CES-27 Processamento Distribuído

Logical Clock

Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

Outline

- Motivation
- Logical clocks, Partial Ordering and Lamport Algorithm
- Total Ordering and Vector Timestamp (*relógios vetoriais*)

Why clock synchronization?

- In a centralized system
  - Time is not ambiguous
  - If a process wants to know the time, it requests the information to the "core"

- In a distributed system
  - We need to synchronize clocks!
  - Communication between processes is important
 - Cooperation is essential

Why clock synchronization?

- Processes share resources (mutual exclusion)
  - Each process has to access exclusively for a time

- Processes must agree with ordering of events
  - Message m1 of P was sent before of message m2 of Q
  - Transaction T transfers $10,000 from S1 to S2
    - Possible problems:
      - State of S1 is recorded after the deduction and state of S2 is recorded before the addition

        > S1 with -$10,000
        > S2 without change

      - State of S1 is recorded before the deduction and state of S2 is recorded after the addition

        > S1 without change
        > S2 with +$10,000

Why clock synchronization?
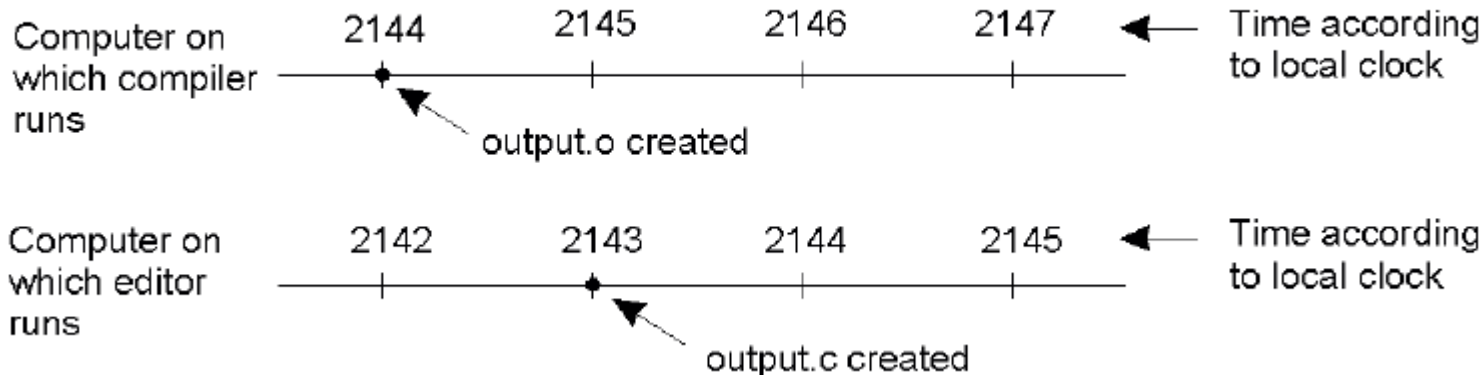
▫ Each process can have a distinct time ⇨ problems!
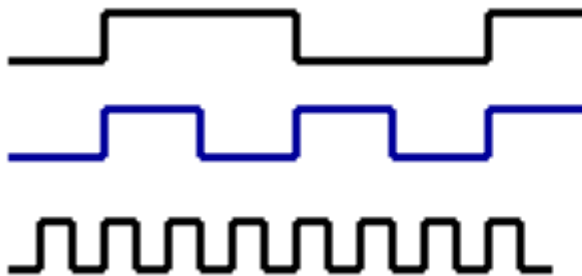
What data is being transmitted? 0101?

Yes, if this is the clock

If this is the clock, then 01110001

• When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Computer on which compiler runs | 2144 | 2145 | 2146 | 2147 | ← Time according to local clock

output.o created

Computer on which editor runs | 2142 | 2143 | 2144 | 2145 | ← Time according to local clock

output.c created

In this case, we will have the old output.o

Physical clocks

- Physical clocks in computers are realized as **crystal oscillation counters** at the hardware level
- Problems:
  - **Skew**: Instantaneous difference between the readings of two clocks

Network
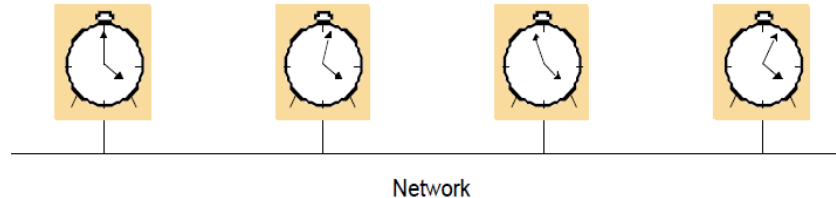
Each computer can have a different local time!

