Advanced Operating Systems

Chapter 5: Synchronization

Reihaneh khorsand

Contents

- CLOCK SYNCHRONIZATION
- LOGICAL CLOCKS
- GLOBAL STATE
- ELECTION ALGORITHMS
- MUTUAL EXCLUSION
- DISTRIBUTED TRANSACTION

Clock Synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it.

If process A asks for the time. and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got**.** ***It will certainly not be lower.*** In a distributed system, achieving agreement on time is not trivial.

And in a distributed system were no global agreement on time

Is it possible to synchronize all the clocks in a distributed system?

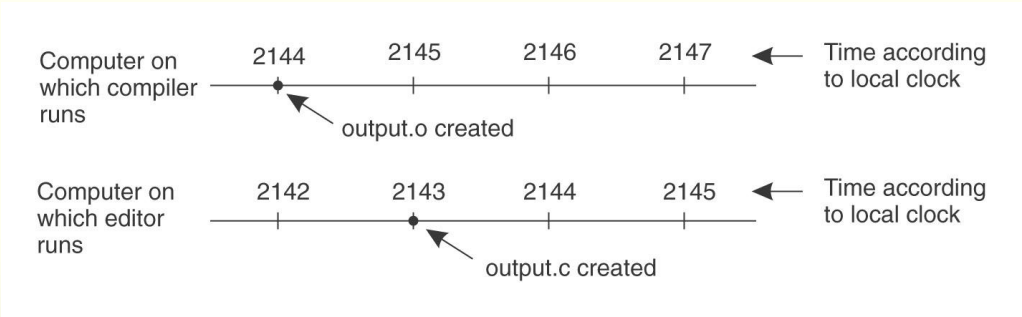


Fig1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clock Synchronization

Physical Clocks

✓A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency

✓each crystal are two registers, a counter and a holding register.

✓ Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register

✓But in the distributed system , it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency.

Clock Synchronization

Physical Clocks

✓when a system has *n* computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of synch and give different values when read out. This difference in time values is called **clock skew**

✓In some systems (e.g., real-time systems), the actual clock time is important. Under these circumstances, external physical clocks are needed

For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems:

1. **How do we synchronize them with real world clocks?**
2. **How do we synchronize the clocks with each other?**

The Berkeley Algorithm

Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there

Based on the answers, *it computes an average time* and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved

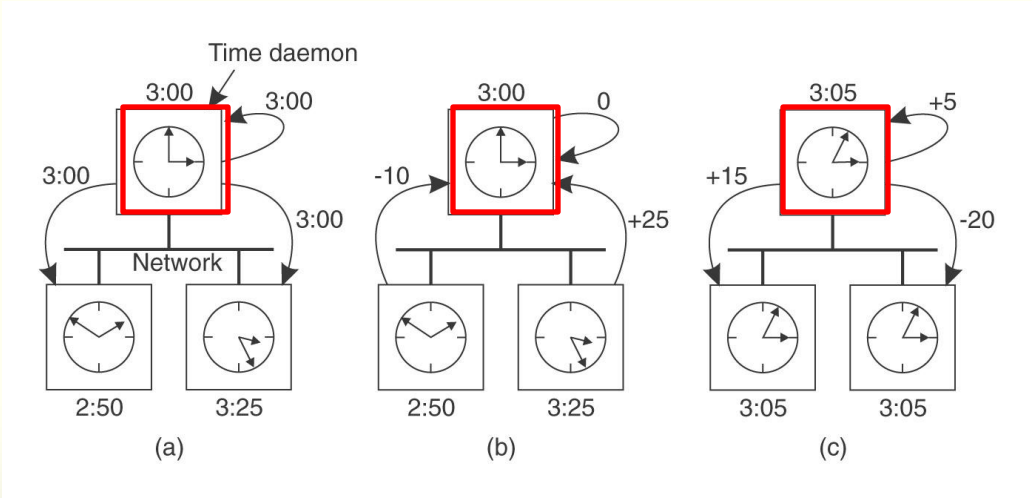


Fig2: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer(c) The time daemon tells everyone how to adjust their clock.

Logical Clocks

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with the real time as announced on the radio every hour.

for the curtain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. for these algorithms, it is conventional to speak of the clocks as **logical clocks.**

Lamport's Logical Clocks

Lamport showed **that although clock synchronization is possible, it 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 events occur

Lamport's algorithm, synchronizes logical clocks.

Lamport's timestamps

To synchronize logical clocks, Lamport defined a relation called happens-before:

The expression (a $\longrightarrow$ b) is read "a happens before b" and means that all processes agree that first event a occurs, then afterward, event b occurs, then afterward, event b occurs.

The happens-before relation can be observed directly in two situations:

1. If a and b are events in the same process, and a occurs before b, then (a $\rightarrow$ b) is true.
2. 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* $\rightarrow$ *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, nonzero amount of time to arrive.

Lamport's timestamps

Happens-before is a transitive relation, so if $(a \rightarrow b)$ *and* $(b \rightarrow c)$ *, then* $(a \rightarrow c)$.

If two events, *x and y, happen in different processes that do not exchange messages,* then $x \sim y$ *is not true, but neither is* $y \sim x$. These events are said to be **concurrent, which simply means that nothing can be** said (or need be said) about when the events happened or which event happened first.

a If *a and b are events in the same process, and a occurs before b:*

$(a \rightarrow b)$ *then* $C(a)<C(b)$

b Similarly, 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 :

$(a \rightarrow b)$ *then* $C(a)<C(b)$

In addition, the clock time, C, must always go forward (increasing), never backward (decreasing).

