ECE151 – Lecture 8

Chapter 5

Synchronization

Mutual Exclusion

Problem: A number of processes in a distributed system want exclusive access to some resource.

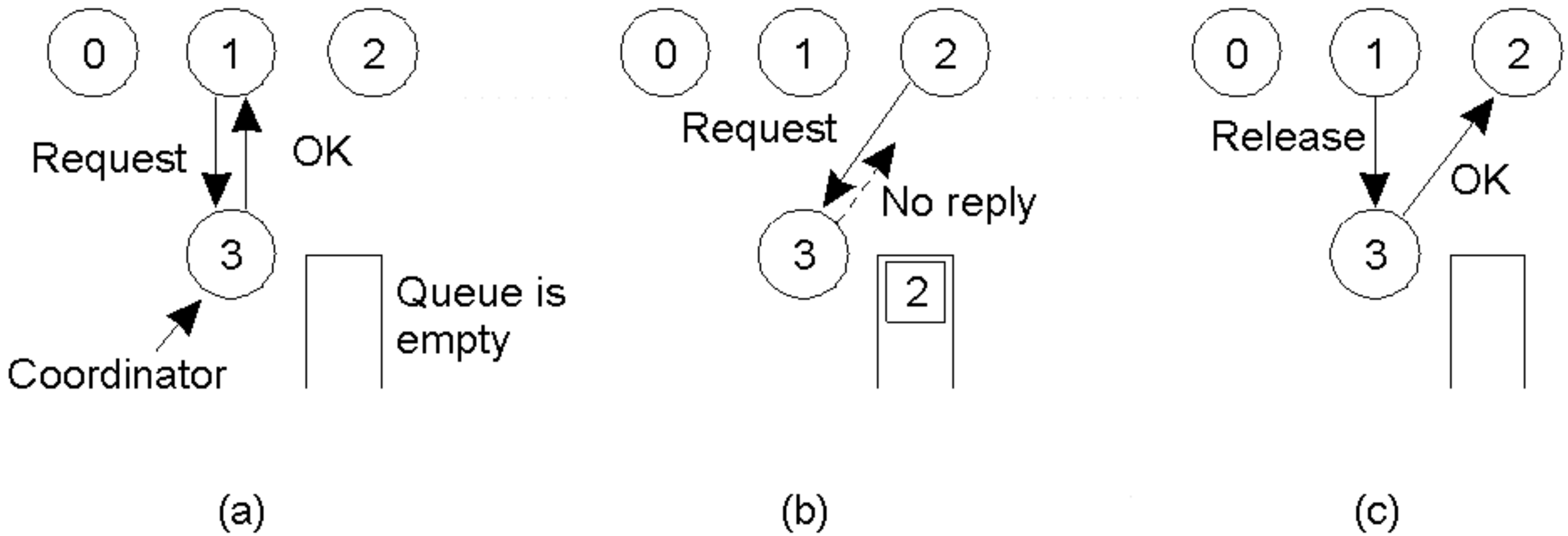
Basic solutions:

Via a centralized server.

Completely distributed, with no topology imposed.

Completely distributed, making use of a (logical) ring.