  - **Drift**: Difference in the rate at which two clocks count the time
    - Due to physical differences in crystals, plus heat, humidity, voltage, etc.
    - Accumulated drift can lead to significant skew

  - **Clock drift rate**: Difference in precision between a perfect reference clock and a physical clock
    - Usually 10^{-6} sec/sec
    - 10^{-7} to 10^{-8} for high precision clocks

Clock synchronization

- In a geographically distributed system, it is difficult to synchronize the <u>physical clocks</u> of different processors precisely
  - ▫ Because of uncertainty in communication delays between them

- Software-based solutions to synchronize physical clocks:
  - ▫ Cristian's algorithm
  - ▫ Berkeley algorithm   *Out of our scope*

- Synchronization in distributed systems
  - ▫ For many applications, we are not interested in the precise moments in time at which events occur, but only in the **ordering of these occurrences in time**
 - ▫ It is rare to use physical clocks
  - ▫ We use logical clocks!

Outline

- Motivation
- Logical clocks, Partial Ordering and Lamport Algorithm
- Total Ordering and Vector Timestamp (*relógios vetoriais*)

Logical clocks

- A logical clock C maps occurrences of events in a computation to a partially ordered set such that
  - *a => b* then *C(a) < C(b)*
  - Relation **precedes** (=>)

  - A total order among events can be observed in a single processor
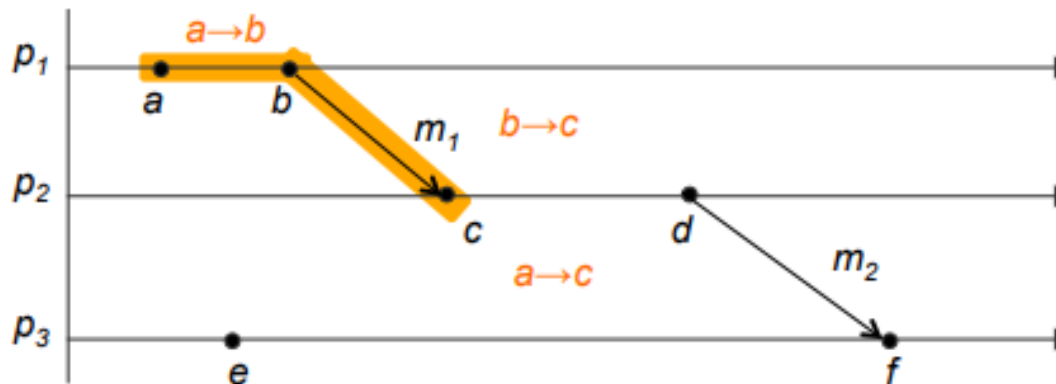
- Lamport:
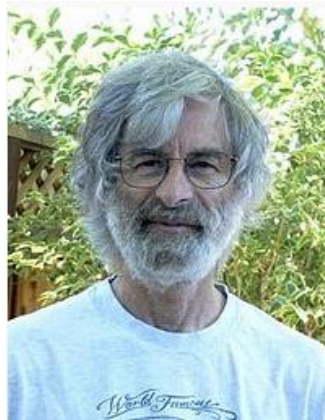
**Leslie Lamport
(1941 - ?)**

Turing Award in 2013

  - In a true distributed system only a partial order can be determined between events

  - Happened-before relation
    - Causal ordering relation or potential causal ordering relation.

Happened-before relation

- a => b
  - If **a** and **b** are events of the same process and **a** happens before **b**.
  - If **a** is the sending of a message by a process and **b** is the receiving of that message by another process

- a => b
  - It means that it is possible that **a** causally affects **b** event.
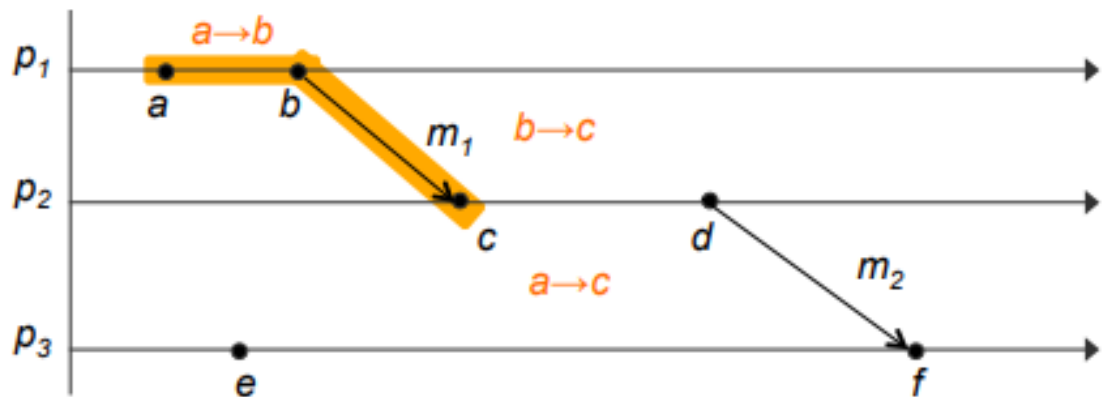
Happened-before relation