Lamport's algorithm

Lamport's solution follows directly from the happens-before relation.

Example:

Consider the three processes depicted in Figure
The processes run on different machines, each with its own clock, running at its own speed.

As can be seen from the figure, when the clock has ticked 6 times in process PI, it has ticked 8 times in process Pz and 10 times in process P3' Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

Lamport's solution follows directly from the happens-before relation.

Lamport's algorithm

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.

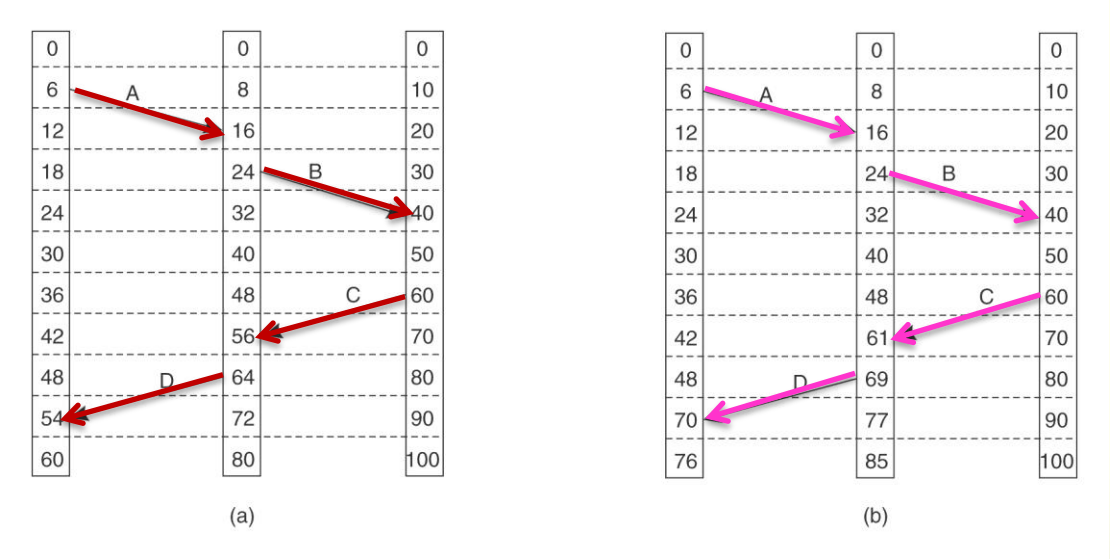


Fig3: The processes run on different machines, each with its own clock, running at its own speed.

Lamport's algorithm

To implement Lamport's logical clocks, each process *Pi maintains a local counter* Ci. These counters are updated as follows steps :

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), *Pi executes Ci← Ci+ 1.*

2. When process Pi sends a message m to Pj it sets ts (m) equal to Ci after having executed the previous step.

3. Upon the receipt of a message m, process Pj adjusts its own local counter as Cj ← max{Cj, ts (m) }, after which it then executes the first step and delivers the message to the application.

Lamport's algorithm

then **using Lamport's timestamps**, we now have a way to assign time to all events in a distributed system subject to the following conditions:

1. If a happens before b in the same process ,C(a)<C(b).
2. If a and b represent the sending and receiving of message, respectively, C(a)<C(b).
3. For all distinctive events a and b , C(a)≠ C(b).

Election Algorithms

Many distributed algorithms require **one process to act as coordinator, initiator**, or otherwise **perform some special role**.

It does not matter **which process takes on this special responsibility**, but one of them has to do it.

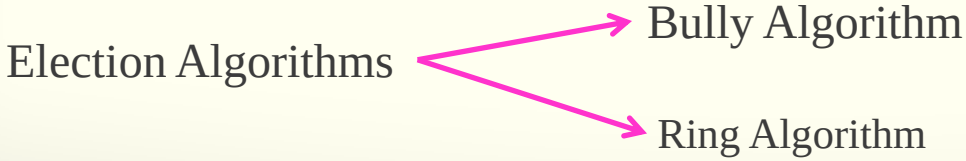
If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special.

Election Algorithms

- we will assume that each process has a unique number, for example, its network address.

- In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator.

- **The algorithms differ in the way they do the location**.



The Bully Algorithm

When any process notices that the coordinator is no longer responding to requests, it initiates an election.

A process, *P, holds an election as follows:*

1. **P sends an ELECTION message to all processes with higher numbers.**
2. **If no one responds, P wins the election and becomes coordinator.**
3. **If one of the higher-ups answers, it takes over. P's job is done.**

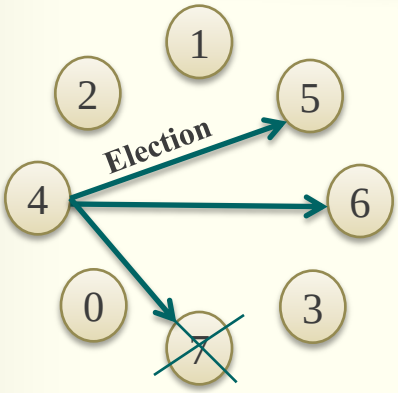
The Bully Algorithm

At any moment, a process can **get an ELECTION message** from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an **OK message** back to the sender to indicate that he is alive and will take over.

