Synchronization

Physical and Logical Clocks

Introduction

- Electronic clocks in most servers and network devices keep inaccurate time.
- Small errors can add up over a long period.
- Assume two clocks are synchronized on January 1.
  - One of the clocks consistently takes an extra 0.04 milliseconds to increment itself by a second.
  - On December 31 the clocks will differ by 20 minutes

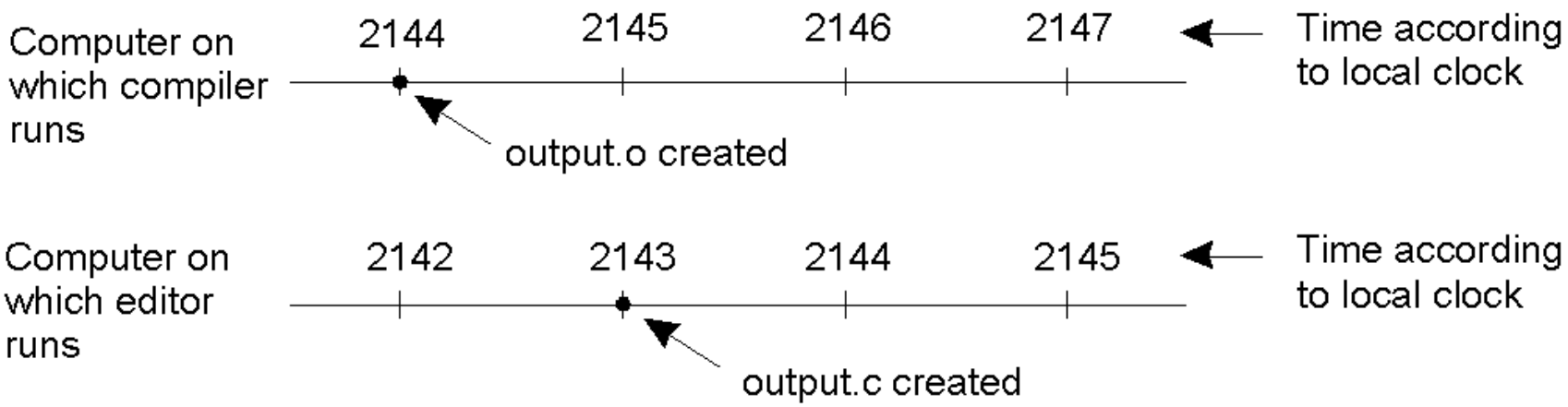
Clock Synchronization

- In a centralized system:
  - Time is unambiguous. A process gets the time by issuing a system call to the kernel. If process *A* gets the time and later process *B* gets the time then the value *B* gets is higher than (or possibly equal to) the value *A* got.
  - Example: UNIX ***make*** examines the times at which all the source and object files were last modified.
    - If time (*input.c*) > time(*input.o*) then recompile *input.c*
    - If time (*input.c*) < time(*input.o*) then no compilation is needed.

Clock Synchronization

- In a distributed system, achieving agreement on time is not easy.
- Assume no global agreement on time. Let's see what happens:
  - Assume that the compiler and editor are on different machines
  - *output.o* has time 2144
  - *output.c* is modified but is assigned time 2143 because the clock on its machine is slightly behind.
  - *Make* will not call the compiler.
  - The resulting executable will have a mixture of object files from old and new sources.

Clock Synchronization



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

Physical Clocks

- Every computer has a local clock -- a timer is more appropriate.
- A timer is usually a **precisely machined quartz crystal**. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension.

- Associated with each crystal are **two registers, a counter and a holding register**. Each oscillation of the crystal decrements the counter by one. When the counter reach zero, an interrupt is generated and the counter is reload from the holding register. Each interrupt is called a **clock tick**.
- For PC, the clock ticks are 54.9 msec apart (18.3 per second)

- Within a single CPU system, it does not matter much if this clock is off by a small amount. Since all processes used the same clock, they will be internally consistent.

- In a distributed system, although the frequency at which a crystal oscillator runs is fairiy stable, there is no way to guarantee that the crystals in different computer all run at exactly the same frequency.

- Crystals running at different rates will results in clocks gradually out of sync and give different value when read out. This differences in time value is called **clock skew**.

