MONASH University
Information Technology

FIT3143

LECTURE WEEK 6

ELECTION ALGORITHMS, DISTRIBUTED TRANSACTIONS, CONCURRENCY CONTROL

Overview

- Election Algorithms
- Distributed Transactions
- Concurrency Control in Distributed System

Learning outcome(s) related to this topic

- Compare and contrast different communication and concurrency schemes (LO2)

Breakdown of Election Algorithms, Distributed Transactions and Concurrency Control

- **Election Algorithms**
  - Bully Algorithm
  - Ring Algorithm

- **Distributed Transactions**
  - System Model
  - Properties of transaction
  - Implementation of transaction
    - Private workspace
    - Write-ahead Log
  - Distributed Commit Protocols
    - 2 Phase commit
    - 3 Phase Commit

- **Concurrency Control in Distributed Systems**
  - Mechanisms for implementing Concurrency control
    - Lock based CC
    - Timestamp based CC
    - Optimistic CC

Election Algorithm

- Distributed algorithms often elect one process to acts as a coordinator or a server.
- Election Assumptions
  - We assume each process is assigned a unique process number
  - A process knows all process numbers
  - A process does not know which process number is alive.
- An Election:
  - Select the maximum process number among active processes
  - Appoint the process the coordinator.
  - All the processes must agree upon this decision.

The Bully Algorithm (by Garcia-Molina)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election:
  - P sends an ***Election message*** to all processes with higher numbers.
  - If one of the higher numbered answers, that process becomes the new leader.
  - If no one responds, P wins the election and becomes coordinator.
    - Timeout overhead
    - Delayed response

The Bully Algorithm

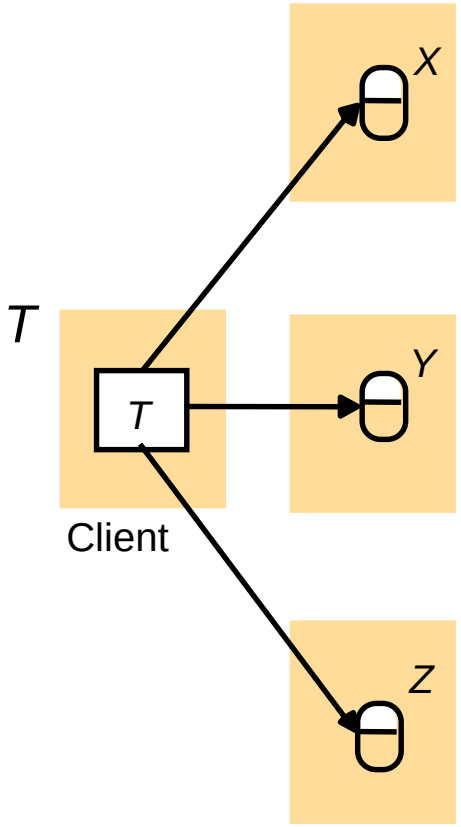
- At any moment, a process can get an ***Election message*** from one of its lower-numbered colleagues. The receiver sends an ***OK message*** back to the sender.
- The receiver then holds an election, unless it is already holding one.
- Eventually, all processes give up but one, and that one is the new coordinator.
- It announces its victory by sending ***Coordinator messages*** to all processes.
- 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.
- The highest numbered process always wins, hence the name "bully algorithm."

A Ring Algorithm

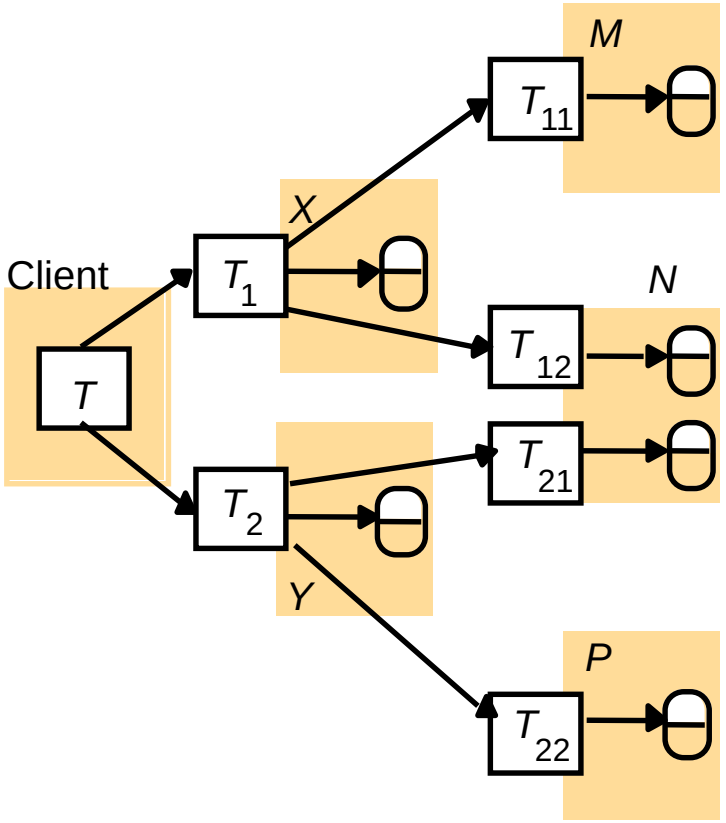
- Processes form a logical ring. Each process knows which process comes after it.
- If a process finds the coordinator dead, it sends an ***Election message*** which contains its own process number to its successor in the ring. If the successor is down, the message is sent to the next process along the ring.
- Each process appends its process number to the message upon receiving an ***Election message*** and forward it to the successor.
- When the message returns to the process that originally sent the message, the process changes the message type from ***Election message*** to ***Coordinator message*** and circulates the message again.
- The purpose of the ***Coordinator message*** is to inform all processes that the highest-numbered process is the new coordinator and the processes contained in the message are the members of the ring.