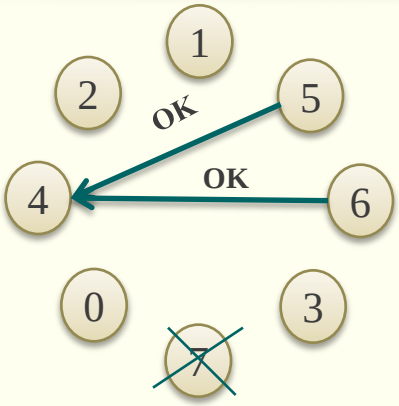
Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, **it holds an election**. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. **Thus the biggest guy in town always wins, hence the name "bully algorithm."**

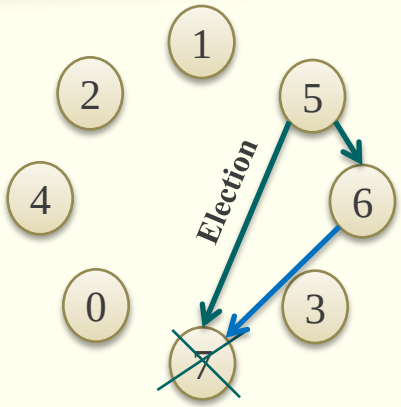
The Bully Algorithm



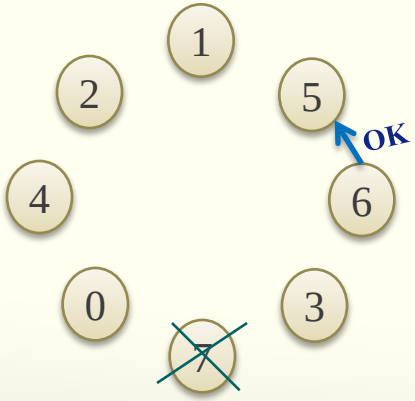
(a)



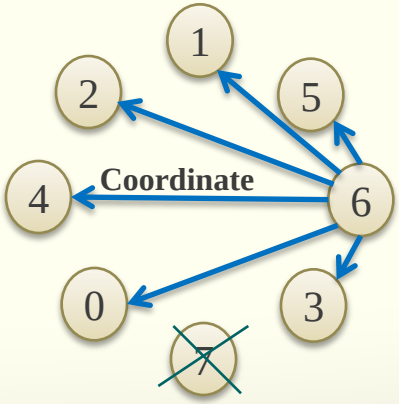
(b) Previous coordinator has crashed



(c)



(d)



(e)

A Ring Algorithm

✓This algorithm is based on the use of a ring.

✓We assume that the processes are physically or logically ordered, so that each process knows who its successor is.

✓When any process notices that the coordinator is not functioning, it builds an *ELECTION* message containing **its own process number** and **sends the message to' its successor.**

✓If the successor is down, the sender skips over the successor and goes to the next member along the ring. or the one after that, until a running process is located.

✓At each step along the way, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

A Ring Algorithm

Eventually, the message gets back to the process that started it all.

That process recognizes this event when it receives an incoming message containing its own process number.

At that point, the message type is changed to *COORDINATOR* and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are.

When this message has circulated once, it is removed and everyone goes back to work.

A Ring Algorithm

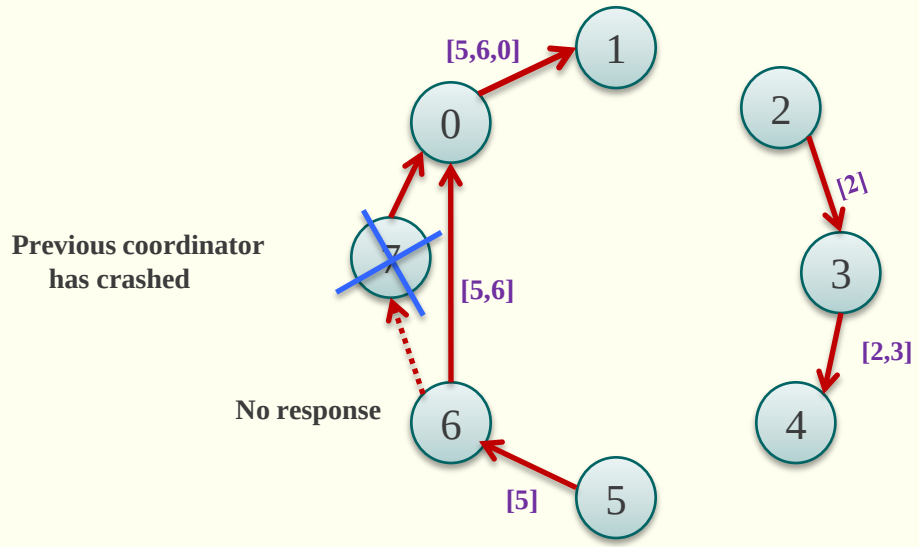


Fig3: Election algorithm using a ring.

MUTUAL EXCLUSION

Centralized Algorithm

Distributed Algorithm

Token Ring Algorithm

A Centralized Algorithm

One process is elected as the **coordinator**

Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.

If no other process is currently accessing that resource, the coordinator sends back a reply granting permission When the reply arrives, the requesting process can go ahead.

A Centralized Algorithm

Now suppose that another process asks for permission to access the resource.

The coordinator knows that a different process is already at the resource, so it cannot grant permission.

The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply.

Alternatively, it could send a reply saying "**permission denied**."

A Centralized Algorithm

When process 1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access

The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.

If the process was still blocked , it unblocks and accesses the resource.