Logical Clocks

- Lamport (1978) showed that clock synchronization is possible and presented an algorithm for it. He also pointed out that clock synchronization need not be absolute.
- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.
- What usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which event occur.

- Therefore, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. It is conventional to speak of the clocks as logical clocks.
- On the other hand, when clocks are required to be the same, and also must not deviate from the real time by more than a certain amount, these clocks are called physical clocks.

Clocks

- On each of the computers, the crystals will run at slightly different frequencies, causing the software clocks gradually to get out of sync. This is called ***clock skew* or *clock drift***.
- Ordinary quartz clocks drift by ~ 1 sec in 11-12 days. (10^{-6} secs/sec).
- High precision quartz clocks drift rate is ~ 10^{-7} or 10^{-8} secs/sec

Universal Coordinated Time

- TAI is stable and available to anyone who wants to go to the trouble of buying a cesium clock.
- Problem:86,400 TAI seconds is now about 3 *msec* less than a mean solar day since a solar day is getting longer.
- Using TAI means that over the course of years, noon would get earlier and earlier, until it would eventually occur in the wee hours of the morning.

Universal Coordinated Time

- Since the mean solar day gets longer, the BIH made necessary corrections (by introducing leap seconds) resulting in Universal Coordinated Time (UTC).
- UTC is provided to those who need precise time:
  - National Institute of Standard Time (NIST) operates a shortwave radio station with call letters WWV from Fort Collins, Colorado with +1 to –1 msec accuracy.
  - Earth satellites also offer a UTC service: accurate to 0.5 msec
  - By telephone from NIST: cheaper but less accurate.
  - Need to compensate for signal propagation delay.

Universal Coordinated Time

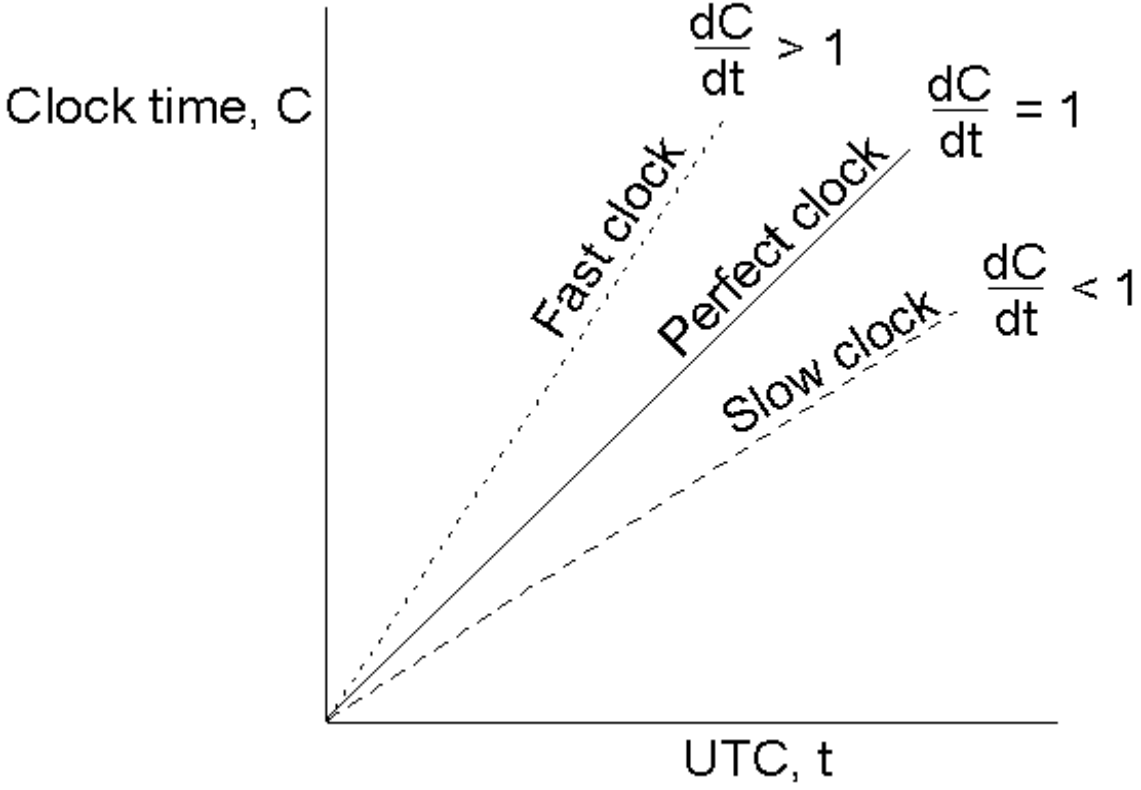
- If one machine has a WWV receiver, the goal becomes keeping all the other machines synchronized to it.
- If no machines have WWV receivers, each machine keeps track of its own time; the goal is to keep the machines synchronized.