- Properties:
  - ▫ If a => b and b => c, then a => c

  - ▫ Events a and b are **concurrent** (we say a||b) if
    - ~(a => b) and ~(b => a)
    - i.e. a and b are not causally related

  - ▫ Examples:
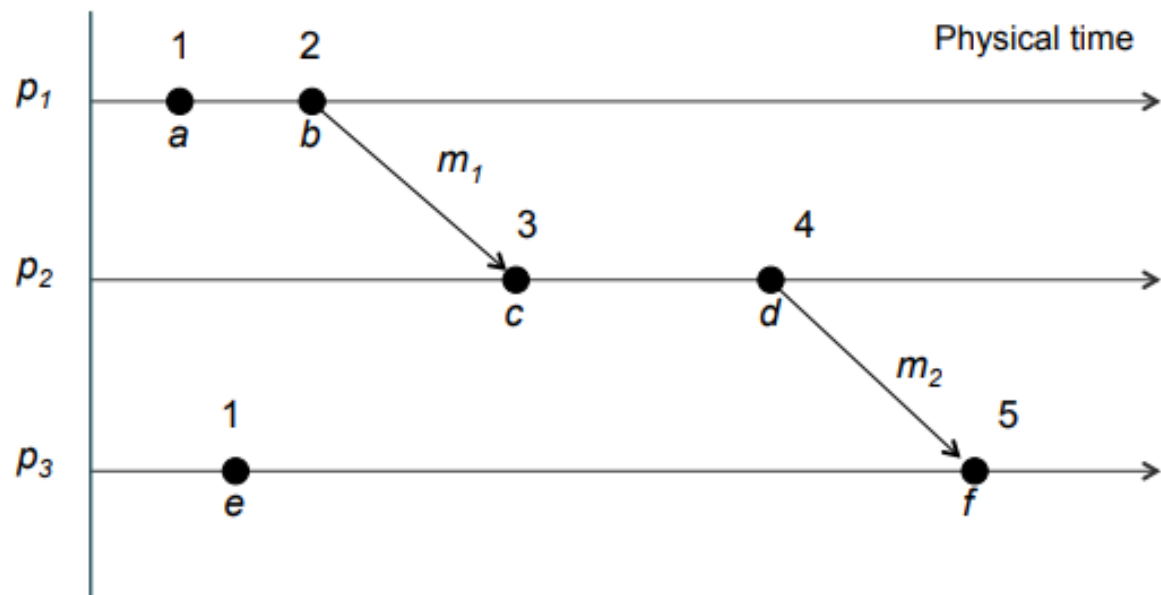   - c||e
    - a||e
    - b => f

Lamport algorithm for logical clocks

- Each process P_i has **one logical clock C_i** used to **measure** the **local time** in terms of **events**
  - ✓ Initially **o** (zero) and only **increases**.

- Each message m sent from P_i is labeled with C_i and provides the identifier of **processor i**.
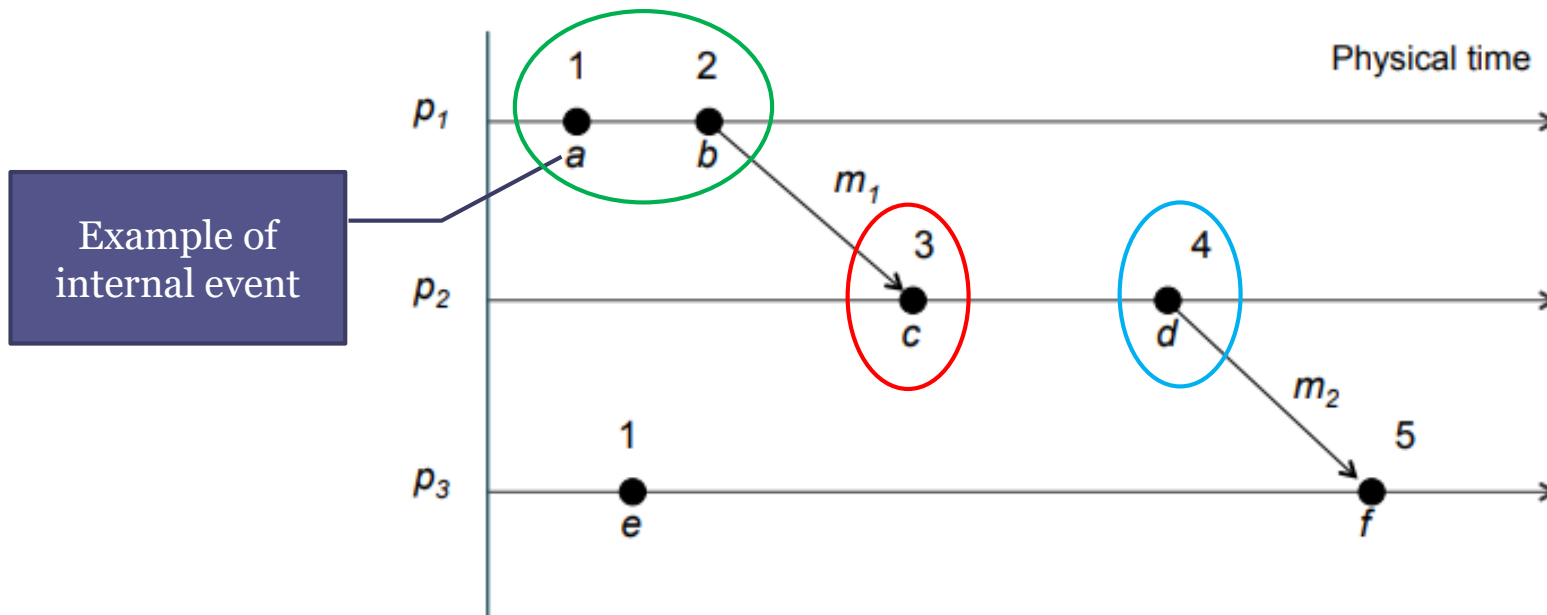 - ✓ In that way, **messages** constitutes the **triple** $<m, C_i, i>$