A Centralized Algorithm

In the OSI model, communication is divided up into seven levels or layers

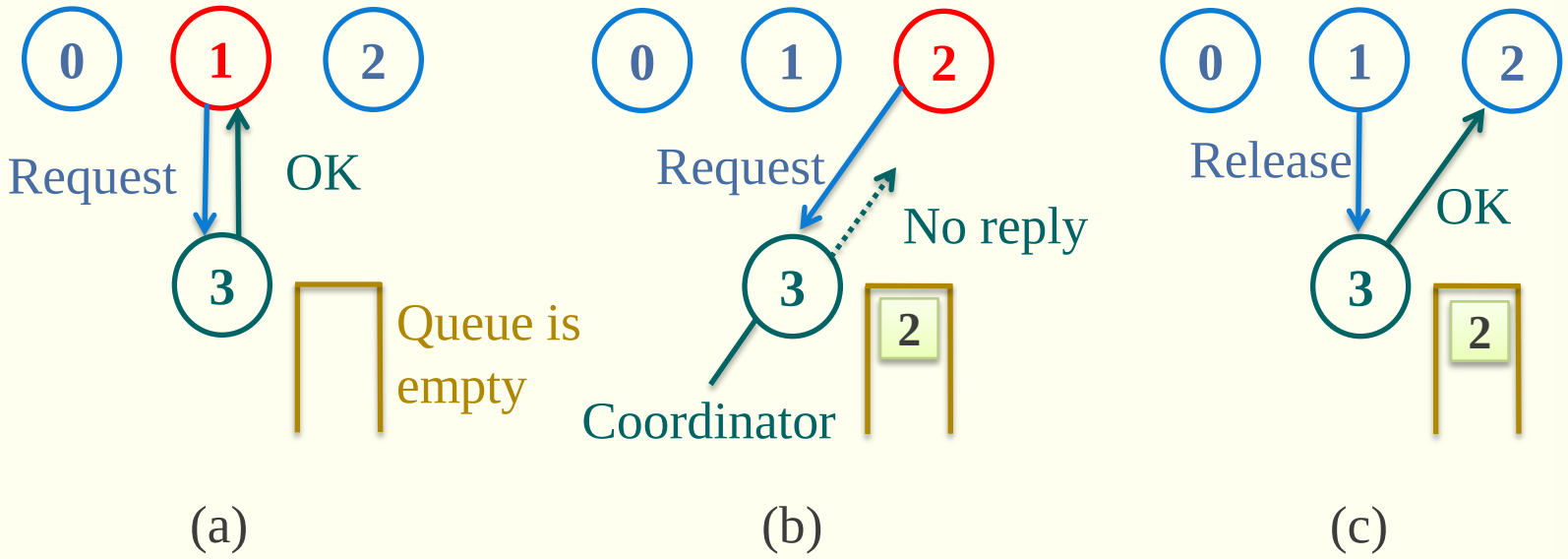


Fig 13: A Centralized Algorithm

A Distributed algorithm

When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time.

When a process receives a request message from another process

Distributed algorithm

1. If the receiver is not accessing the resource and does not want to access it, it sends back an ***OK*** message to the sender.

2. If the receiver already has access to the resource, it simply does not reply. Instead, it **queues** the request.

3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with me. one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

Distributed algorithm

After sending out requests asking permission, a process sits back and waits until everyone else has given permission.

As soon as all the permissions are in, it may go ahead.

When it is finished, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

This algorithm is **slower**, **more complicated**, **more expensive**, and **less robust** that the original centralized one.

Distributed algorithm

- Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12.
- Process 1 is not interested in the resource, so it sends *OK* to both senders.
- Processes 0 and 2 both see the conflict and compare timestamps.
- Process 2 sees that it has lost, so it grants permission to 0 by sending *OK.*
- Process 0 now queues the request from 2 for later processing and access the resource, as shown in the Figure.
- When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to go ahead,

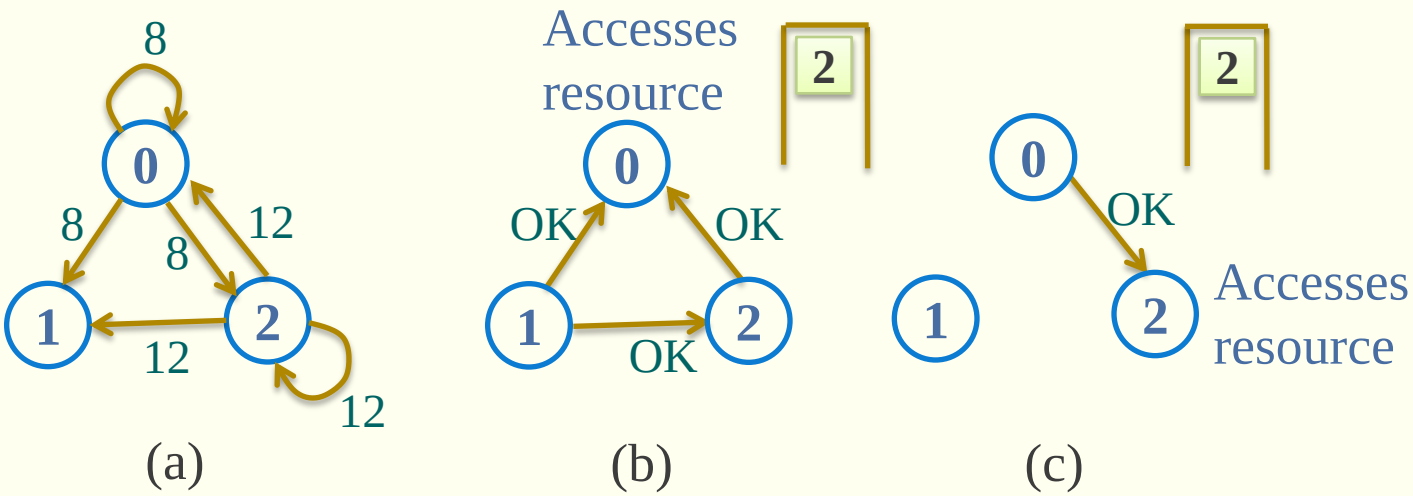


Fig 14: Distributed Algorithm

A Token Ring Algorithm

In software, a logical ring is constructed in which each process is assigned a **position in the ring**, as shown The ring positions may be allocated in numerical order of network addresses or some other means.