Physical Clock Synchronization

- Model of system assumed by clock synchronization algorithms:
  - When UTC is *t*, the value of the clock on machine *p* is $C_p(t)$.
  - In a perfect world, $C_p(t) = t$ for all p and all t;.
  - *d*C/*dt* denotes the ratio of the change in the clock time to actual time.
    - *In "perfect world" dC/dt should be one*

Physical Clock Synchronization

- The relation between clock time and UTC when clocks tick at different rates.



Physical Clock Synchronization

- Each clock has a maximum drift rate ρ.

  $1-\rho <= dC/dt <= 1+\rho$

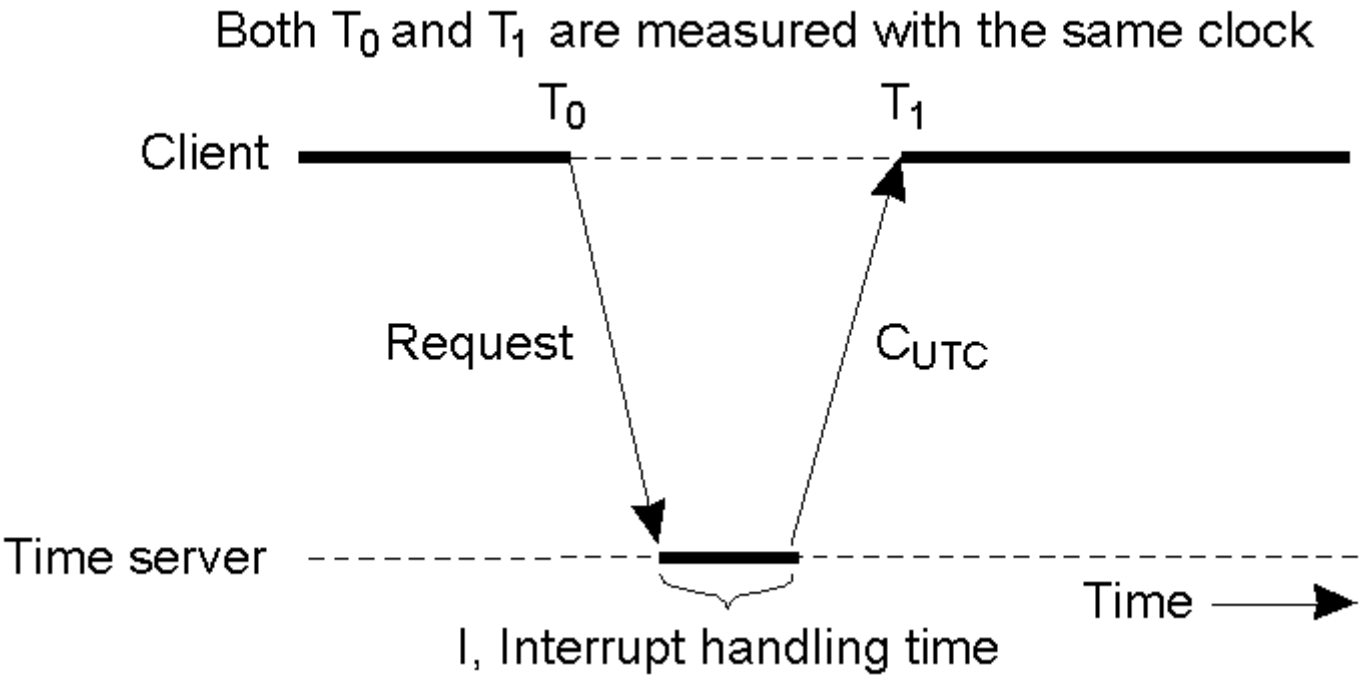
- In time Δt a clock may drift $\rho\Delta t$
- Assume two clocks are synchronized at time *t.* Assume that the clocks drift the maximum possible in Δt in opposite directions. In Δt, they are $2\rho\Delta t$ apart.
- To limit drift to δ
  - $2\rho\Delta t=\delta$
  - $\Delta t=\delta/(2\rho)$.
  - **resynchronize every $\delta/2\rho$ seconds**

Cristian's Algorithm

- One time server (WWV) receiver; all other machines stay synchronized with the time server.
- Periodically (no more than $\sigma/2\rho$ seconds), each machine sends a message to the time server asking for the current time.
- Time server machine responds with C_{UTC}

Cristian's Algorithm

Getting the current time from a time server.