Examples:
$<m_1, 2, p_1>$
$<m_2, 4, p_2>$

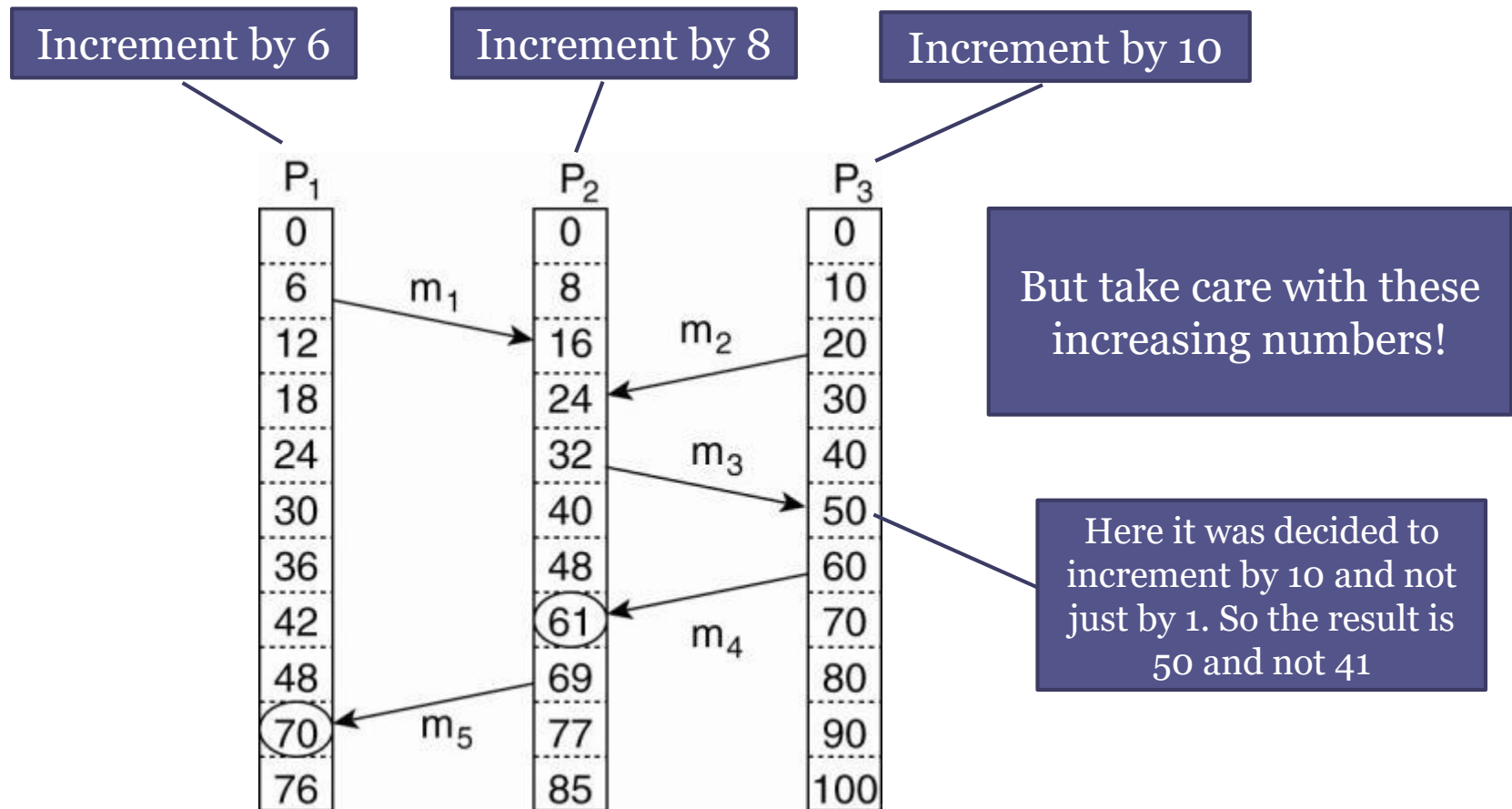# Lamport algorithm

1. When process P_i receives one **message** $<m, C_j, j>$ *
    - ✓ It adjusts the **clock** C_i to $(max(C_i, C_j) + 1)$
    - ✓ This is the **time** of the **event** of **message receiving**

2. When process P_i sends a message $<m, C_i, i>$ *
    - ✓ C_i is incremented before inclusion in the **triple**.
    - ✓ This is **the transmission time** of the **message**.

3. C_i can be incremented arbitrarily between **communication**   *
   **events** of P_i.   Note: Here we increment by 1.

Example of internal event

Physical time

Lamport algorithm

- No need to increment by 1, but any positive number