Mutual Exclusion: A Centralized Algorithm



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

Mutual Exclusion: Ricart & Agrawala

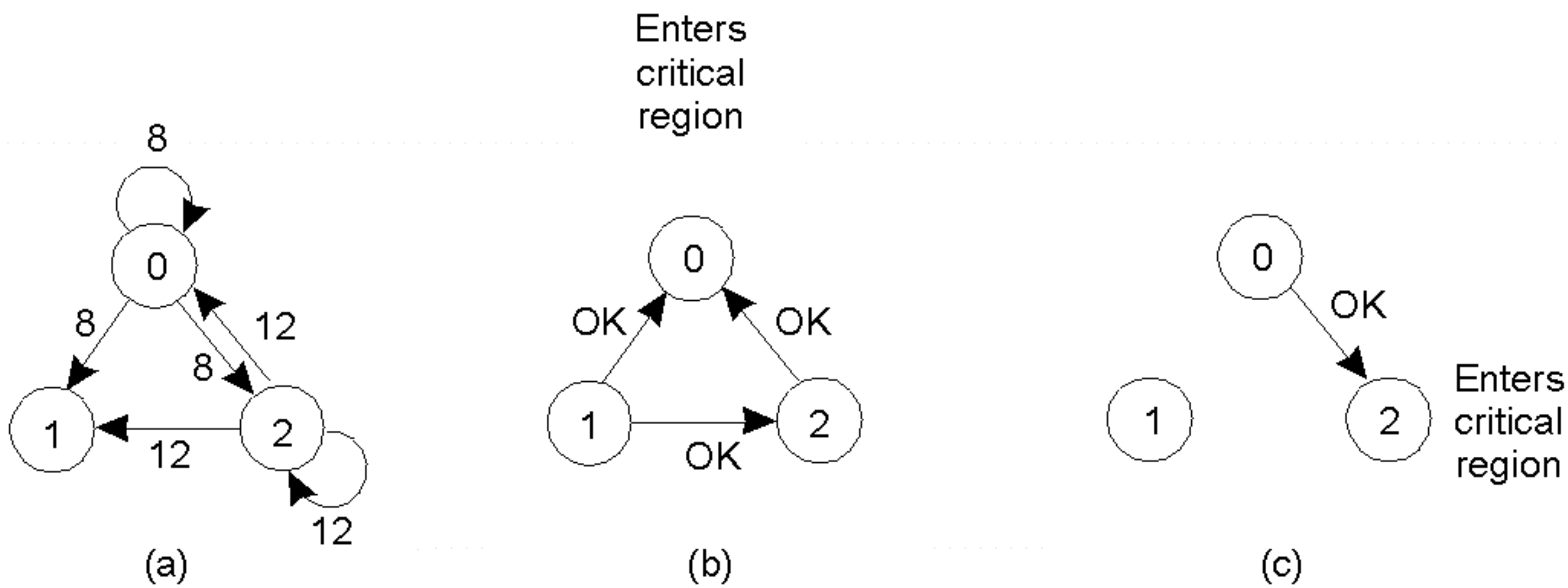
Principle: The same as Lamport except that acknowledgments aren't sent. Instead, replies (i.e. grants) are sent only when:

The receiving process has no interest in the shared resource; or

The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is **deferred**, implying some more local administration.

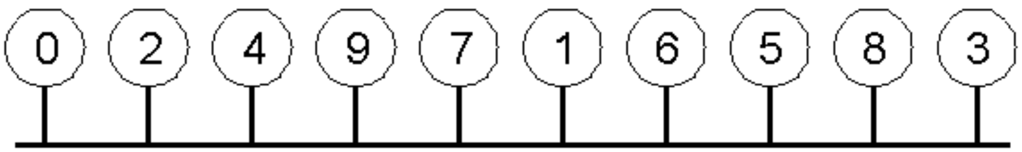
A Distributed Algorithm



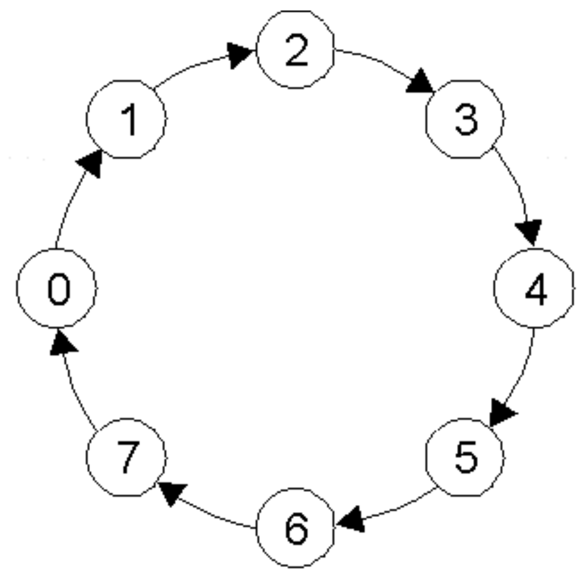
a) Two processes want to enter the same critical region at the same moment.
b) Process 0 has the lowest timestamp, so it wins.
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

A Token Ring Algorithm

Essence: Organize processes in a *logical* ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to)



(a)



(b)

a) An unordered group of processes on a network.
b) A logical ring constructed in software.