When the ring is initialized, process 0 is **given a token**.

The token circulates around the ring.

A Token Ring Algorithm

When a process acquires the token from its neighbor, it **checks** to see if it needs to access **the shared resource**.

If so, the process goes ahead, does all the work it needs to, and releases the resources.

After it has finished, it passes the token along the ring.

It is **not permitted** to immediately enter the resource again using the same token.

As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.

A Token Ring Algorithm

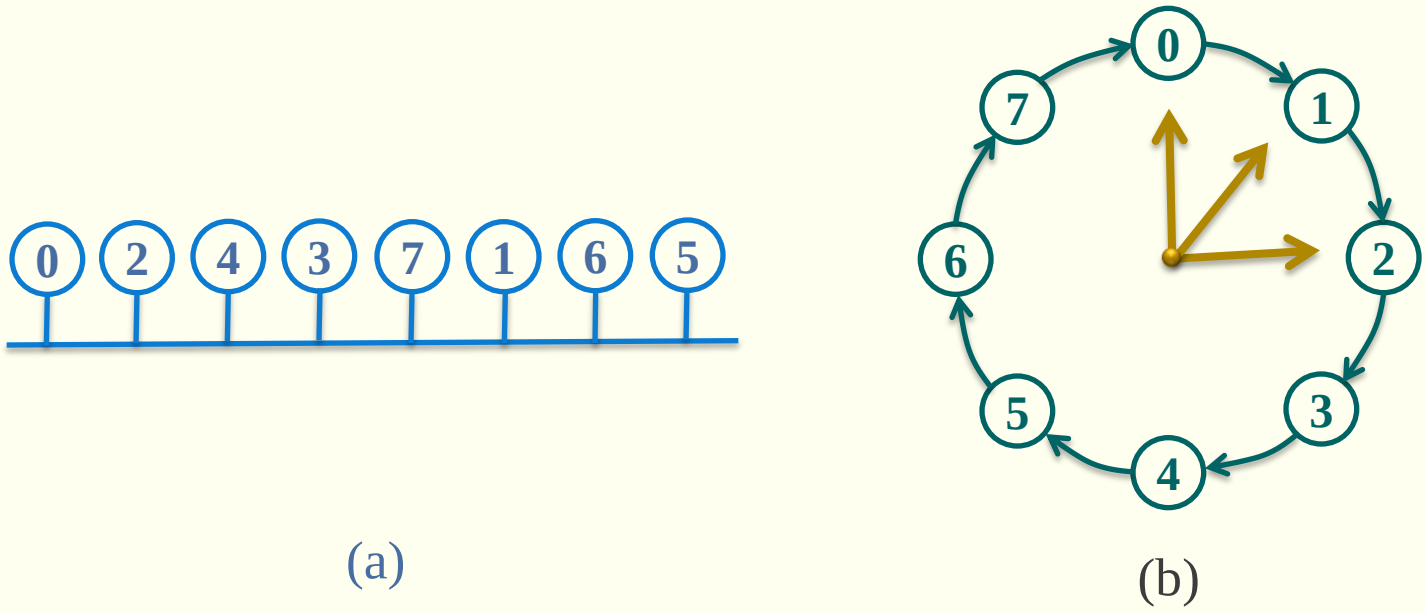


Fig 15: A Token Ring Algorithm

Distributed transactions

- The Transaction model
- Classification of transactions

The Transaction model

- One process announces that it wants to begin a translation with one or more other processes.
- They can negotiate various options, **create** and **delete** entities, and **perform** operations for a while.
- Then the **initiator** announces that it wants all the other to commit themselves to the work done so far .
- If all of them agree, the results are made **permanent**.
- If one or more processes refuse (or crash before agreement), the situation reverts to exactly the state it was in **before the transaction** begin.

The Transaction model

Low level communicative objects in transactions

Primitive	description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore to old value
READ	Read data from a file , a table , or otherwise
WRITE	Write data to a file , a table , or otherwise

Features of transactions

- Atomic: To the outside world, the transaction happens indivisibly.
- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transactions do not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

Atomic : each transaction either happens completely, or not at all,

Classification of transaction

- **Flat transaction**
- **Nested transaction**
- **Distributed transaction**

The strength of the atomicity property of the flat transaction also is partly its weakness .

Nested transaction

A Nested transaction is constructed from a number of sub transactions.

The top-level transaction may fork off children that run in parallel with one another , on deferent machines **, to gain performance** or **simplify programming** .

If an enclosing (higher-Level)transaction aborts , all its underlying **sub transactions** have to be aborted as well.