Distributed Transactions

Distributed Transactions



Simple Distributed Transaction



Nested Distributed Transaction

Distributed (Atomic) Transactions

- An atomic transaction gives a view that either a set of operations is all completed or none of them is completed.
- First a process declares to all other processes (involved in a transaction) that it is starting the transaction.
- The processes exchange information and perform their specific operations.
- The initiator announces that it wants all the other processes to “commit” to all the operations done up to that point.
- If all processes agree to commit, the results are made permanent.
- If at least one of the processes refuse, all the states - usually opened files/databases (on all machines) - are reversed to the point before starting the transaction.

Transaction Processing

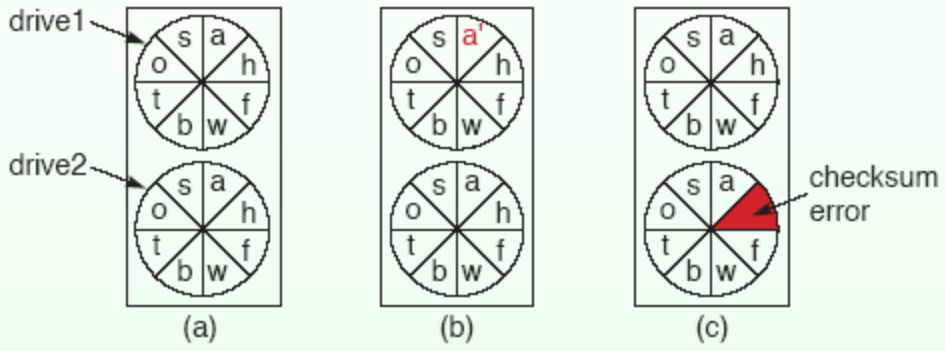
System Model for a Transaction

- A system consists of independent processes where each process may fail at random.
- No communication error may be encountered.
- A stable storage is available.
  - "Stable storage" guarantees no data is lost unless the storage is physically lost by disasters such as floods or earthquakes.

    Note: Data in RAM disappear when power is turned off. Data in a disk becomes inaccessible if the head crashes. Thus they are not stable storages

Stable Storage

- Stable storage can be implemented with a pair of disks.



- Each block on drive 2 is a copy of the corresponding block on drive 1.
- When a block is updated, the block on drive 1 is first updated and verified. Then, the same block on drive 2 is updated and verified.
- If the system crashes after the update of drive 1 but before the update of drive 2, the blocks on the two drives are compared and the blocks on drive 1 are copied to drive 2 after system reboot.
- If checksum error is encountered because of a bad block, the block is copied from the disk without an error.

Transaction Primitives

- Programming a transaction requires special primitives that must be supplied by the OS or by the programming language.

- Examples of the primitives are:
  - ***Begin Transaction***: Specifies the beginning of a transaction.
  - ***End Transaction***: Specifies the end of a transaction, and tries to commit on the results.
  - ***Abort Transaction***: Abort the transaction and restore the old value.
  - ***Read***: Reads data from a file.
  - ***Write***: Writes data to a file.

- ***Begin Transaction*** and ***End Transaction*** specify the scope of a transaction. The operations delimited with the two operations form the transaction.

- These primitives are implemented as either system calls, library functions, or constructs of a programming language.

Properties of Transactions

- Four essential properties:
  - Atomic
  - Consistent
  - Isolated
  - Durable

Atomicity

- A transaction happens indivisibly to the outside world. (Either all operations of the transaction are completed or none is completed.)
- The other process cannot observe the intermediate states of the transaction.

Consistency

- A transaction does not violate system invariants.
  - For example, an invariant in a banking system is the conservation law regarding the total amount of money.

- The sum of money in the source and destination accounts after any transfer is the same as the sum before the transfer (although the conservation law may be violated during the transfer).

Isolation

- Even if more than one transactions are running simultaneously, the final result is the same as the result of running the transactions sequentially in some order.

- This property is also called "***Serializable***."