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	1 to ∞	0 to $n-1$	Lost token, process crash

A comparison of three mutual exclusion algorithms.

Distributed Transactions

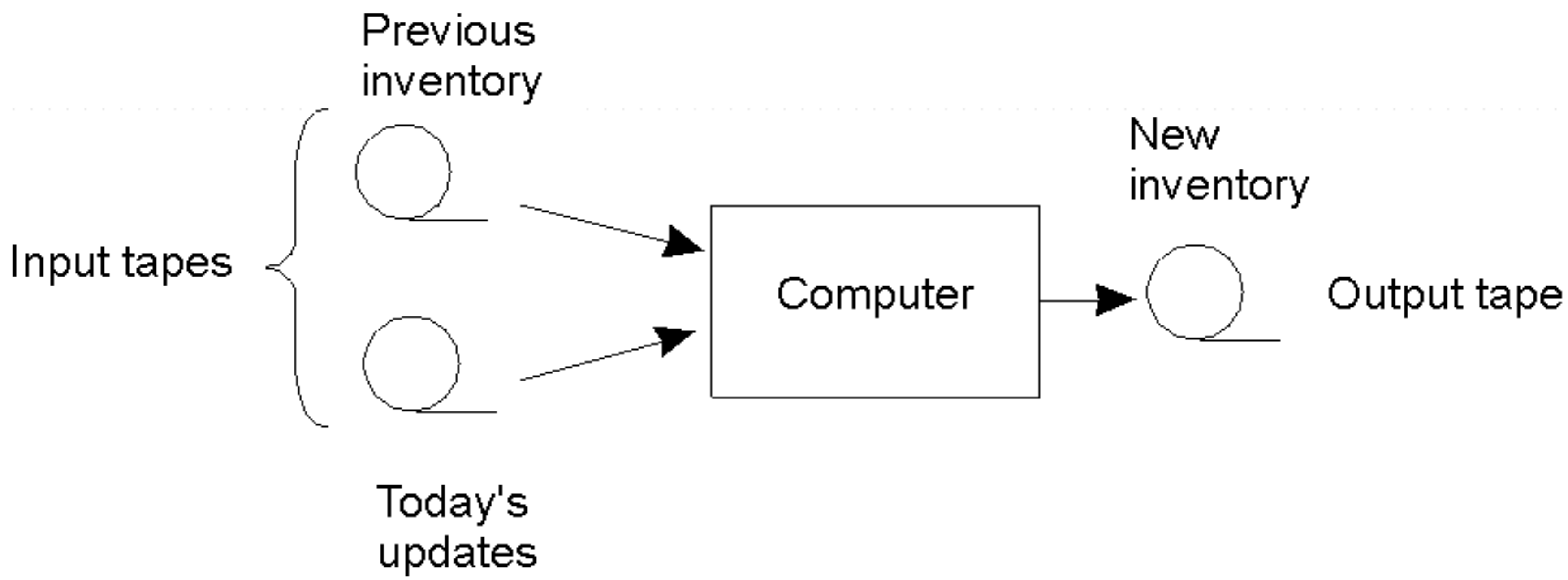
The transaction model

Classification of transactions

Concurrency control

The Transaction Model

Updating a master tape is fault tolerant.



The Transaction Model

Examples of primitives for transactions.

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

The Transaction Model

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
            (a)
```

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
            (b)
```

a) Transaction to reserve three flights commits
b) Transaction aborts when third flight is unavailable

Essential: All READ and WRITE operations are executed, i.e. their effects are made permanent at the execution of END_TRANSACTION .

Observation: Transactions form an **atomic** operation.

ACID Properties

Model: A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties:

Atomicity: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.

Consistency: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction's execution.

Isolation: Concurrent transactions do not interfere with each other. It appears to each transaction *T* that other transactions occur either *before T*, or *after T*, but never both.

Durability: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.