Increment by 6   Increment by 8   Increment by 10

But take care with these increasing numbers!

Here it was decided to increment by 10 and not just by 1. So the result is 50 and not 41

Lamport algorithm

- Results
  - **Events** (internal, sending and receiving) are **labeled** in a **consistent** and **correct way**
    - In relation to the **transmission** and **receiving** of **messages**

  - There must have at least **one tic (increment)** between events in a process.
    - Each **message line** must cross at least **one line of tic**.

  - For every **events a**, **b**
    - If $a => b$ then $C(a) < C(b)$
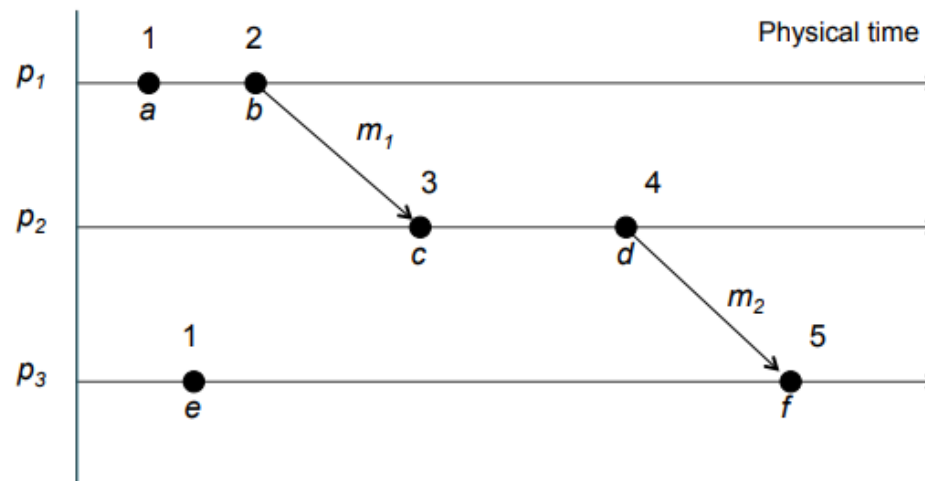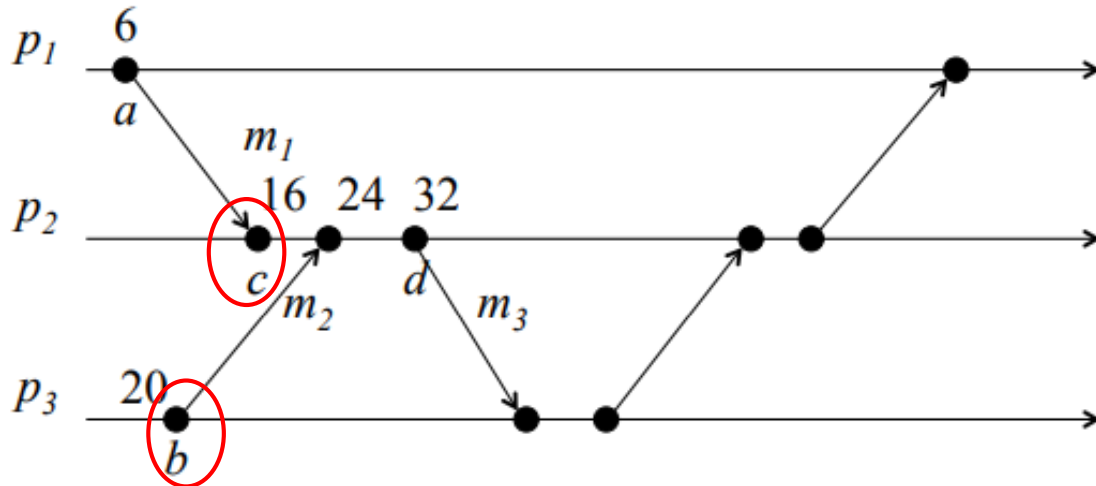
Lamport algorithm