Durability

- Once a transaction commits, changes are permanent.
- No failure ***after the commit*** can undo the results or cause these to be lost.

Nested Transactions

- Transactions may contain sub-transactions. This is called nested transactions.

- Problem with nested transactions
  - Assume a transaction starts several sub-transactions in parallel.
  - One of these commits and makes its results visible to the parent transaction.
  - For some reason parent is aborted, the entire system needs to be restored to its original state.
  - Consequently the results of the sub-transaction that committed must be undone.
    - Undo'ing a committed transaction is a violation.

Nested Transaction

Characteristics

- Permanence of a sub-transaction is only valid in the world of its direct parent and is invalid in the world of further ancestors.
- Permanence of the results is valid only for the outermost transaction.

Addressing Transaction Violation Problem

- One way to implement nested transactions is to create a private copy of all objects in the system for each sub-transaction.
- If a sub-transaction commits, its private copy replaces its parent's copy.

Implementation of Atomic Transactions

- Implementing atomic transactions means
  - Hiding of operations within a transaction from outer processes.
  - Making the results visible only after commit.
  - Restoring the old state if the transaction is aborted.

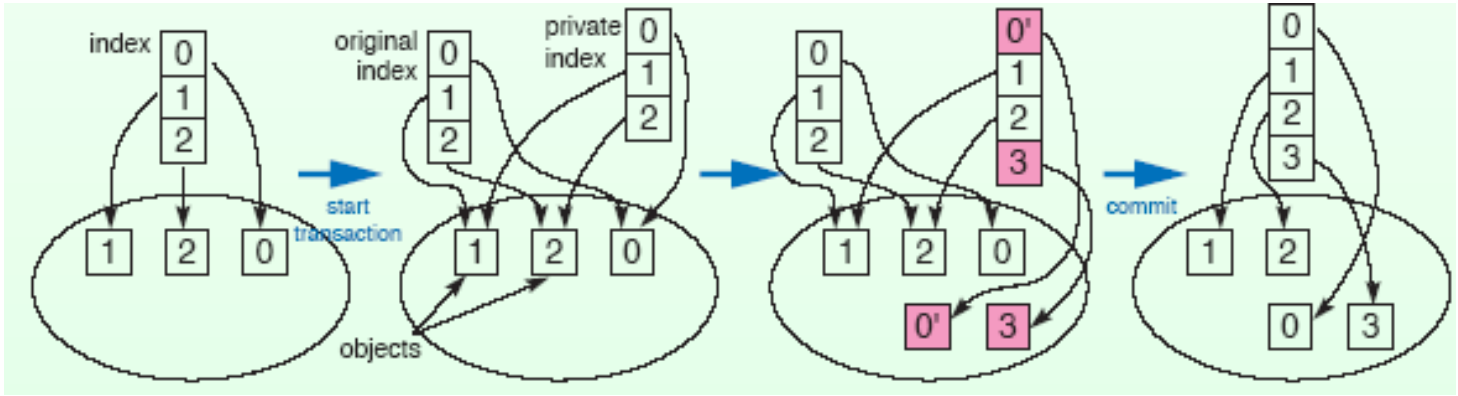
- Two ways to restore the old state:
  - Private workspace
  - Write-ahead log

Private Workspace

- When a process starts a transaction, it is given a private workspace containing all the objects including data and files.
- Read and write operations are performed within the private workspace.
- The data within the private workspace is written back when the transaction commits.
- The problem with this technique is that the cost of copying every object to a private workspace is high.
- The following optimisation is possible:
  - If a process only reads an object, there is no need for a private copy.
  - If a process updates an object, a copy of the object is made in the private workspace.