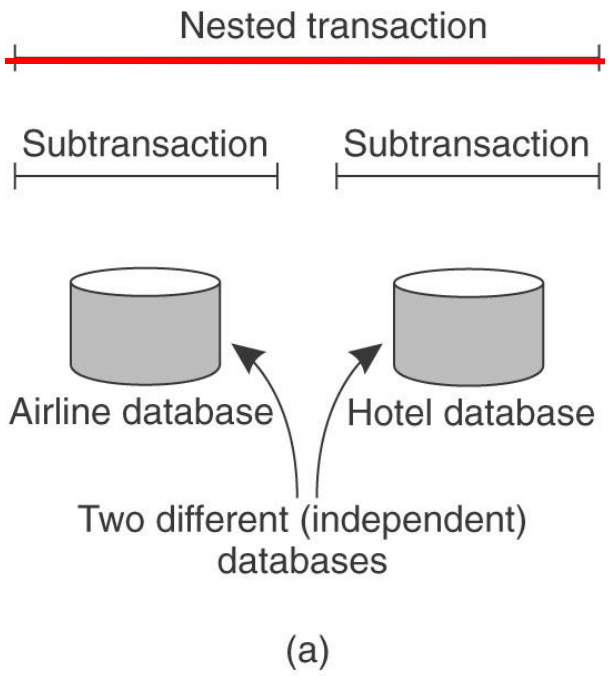
Distributed transaction

A **nested transaction** is a transaction that is logically decomposed into a hierarchy of **sub transactions**.

A **distributed transaction** is logically a flat , **indivisible transaction** that operates on **distributed data**.

The main problem with Distributed transactions is that separate Distributed algorithms are needed to handle the **locking of data** and **committing the entire transaction**.

Distributed transaction



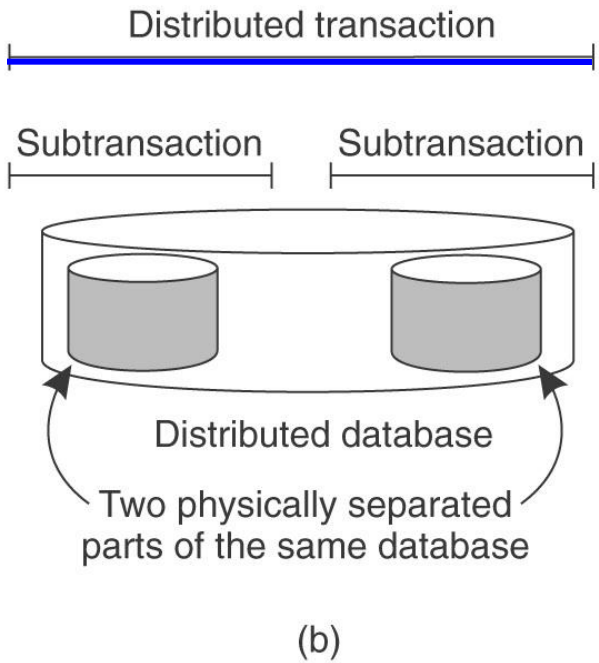


Fig 20: Distributed transaction

Concurrency control

The goal of concurrency control id to allow several transactions to be executed simultaneously , but in such a way that collection of data items that is begin manipulated , is left in a consistent state.

Concurrency control is best understood in terms of three different managers Which are organized in a layered fashion.

Concurrency control

1. The bottom later consists of a data manager that performs the actual read and write operations on data.
2. The middle layer consists of a scheduler and carries tha main responsibility for properly controlling. It determines which transaction is allowed to pass a read or write operation to the data manager and at which time.
3. The highest layer contains the transaction manager , which is primarily responsible for guaranteeing atomicity of transactions.