- Results
  - In other words, if $C_i(a)$ is the **time** of event a in process P_i and…
    - If a, b are events in P_i and $a \Rightarrow b$ then $C_i(a) < C_i(b)$
    - If a is the sending of message by P_i and b is the receiving of message by P_j then $C_i(a) < C_j(b)$

    - Note that $C(a) < C(b)$ does not imply that $a \Rightarrow b$
      - We can't infer a causally preceded b

E.g. Is the sending of m_2 (C(b)=20) causally related to receiving of m1 (C(c)=16)? No!
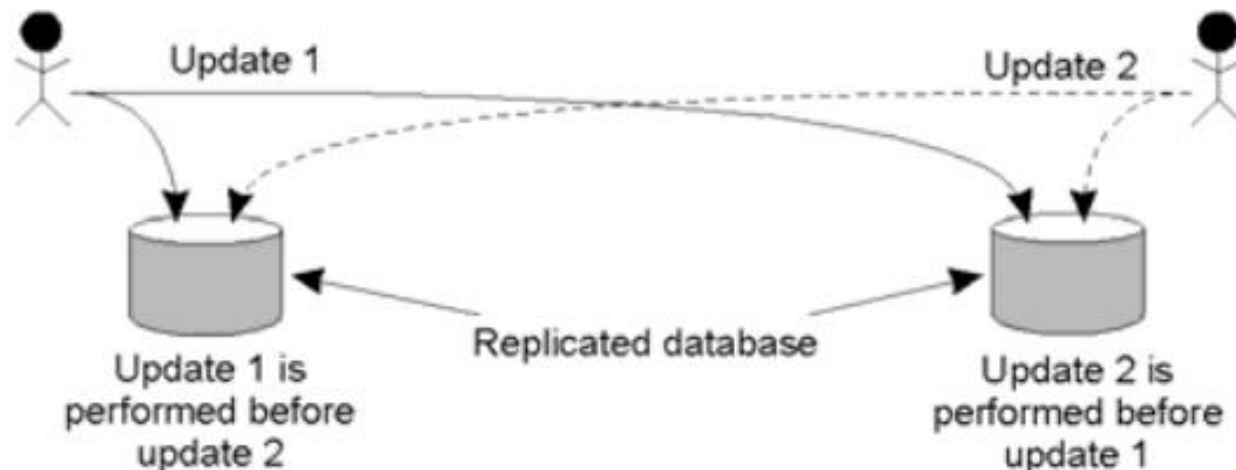c => b

Other applications of partial ordering

- ## Detection of **race conditions**
  - The output is dependent on the sequence or timing of other uncontrollable events.
  - E.g. A **race condition** arises if two or more processes access the same variables or objects concurrently and at least one does updates
    - Suppose X is initially 5
    - After finishing, X=6!
    - We "lost" one update

Thread t1	Thread t2
□ LOAD X	□ ...
	□ LOAD X
□ ADD 1	□ ADD 1
	□ STORE X
□ STORE X	
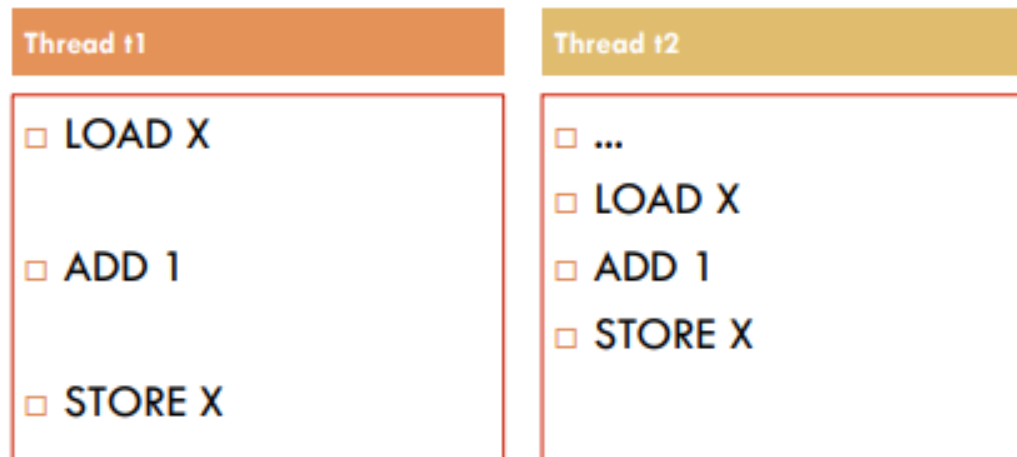
Other applications of partial ordering