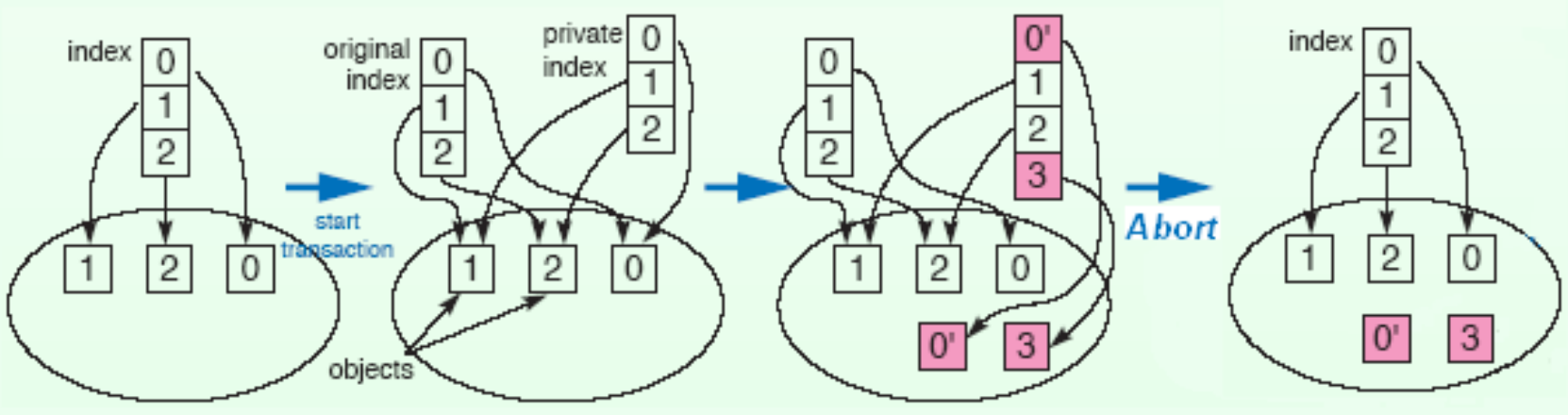
Private Workspace

- If we use indices to objects, we can reduce the number of copy operations.
- When an object is to be updated,
  - A copy of the index and the objects is made in the private workspace.
  - The private index is updated.
  - On commit, the private index is copied to the parent's index.



Private Workspace

- If the transaction aborts, the private copies of objects and the private indices are discarded.

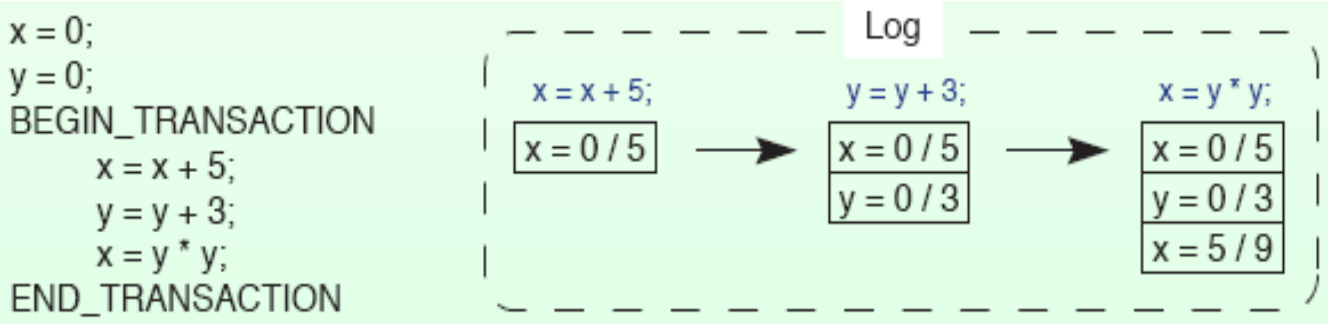


Write-ahead Log

- **Before any changes to objects are made, a record (log) is written to a "write-ahead log" on stable storage.**

- **The log contains the following information**
  - **which transaction is making the change,**
  - **which object is being changed, and**
  - **what the old and new values are.**

- **Only after the log has been written successfully, the change is made to the object.**



Write-ahead Log

- If the transaction committed, a commit record is written to the log (data has already been written).
- If the transaction aborts, the log can be used to rollback to the original state.
- The log can be used for recovering from crashes.

Distributed Commit

Two-Phase Commit Protocol

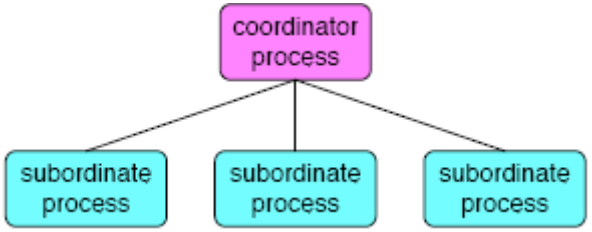
- The action of committing a transaction must be done ***instantaneously*** and ***indivisibly***.