Concurrency control

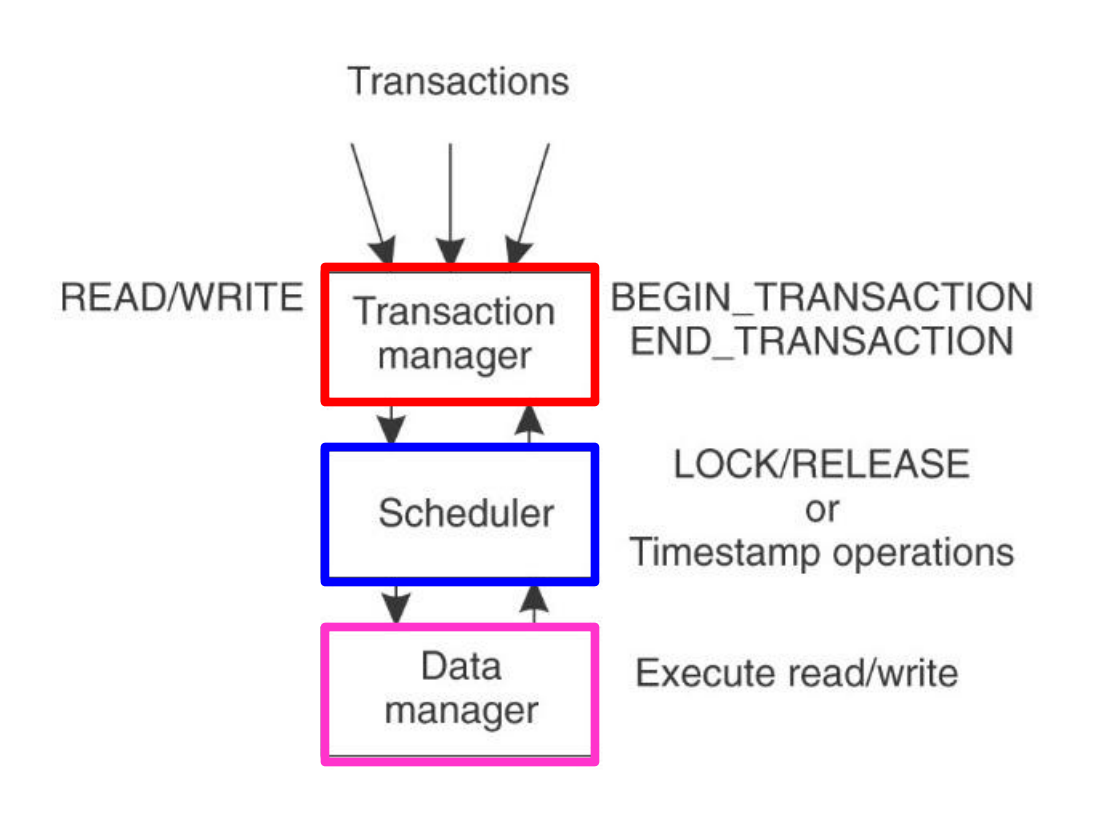


Fig 23: Concurrency control

Two-phase locking

The oldest and most widely used Concurrency control algorithm is **locking**.

When a process needs to read or write a data item as part of a transaction , it requests the scheduler to grand it a lock for that data item.

Scheduler needs to apply Two-Phase locking algorithm.

In TWO-Phase locking (2PL) , The scheduler first acquires all the locks it needs during the growing phase, and the releases them during the shrinking phase.

Two-phase locking

The following three rules

1. When the scheduler receives an operation oper(T.x) from the transaction manager , it tests whether that operation conflicts with any other operation for which it already grandy a lock. If there is a conflict , operation oper(T.x) is delayed (and thus also transaction T). if there is no conflict , the scheduler grants a lock for data item x, and passes the operation to the data manager.
2. The scheduler will never release a lock for data item x , until the data manager acknowledges it has performed the operation for which the lock was set.
3. Once the scheduler has released a lock on behalt of a transaction T,it will never grant another lock on behalf of T, no mather for which data item T is requesting a lock . Any attempt by T to acquire another lock is a programming error that abort T.

Two-phase locking

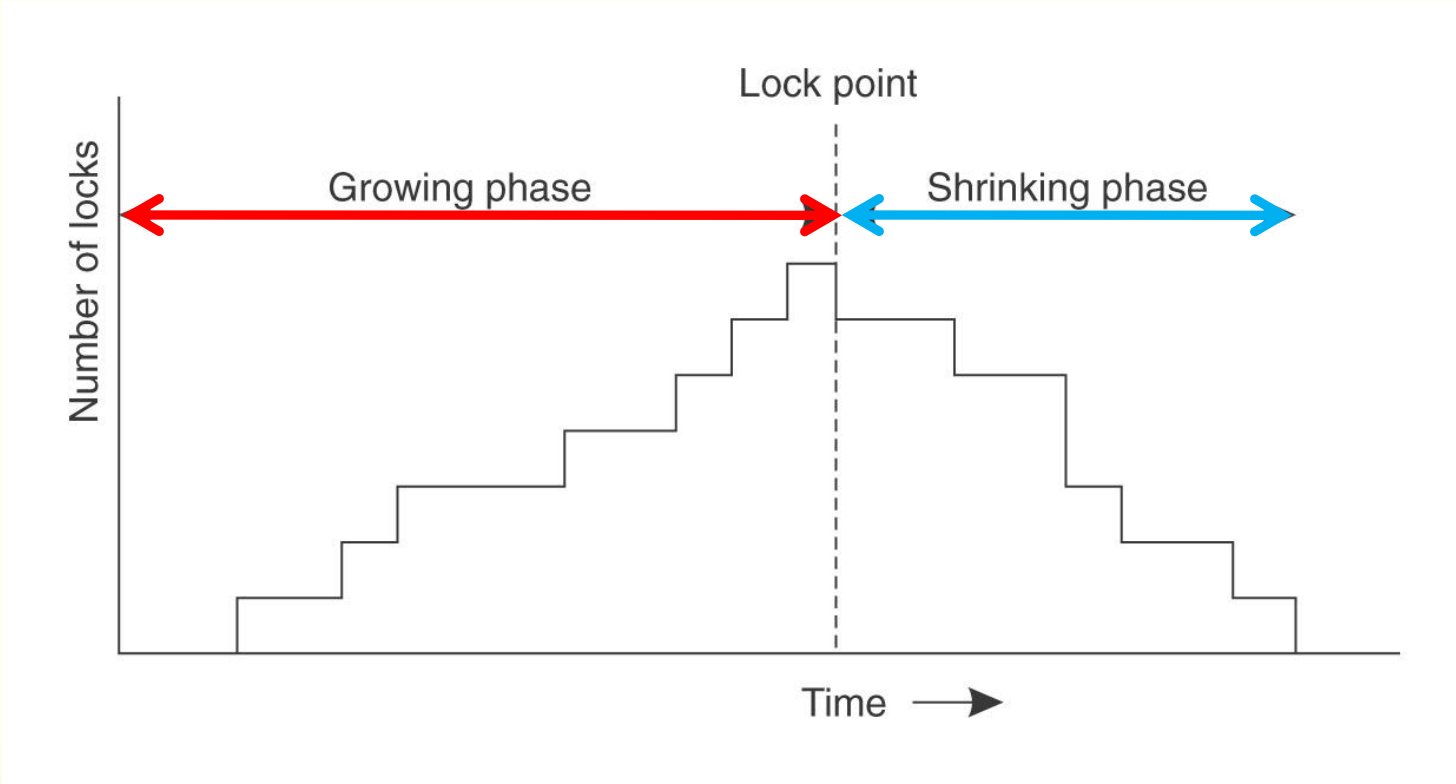


Fig 26: Two-phase locking

Two-phase locking

The shrinking phase does not take place until the transaction has finished running and has either committed or aborted , leading to the release of locks as shown.