- ## Multicast **protocols**
  - Many of them require **causal delivery**
  - E.g. A database replicated across several sites
    - Issue: update operations must be performed in the same order at each copy, so that all copies are exactly the same
      - E.g. Account = 1000
        - Update1: process P1 adds 100
        - Update2: process P2 increments by 1%
        - Error: Replica1 with 1111 and Replica2 with 1110

Update 1   Update 2

Replicated database

Update 1 is performed before update 2

Update 2 is performed before update 1

Other applications of partial ordering

- Consistent **global snapshot**
  - The <u>global state of a distributed system</u> is a collection of the local states of the processes and the states of the communication channels of all the processes
    - **A <u>snapshot</u> of an execution of a distributed algorithm is a configuration of this execution, consisting of the local states of the processes and the messages in transit.**
  - Recording a snapshot of a distributed system is important for many applications
    - To determine offline properties that will remain true as soon as they have become true, such as **deadlock, termination** or **garbage collector**.
    - For check-pointing to restart after a failure (**failure recovering**), or **debugging**, or **simulation**.
  - Algorithms:
    - Chandy-Lamport algorithm
    - Lai-Yang algorithm

    Out of our scope
  - **Consistency** in a distributed system is basically a problem of correctly reflect **causality**!

Outline

- Motivation
- Logical clocks, Partial Ordering and Lamport Algorithm
- Total Ordering and Vector Timestamp (*relógios vetoriais*)

Total ordering of events

- We wish to record, **locally** in a **process**, all **events** that can (causally) **precede** one particular **event b**
  - The **causal history** of the **b event**

- Can **Lamport's logical clock** provide this information?
  - $C(p_2) < C(r_4)$ implies that p_2 can causally affect r_4? No!
  - The **timestamp algorithm** (Lamport's logical clock) allows a **partial ordering** of **events**

- Without knowing the **precedence relations**, we **can't tell** if **two events a** and **b** are **causally related (dependent)**
  - If $C(a) < C(b)$ then $a => b$ or $a \mid\mid b$

- We need a **mechanism** that represents **causality**
  - Instead of using **scalar time**, we can use **vector timestamps** to record the last known **clock values**
   - And hence, the **precedent causally events** of all other processes

Vector Timestamp

- Each process i has a Vector Timestamp **VT** of size **N** (number of processes): VT_i
  - So VT_i has **N** clock values. A clock value for each process **k =1,..,N**

- Event **a** is labeled in VT_i as $VT_i(a)$
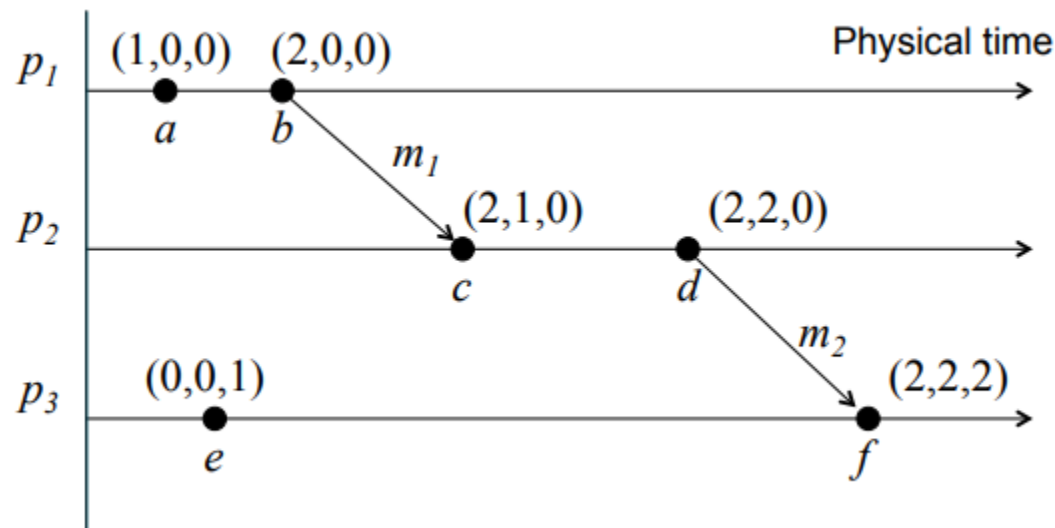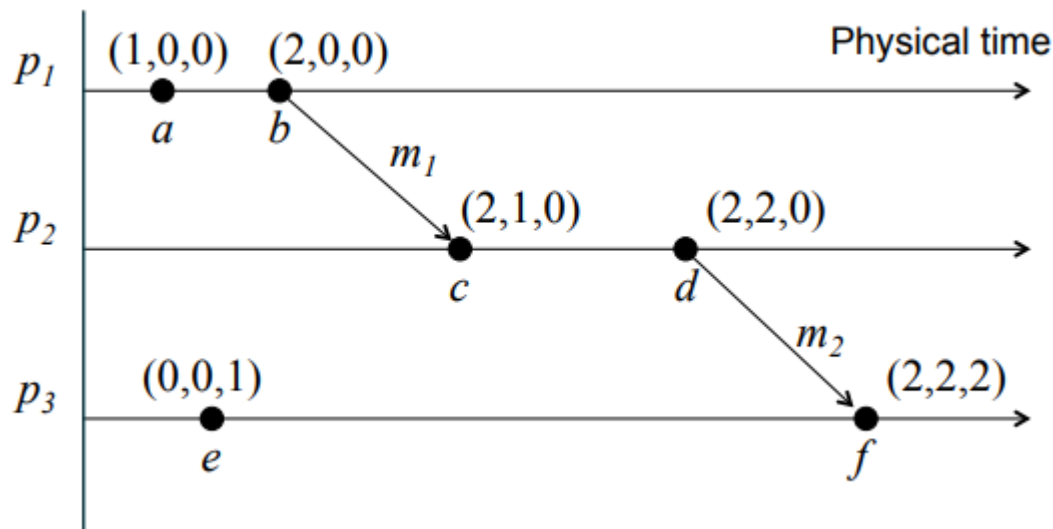
Vector Timestamp