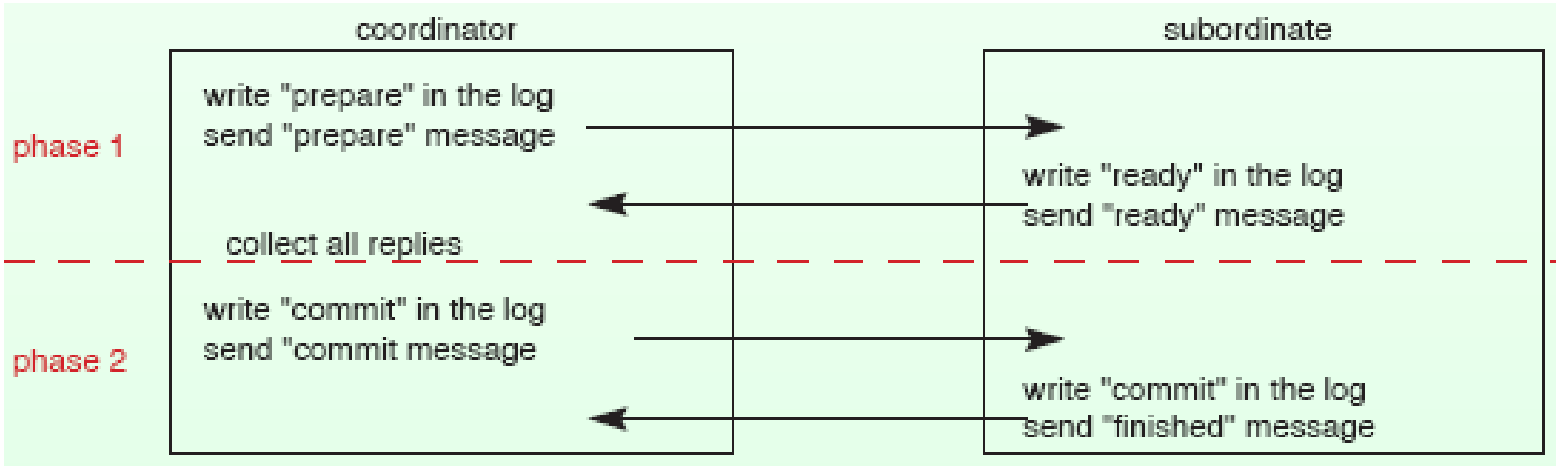
- In a distributed system, the commit requires the ***cooperation*** of multiple processes (may be) on different machines.

- Two-phase commit protocol is the most ***widely used protocol*** to implement atomic commit.

Two Phase Commit Protocol

- One process functions as the coordinator.
- The other participating processes are subordinates (Cohorts).





Two Phase Commit Protocol

- The coordinator writes an entry “prepare” in the log on a stable storage and sends the other processes involved (the subordinates) a message telling them to prepare.
- When a subordinate receives the message, it checks to see if it is ready to commit, makes a log entry, and sends back its decision.
- After collecting all the responses, if all the processes are ready to commit, the transaction is committed. Otherwise, the transaction is aborted.
- The coordinator writes a log entry and then sends a message to each subordinate informing it of the decision. The coordinator’s writing commit log actually means committing the transaction and no rolling back occurs no matter what happens afterward.

Two Phase Commit Protocol

- This protocol is highly resilient in the face of crashes because it uses stable storages.

- If the coordinator crashes after writing a “prepare” or “commit” log entry, it can still continue from sending a “prepare” or “commit” message.

- If a subordinate crashes after writing a “ready” or “commit” log entry, it can still continue from sending a “ready” or “finished” message upon restart.

Three Phase Commit Protocol

- **Assumptions**
  - each site uses the write-ahead-log protocol
  - at most one site can fail during the execution of the transaction

- **Basic Concept**

  Before the commit protocol begins, all the sites are in state q. If the coordinator fails while in state q1, all the **cohorts** perform the **timeout transition**, thus aborting the transition. Upon recovery, the **coordinator** performs the **failure transition**.

Concurrency Control in DS

- When several transactions run simultaneously in different processes, we must guarantee the final result is the same as the result of running the transactions sequentially in some order. That is, we need a mechanism to guarantee isolation or serializability.

- This mechanism is the Concurrency Control.

- Three typical concurrency algorithms:
  - Using locks
  - Using timestamps
  - Optimistic

Concurrency Control Algorithm Using Locks

- This is the oldest and most widely used concurrency control algorithm.
- When a process needs to access shared resource as part of a transaction, it first locks the resource.
- If the resource is locked by another process, the requesting process waits until the lock is released.
- The basic scheme is overly restrictive. The algorithm can be improved by distinguishing read locks from write locks.
- Data only to be read (referenced) is locked in shared mode.
- Data need to be updated (modified) is locked in exclusive mode.

Lock Based Concurrency Control

- Lock modes have the following compatibility:
  - If a resource is not locked or locked in shared mode, a transaction can lock the resource in shared mode.
  - A transaction can lock a resource in exclusive mode only if the resource is not locked.

	Locked Mode		
Locking Mode	Unlocked	Shared	Exclusive
Shared	○	○	×
Exclusive	○	×	×

Lock Based Concurrency Control

- If two transactions are trying to lock the same resource in incompatible mode, the two transactions are in ***conflict***.

- A conflict relation can be
  - shared-exclusive conflict (or read-write conflict)
  - exclusive-exclusive conflict (or write-write conflict).

Two Phase Locking

- Using locks does not necessarily guarantee serializability.

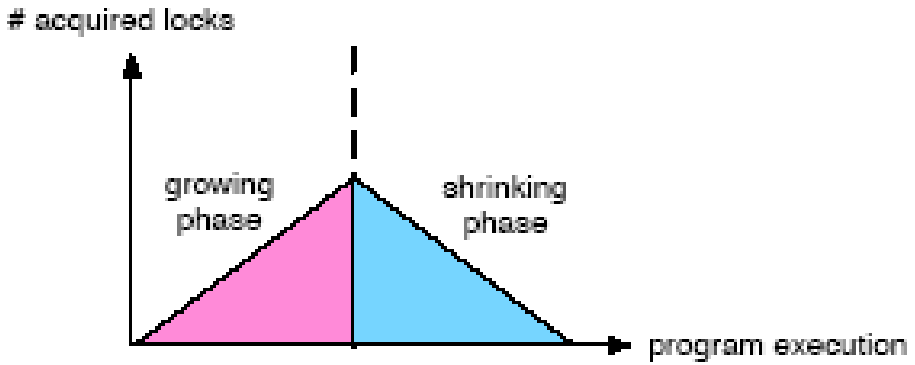
- Only transactions executed concurrently as follows will be serializable.