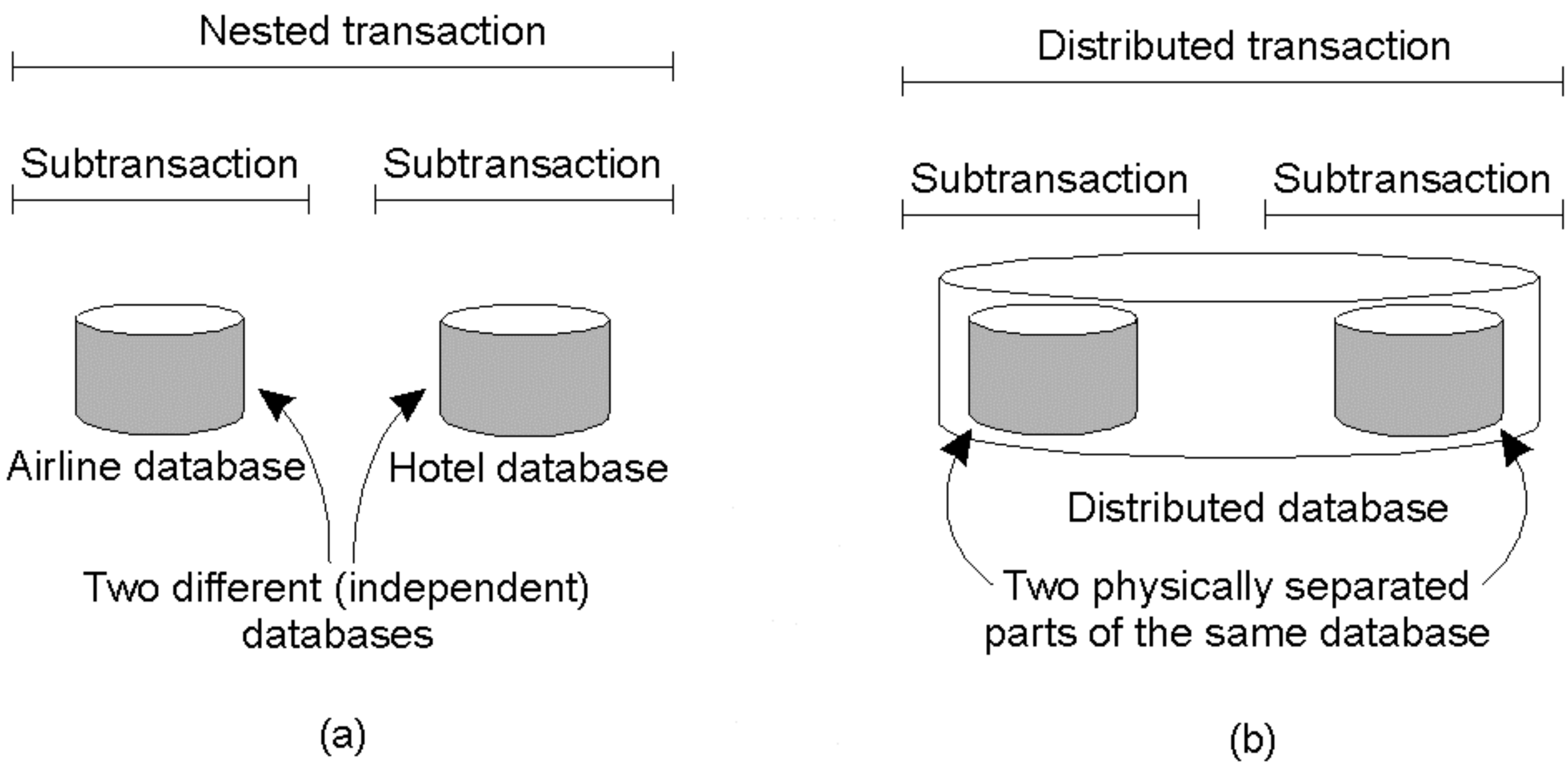
Transaction Classification

Flat transactions: The most familiar one: a sequence of operations that satisfies the ACID properties.

Nested transactions: A *hierarchy* of transactions that allows (1) concurrent processing of subtransactions, and (2) recovery per subtransaction.