Vector timestamp of process i:

1. Initially $VT_i[k]$=0 for all processes $k = 1,..,N$.
2. Increment $VT_i[i]$ for each internal event in process i.
3. Increment $VT_i[i]$ and attach to message m before sending m.
4. Update VT_i on receiving of message m with $VT(m)$
   - ✓ Increment $VT_i[i]$
   - ✓ $VT_i[k] := \max (VT_i[k], VT(m)[k]$) for $k = 1,.., n$ *if $i \neq k$*
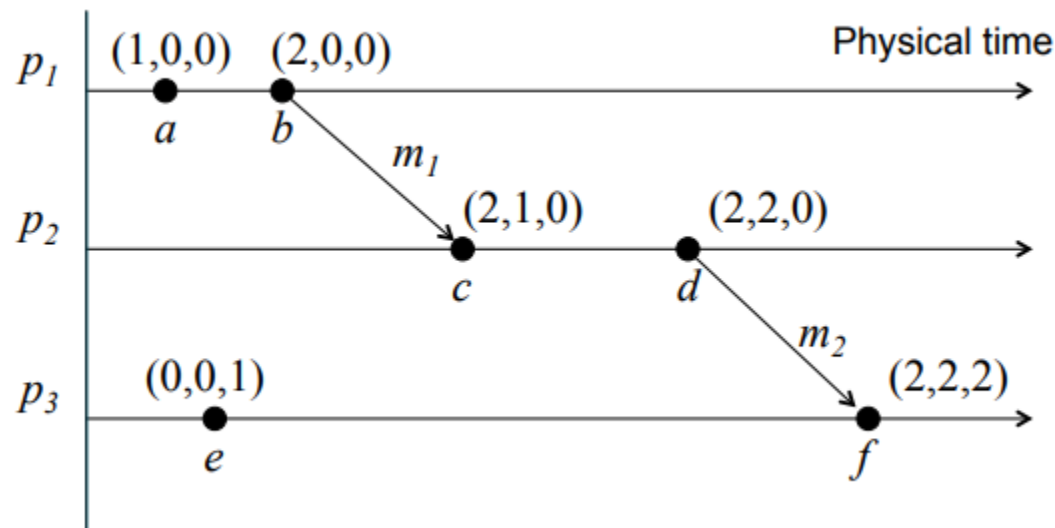
Properties of Vector Timestamp

- $VT_i[i]$: number of events that have taken place at process **i**
- $VT_i[j]$: number of events that process **i** knows have taken place at process **j** (i.e., that have potentially affected process **i**)

- Comparing vector timestamps:
  - $VT_i <= VT_j$ if $VT_i[k] <= VT_j[k]$ for all k

  - $VT_i < VT_j$ if $VT_i <= VT_j$ and $VT_i \neq VT_j$

  - $VT_i || VT_j$ if not $VT_i <= VT_j$ and not $VT_j <= VT_i$

Examples:
c||e
a||e
b => f

Scalar vs. Vector Timestamp

- **Scalar timestamps** provide an **ordering** that is consistent with **causality**
  - ▫ But don't characterize **causality**
  - ▫ Hence they **cannot** be **used** to **proof events** that are not causally related

- **Vector timestamps** represent **causality** in a precise form!

References

- Lamport, Leslie. **"Time, Clocks, and the Ordering of Events in a Distributed System"**, CACM, Vol.21, No.7, July 1978, pp.558-565.
- Mattern, Friedemann. "**Virtual time and global states of distributed systems**", *Parallel and Distributed Algorithms* 1.23 (1989): 215-226.
- Fidge, Colin. "**Logical time in distributed computing systems"**, *Computer* 24.8 (1991): 28-33.