  1. A transaction locks shared resource before accessing the resource and release the lock after using it.
  2. Locking is granted according to the compatibility constraint.
  3. A transaction does not acquire any locks after releasing a lock.

- The usage of locks that satisfies above conditions are 2 phase lock protocol.

Two Phase Locking

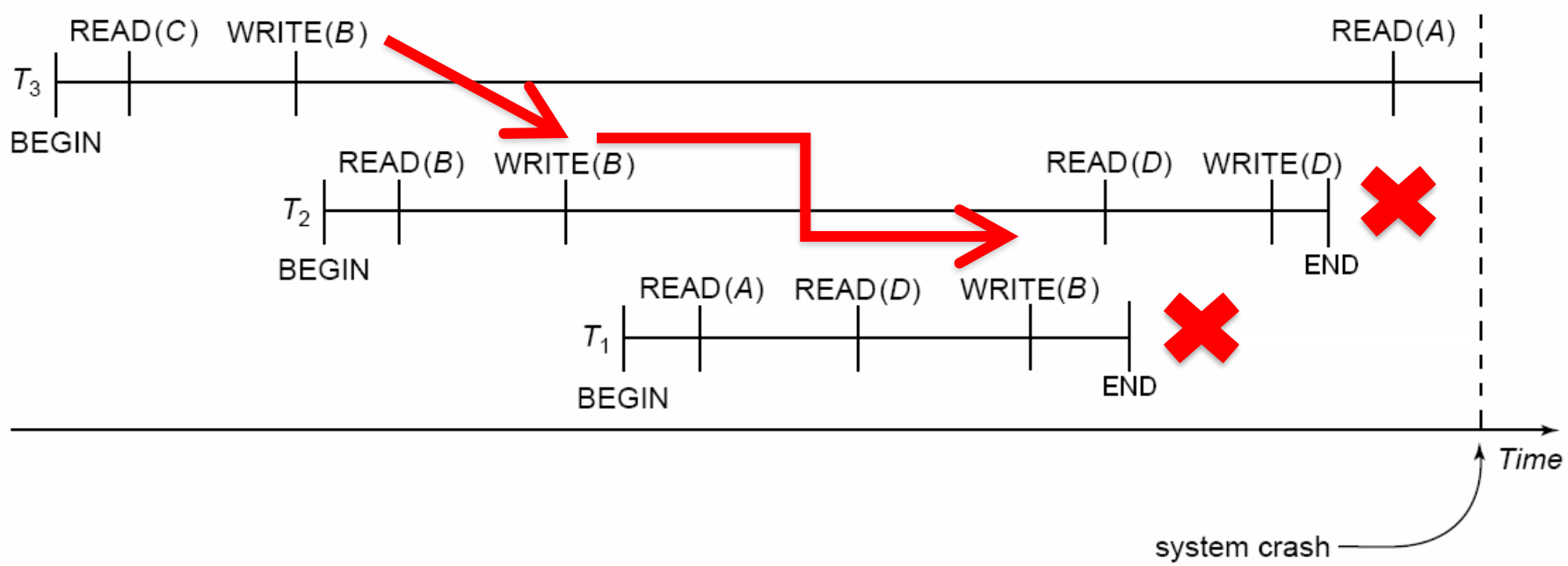
- A transaction acquires all necessary locks in the first phase, growing phase. The transaction releases all the locks in the second phase, shrinking phase.