Cristian's Algorithm

- C_{UTC} could be smaller then requestor's clock. Must not set the clock backwards (would this work for the *makefile* example?).
  - Suppose that a timer is set to generate 100 interrupts per second.
  - Each interrupt would add 10 msec to the time.
  - Instead the interrupt routine could add only 9 msec each time until the correction has been made.
  - A clock can be advanced gradually by adding 11 msec at each interrupt.

Cristian's Algorithm

- Problem with client setting the clock to C_{UTC}
  - It takes a nonzero amount of time for the time server's reply to get back to the sender.
  - The delay may be large and it varies with network load.
  - $T_{0:}$ Starting time of the client's request
  - T_1: be the time that the client receives an answer (measured using the same clock).
  - The best estimate of the message passing propagation time is this: $(T_1-T_0)/2$
  - The new time should be $C_{UTC} + (T_1-T_0)/2$

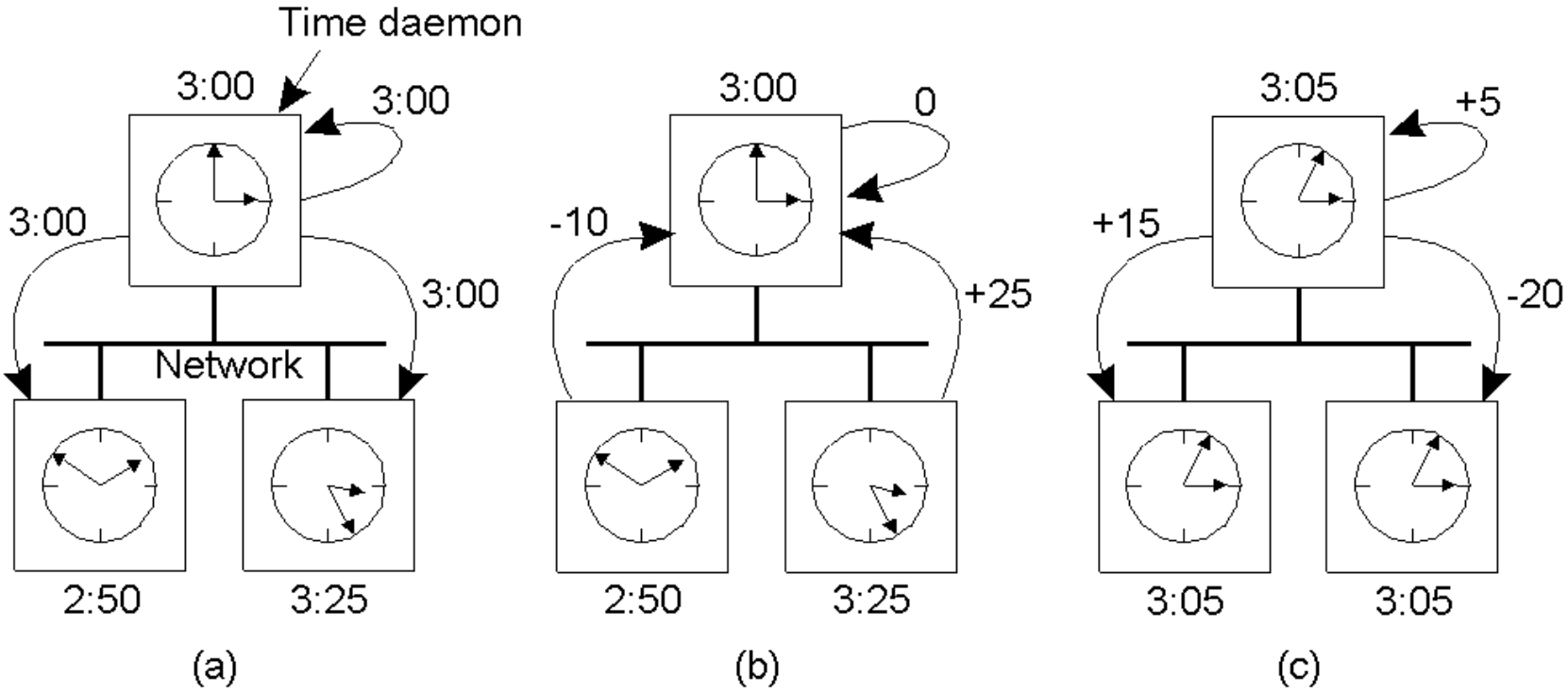
Cristian's Algorithm

- Problem with client setting the clock to C_{UTC} (continued)
  - Also should take into account time it takes the time server to handle the interrupt and process the incoming message. This is the *interrupt handle time*, *I*.
  - The new time should be based on T_1-T_0 -I
  - Christian suggested sending several messages and the fastest reply would be the most accurate since it presumably encountered the least traffic.

Berkeley Algorithm

- Suitable when no machine has a WWV receiver.
- The time server (daemon) is active:
  - Time daemon polls every machine periodically to ask what time is there
  - Based on the answers, it computes an average time
  - Tells all other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.
  - Propagation is taken into account.
- The time daemon's time is set manually by operator periodically.

Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values using a polling mechanism and providing “current time”

b) The machines answer indicating how they differ from time sent.

c) The time daemon tells everyone how to adjust their clock based on a calculation of the “average time value”.

Berkeley Algorithm