Distributed transactions: A (flat) transaction that is executed on distributed data often implemented as a two-level nested transaction with one subtransaction per node.

Distributed Transactions



Flat Transactions: Limitations

Problem: Flat transactions constitute a very simple and clean model for dealing with a sequence of operations that satisfies the ACID properties. However, after a series of successful operations *all* changes should be undone in the case of failure. Sometimes unnecessary:

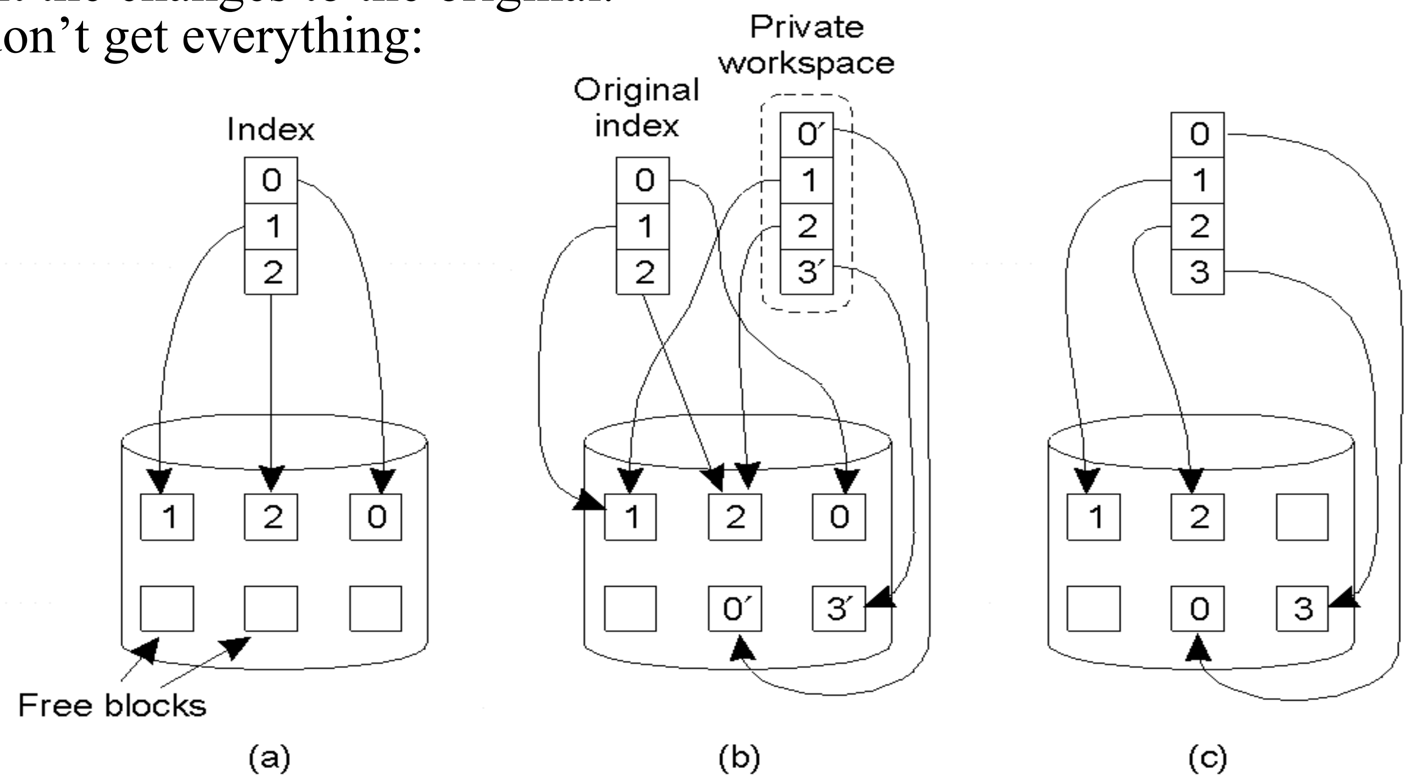
Trip planning. Plan a intercontinental trip where all flights have been reserved, but filling in the last part requires some "experimentation." The first reservations are *known* to be in order, but cannot yet be committed.