- Two phase locking is the sufficient condition to guarantee serializability

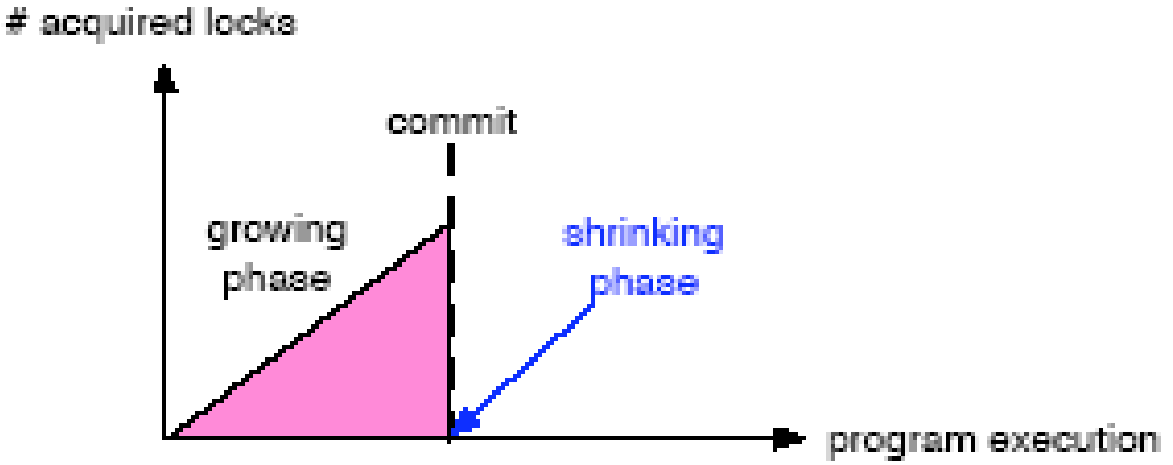
- Suppose a transaction aborts after releasing some of the locks. If other transactions have used resources protected by these released locks, the transactions must also be aborted - ***cascading abort***.

Cascading Abort



Strict Two Phase Locking

- To avoid the cascading abort, many systems use “strict two-phase locking” in which transactions do not release any lock until commit is completed.
- Strict two-phase locking is done as follows:
  - ***Begin transaction***
  - Acquire locks before read or write resources
  - Commit
  - Release all locks
  - ***End transaction***



Strict Two Phase Locking

- The strict two-phase locking has the following advantages:
  - A transaction always reads a value written by a committed transaction.
    - A transaction never has to be aborted - the cascading abort never happens.
  - All lock acquisitions and releases can be handled by the system without the transaction being aware of these.
    - Locks are acquired whenever data are accessed and released when the transaction has finished.
- Two-phase locking, can cause deadlocks to occur.
  - two processes try to acquire the same pair of locks but in the reverse order.

Lock Granularity

- The size of resources that can be locked by a single lock is called "lock granularity."
- Smaller the granule, the more concurrent processing is possible while it requires more locks.
- For example, let's compare locking ***by file*** and locking ***by record*** contained in a file.
  - Suppose two transactions access different records in the same file. If record is the unit of lock, two transactions can access the data simultaneously.
  - On the other hand, if file is the unit of lock, the two transactions cannot access the data simultaneously

Concurrency Control Algorithm Using Timestamps (by Reed)

- Assign each transaction a timestamp at ***Begin Transaction.***
- Timestamps must be unique (we can ensure the uniqueness using Lamport's algorithm).
- Each resource has a read timestamp and a write timestamp.

Timestamp Based Concurrency Control

- A ***read timestamp*** is the timestamp of the transaction which read the resource most recently. Let $\boldsymbol{TRD(x)}$ be the read timestamp of resource $\boldsymbol{x}$.

- A ***write timestamp*** is the timestamp of the transaction which updated the resource most recently. Let $\boldsymbol{TWR(x)}$ be the write timestamp of resource $\boldsymbol{x}$.

- Note that Read timestamps and Write-timestamps are **NOT** the values of actual time when the data items are being actually read or written.

Timestamp Based Concurrency Control

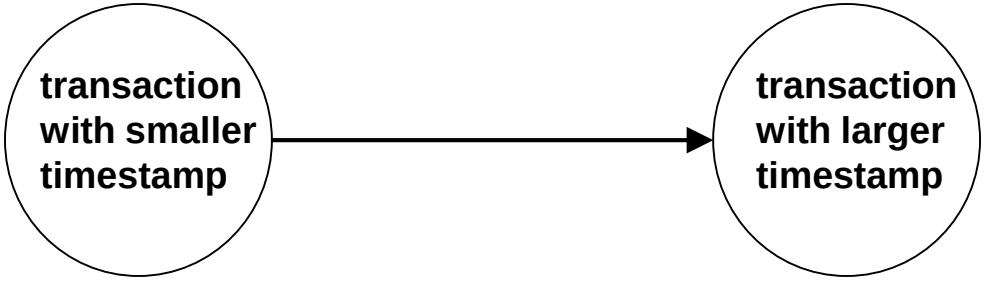
- When a transaction with timestamp T tries to read resource x,
  - If $T < TWR(x)$, that means the transaction needs to read a value of x that is already overwritten. Hence the request is rejected and the transaction is aborted.
  - If $T \geq TWR(x)$, the transaction is allowed to read the resource.

Timestamp Based Concurrency Control

- When a transaction with timestamp T tries to update resource x,
  - If $T < TWR(x)$ the transaction is attempting to write an obsolete value of x the request is rejected and the transaction is aborted.
  - if $T < TRD(x)$, then the value of x that the transaction is producing was needed previously, and the system assumed that value would never be produced, the request is rejected and the transaction is aborted.
  - Otherwise, the transaction is allowed to update the resource.

Timestamp Based Concurrency Control

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



- Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be recoverable.