- The master takes a *fault-tolerant average*.
  - A subset of clocks is chosen that do not differ from one another by more than a specified amount and the average is taken of readings from only these clocks.
- Should the master fail, then another can be elected (discussed later) using one of the election algorithms.

Averaging Algorithms

- Every *R* seconds, each machine broadcasts its current time.
- The local machine collects all other broadcast time samples during some time interval, *S*.
- **The simple algorithm**:: the new local time is set as the average of the value received from all other machines.

Averaging Algorithms

- **A slightly more sophisticated algorithm** :: Discard the *m* highest and m lowest to reduce the effect of a set of faulty clocks.
- **Another improved algorithm** :: Correct each message by adding to the received time an estimate of the propagation time from the i^{th} source.
  - extra probe messages are needed to use this scheme.
- One of the most widely used algorithms in the Internet is the **Network Time Protocol (NTP).**
  - Achieves worldwide accuracy in the range of 1-50 msec.

Network Time Protocols (NTP)

- Cristian's method and Berkeley algorithm intended for intranets
- NTP intended to provide the ability to externally synchronize clients across the Internet to UTC.

Outline

- Clock Synchronization
- Physical Clock Synchronization Algorithms
- **Logical Clocks**
- Election Algorithms
- Mutual Exclusion
- Distributed Transactions
- Concurrency Control

Logical Clocks

- For distributed purposes, it is sufficient to know that, all machines agree on the same time order.
- **RELIABLE WAY OF ORDERING EVENTS IS REQUIRED**
- *It is not essential that the time also agrees with the real time.*
- Hence, it is internal consistency of the clocks that matters, and not whether they are close to real time.
- These are said to be ***Logical clocks***

- There is a clock Ci at each process pi
- The clock Ci can be thought of as a function that assigns a number Ci(a) to any event a, called the **timestamp** of the event a, at pi.
- These clocks can be implemented by counters and have no relation to physical time.