Bulk updates. When updating bank accounts for monthly interests we have to lock the entire database (every account should be updated exactly once: it is a transaction over the entire database.)

Better: each update is immediately committed. However, in the case of failure, we'll have to be able to continue where we left off.

Private Workspace

Solution 1: Use a **private workspace**, by which the client gets its own copy of the (part of the) database. When things go wrong delete copy, otherwise commit the changes to the original.
Optimization: don't get everything:



a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

Writeahead Log

Solution 2: Use a writeahead log in which changes are recorded allowing you to **roll back** when things go wrong:

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

a) A transaction

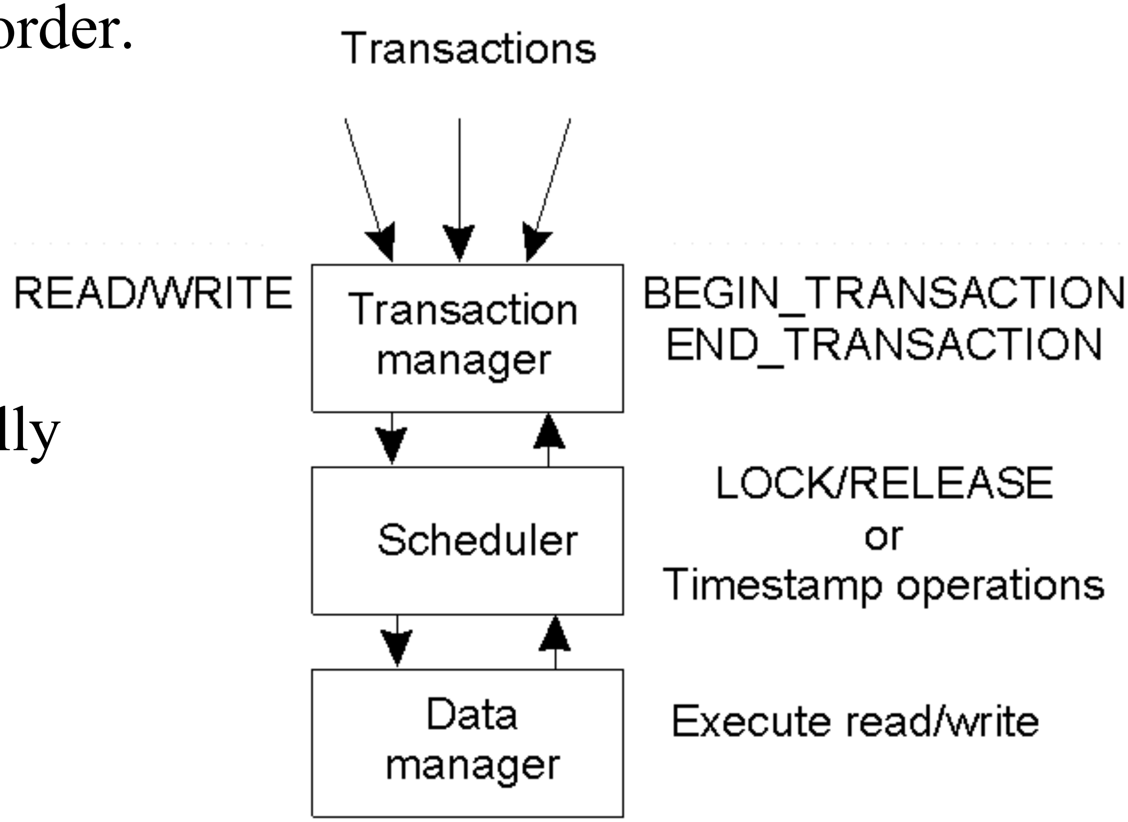
b) – d) The log before each statement is executed

Concurrency Control

Problem: Increase efficiency by allowing several transactions to execute at the same time.