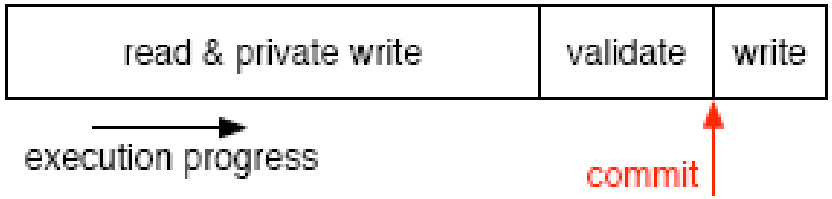
Optimistic Concurrency Control (by Kung & Robinson)

- Run the transactions, be optimistic about transaction conflicts.

- At the end of a transaction, i.e. when it is committing, existence of conflict is examined. If no conflict is detected the local copies are written on the real resource. Otherwise, the transaction is aborted.

  → Progression in phases

  Phase: 1 2 and 3



- The conflict check is done by examining <u>whether the resource used by this transaction is updated by other transaction(s) while this transaction was executing</u>.
  - To make this check possible, the system must keep track of the resources used by the transactions.

Optimistic Concurrency Control

Time-stamping

Each transaction is assigned a *timestamp* TS(Ti) at the beginning of the validation phase.

Validation conditions

For every pair of transactions Ti, Tj, such that TS(Ti) < TS(Tj), one of the following validation conditions must hold:

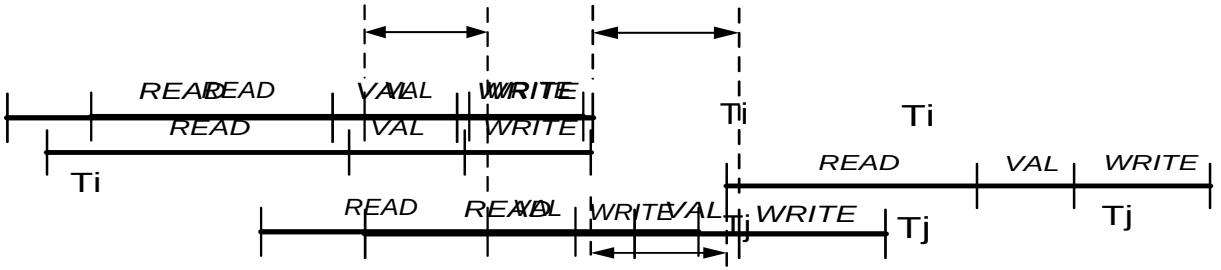
1. Ti completes (all three phases) before Tj begins.

2. Ti completes before Tj starts its write phase, and Ti does not write any object read by Tj.

3. Ti completes its Read phase before Tj completes its Read phase, and Ti does not write any object that is either read or written by Tj.

Conditions for validation

3.
2. 1.



and Read Set (Ti) ∩ Write Set (Tj) = Ø

and Read Set (Tj) ∩ Write Set (Ti) = Ø

and Write Set (Tj) ∩ Write Set (Ti) = Ø

Optimistic Concurrency Control Observations

- Optimistic concurrency control is designed with the assumption that conflicts are rare and we do not need to abort transactions often.
- Update is performed on local copies. This method fits well with the private workspace.
- The advantage is that it allows maximum parallel execution since transactions never wait and deadlocks never occur.
- The disadvantage is that a transaction may be aborted at the very end since the conflict check is done at the commit point. All operations of the aborted transaction must be done again from the start.

Summation

- **Why do we need Election Algorithms?**
  - To coordinate distributed transactions
- **Two Election Algorithms?**
  - Bully Algorithm, Ring Algorithm
- **Four Key properties of a distributed transaction?**
  - Atomic, consistent, isolated, durable
- **Main programming primitives of a distributed transaction?**
  - Begin Transaction, Abort, Read, Write, End Transaction
- **Two main approaches to implement distributed transactions?**
  - Private workspace, Write-ahead Log
- **Two distributed commit protocols?**
  - 2 Phase commit, 3 Phase Commit
- **Why Concurrency Control is necessary for distributed transactions?**
  - For implementing serialization, deadlock prevention, improving parallelism
- **Three Schemes for implementing Concurrency Control in Distributed Systems?**
  - Lock based CC
  - Timestamp based CC
  - Optimistic CC

References

References

[1] H. Garcia-Molina, 1982. Elections in a Distributed Computing System, *IEEE Transactions on Computers. C-31,1* January 1982) pp 48-59

[2] Witchel, Distributed Coordination, Univeristy of Texas, Available at: http://www.cs.utexas.edu/users/witchel/372/lectures/25.DistributedCoordination.ppt

[3] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213-226. DOI=10.1145/319566.319567 http://doi.acm.org/10.1145/319566.319567

[4] Q.E.K. Mamun, H. Nakazato. 2005. Timestamp Based Optimistic Concurrency Control, *TENCON 2005 2005 IEEE Region 10* , vol., no., pp.1-5, 21-24 Nov. 2005