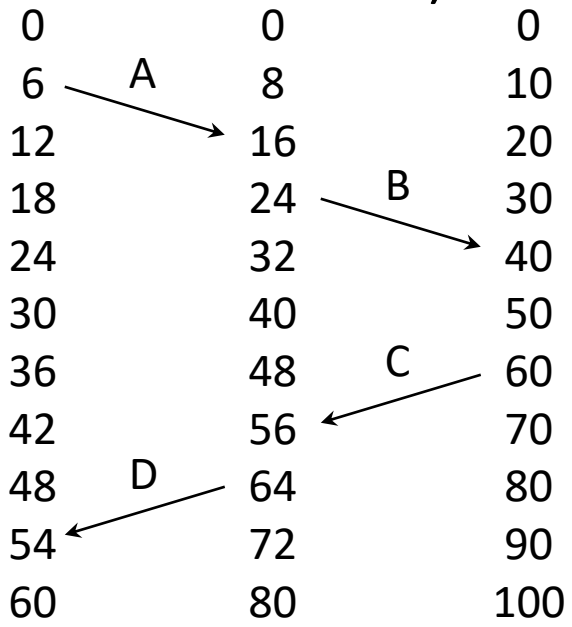
Lamport's Algorithm

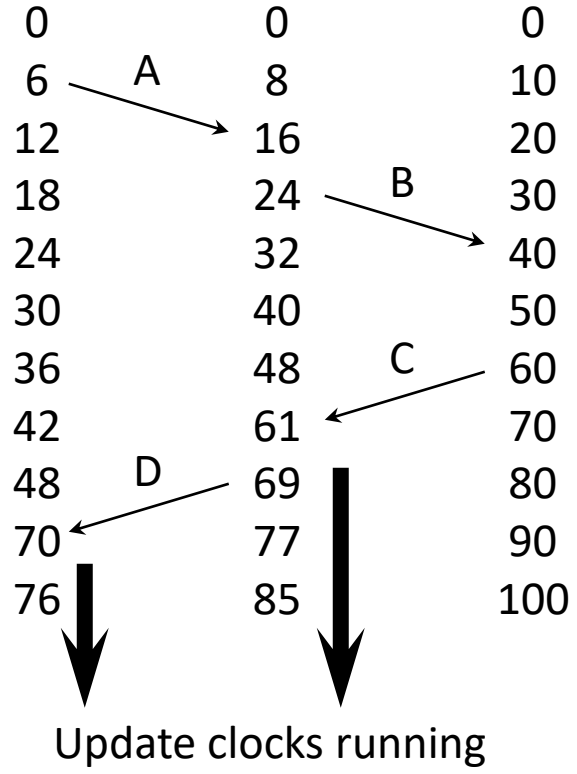
- Lamport defined a relation called happens-before
- The expression a->b is read "a happens before b" and means that all processes agree that the first event a occurs, then afterward, event b occurs.
- The happens-before relation can be observed in two situation:
  - If a and b are events in the same process, and event a occurs before event b, then a -> b is true.
  - If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a ->b is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive.

- Extra note:
  - Happens-before is a transitive relation.
  - If two events, x and y, happen in different processes, and do not exchange message (not even through 3rd party), then x->y is false and these events are said to be concurrent
  - For every event a, we assign a time value C(a) on which all processes agree.