This policy , called **strict two-phase locking** , has two main advantages.

First , a transaction always read a value written by a committed transaction ; therefore, one never has to abort a transaction

Second, all lock acquisitions and releases can be handled by the system without the transaction begin aware of them

lock are acquired whenever a data item is to be accessed are released when the transaction has finished.

Strict two-phase locking

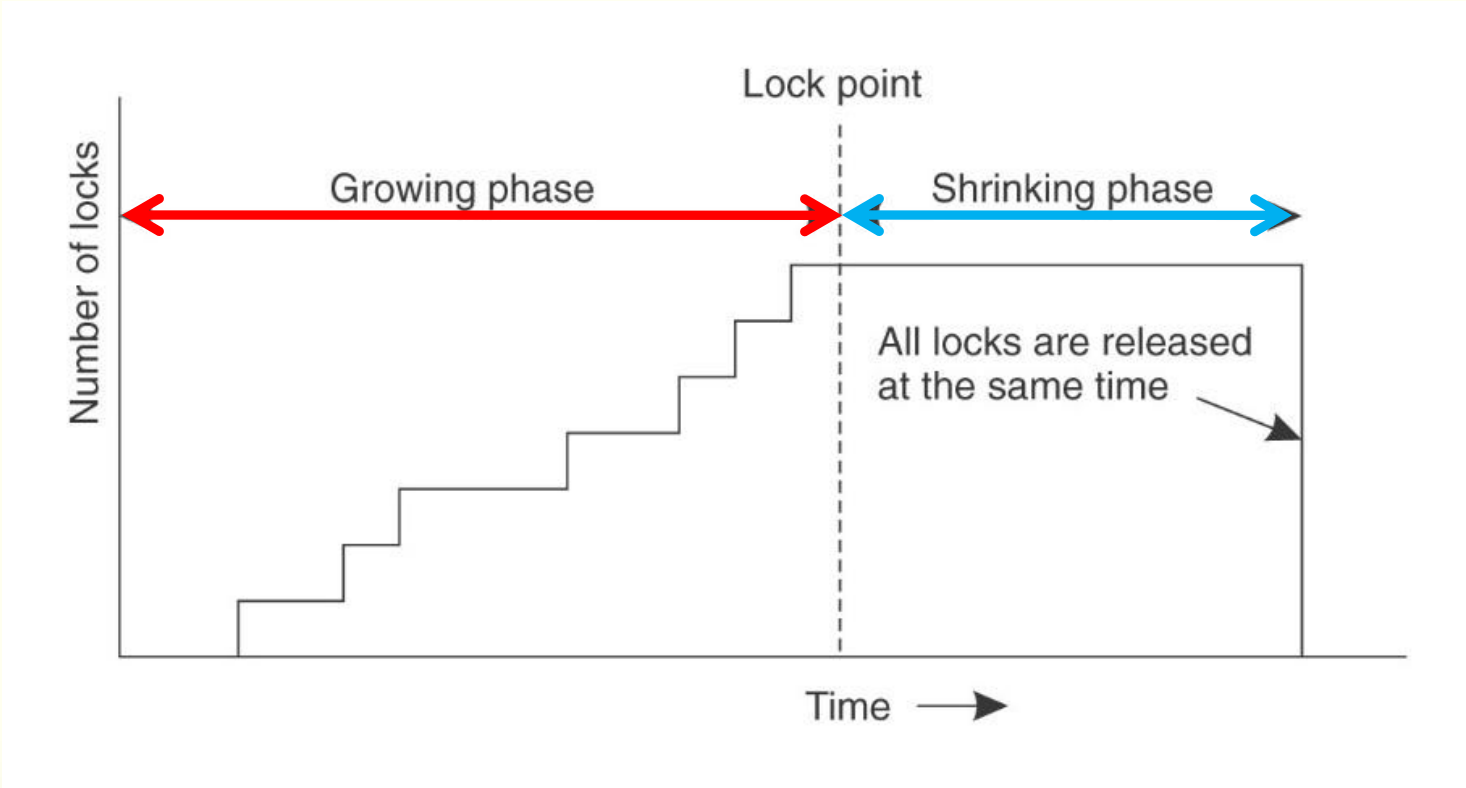


Fig 27: Two-phase locking

Two-phase locking

Both two-phase locking and strict two-phase locking can lead to **deadlocks**.

The usual techniques apply here , such as acquiring all locks in some canonical order to prevent **hold-and-wait** cycles.

Also possible is deadlock detection by maintaining an explicit graph of which process has which locks and wants which locks.

when it is known in advance that a lock will never be held longer than t sec, a timeout scheme van be used: if a lock remains continuously under the same ownership for longer than t sec , there must be a deadlock.

Two-phase locking

Basic two-phase locking scheme can be implemented in a distributed system.

Transaction manager communicates with this centralized lock manager, from which it receives lock grants.

When a lock has been granted, the transaction manager subsequently communicates directly with the data managers .

Finally , in **distributed 2PL** , it is assumed that **data** may be replicated across multiple machines.

End of Chapter 5