Is No Event That Happened before a Given Event