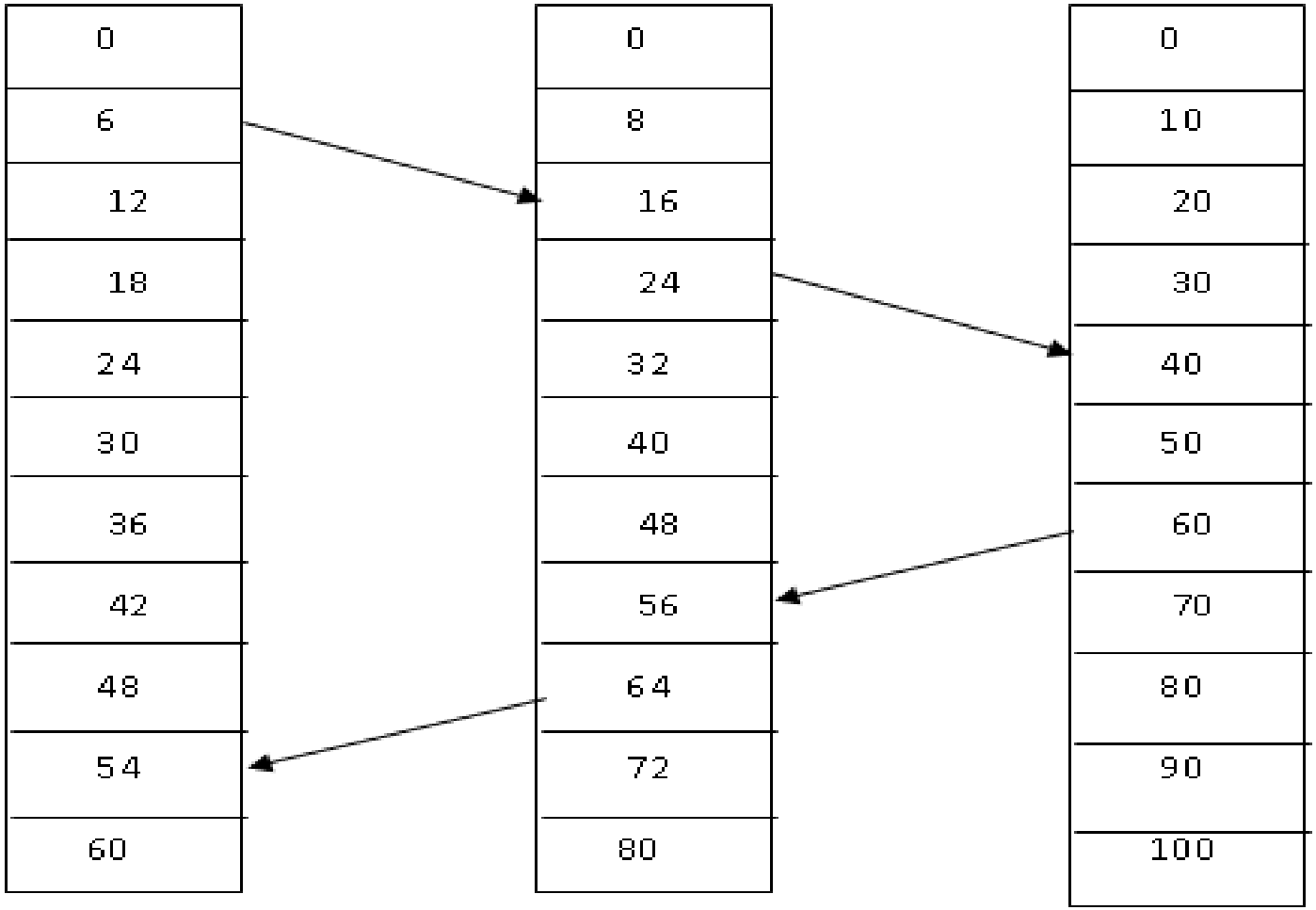
Lamport's Algorithm (continue)

- Consider the three processes in Figure :
  - Each process runs on a different machine
  - Each with its own clocks
  - Each clock is running at its own speed. (constant speed but different rates)

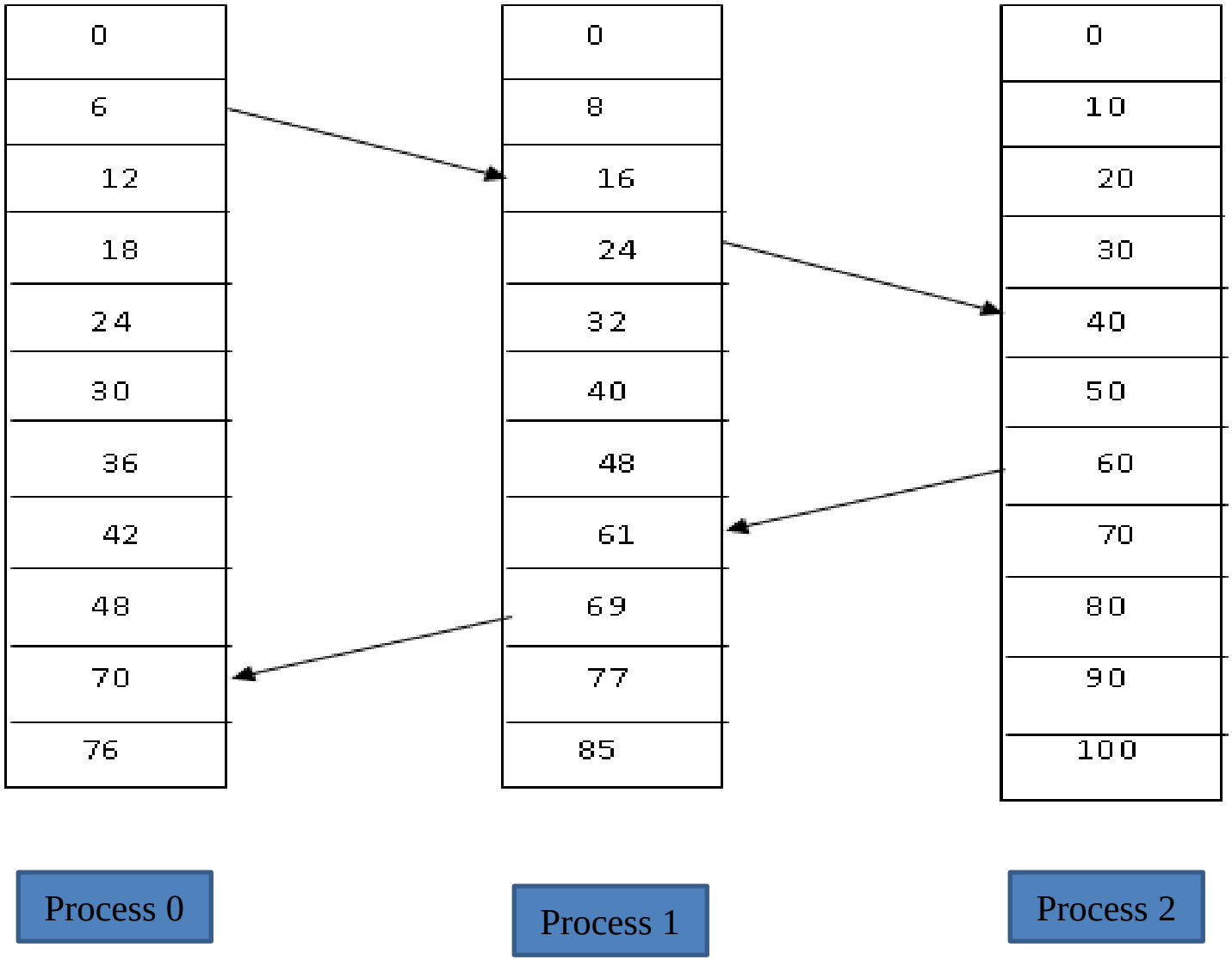




Lamport's algorithm :



Correction using Lamport's Algorithm:



Lamport clocks:

- Lamport showed that what usually matters is not that all processes agree on exactly what time it is, but rather ***that they agree on the order in which events occur.***

- *In the 'make' example, what matters is whether input.c is older to input.o or is newer.*

Lamport's algorithm

- Consider three processes running on different machines, each with its own clock, running at its own speed.
- When the clock has ticked

  6 times in p0,

  it has ticked 8 times in p1 &

  10 times in p2.
- Each clock runs at a constant rate , but the rates are different due to differences in crystals

- These values are clearly *impossible* and must be *prevented.*
- ***Lamport's algorithm*** is used to correct this situation.
- Since message C left at 60, it must arrive at 61 or later
- Each message carries the sending time according to the sender's clock.
- When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.
- Hence, C arrives at 61 and D arrives at 70

- Between any two events, the clock must tick at least once.
- When all the events in the system are completely ordered, there can be situation in which, two events take place exactly at the same time.
- Hence, both the events will have exactly the same time stamp, say 100
- But if the events have to be totally ordered, no two events can take place at exactly same time.
- Hence, one event is given time 100, and the other is given time 100.1

Vector Timestamps

The happens-before relation is observed in two situations:

1. If a & b are events in the same process, & a occurs before b, then a→b is true

2. If a is an event of a message being sent by one proces, & b is the event of message being received by another process, then, a→b is true

   As the message can not be received before it is sent.

   Happens-before relation is a ***transitive relation.*** i.e.

   If a→b and b→c then a→c is true.

Concurrent events:

- If two events x and y happen in two different processes, and they do not exchange messages with each other (Don't communicate), then neither x→y nor y→x is true.

- We cannot say whether which one event happened before the other.

- *These events are said to be concurrent. Indicated as (a||b)*

- For any two events a and b in a distributed system,

  either a→b or b→a or a||b

Using Lamport's algorithm, all the events are assigned time, subject to following conditions:

- If a happens before b in the same process, then, C(a) <C(b).
- If a and b represent the sending and receiving events of the same message, then, C(a) < C(b).
- For all events a and b, C(a) != C(b).

Questions

- Illustrate the need for clock synchronization (slides 3-5).
- Calculate the resynchronization period/On a typical DS, how will you design the resynchronization interval? (slides 14-16).
- Physical clock synchronization Algorithms. Compare
- Illustration of Lamport's algorithm with example