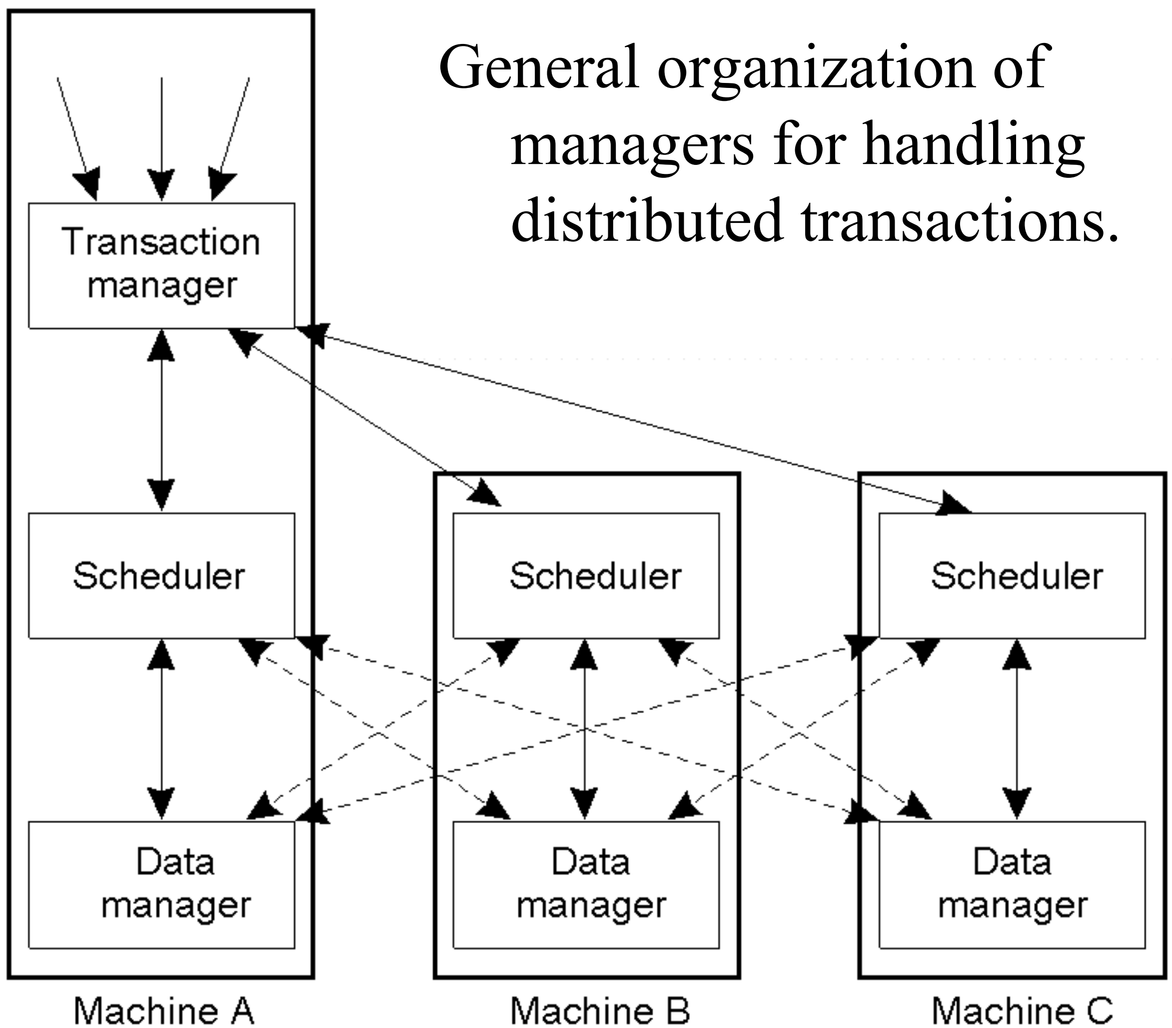
Constraint: Effect should be the same as if the transactions were executed in some serial order.

Question: Does it actually make sense to allow concurrent transactions on a single server?



General organization of managers for handling transactions.

Concurrency Control



General organization of managers for handling distributed transactions.

Question:
What about a distributed transaction manager?

Serializability

Consider a collection E of transactions $T1$, … Tn.

Goal is to conduct a **serializable execution** of E:

- Transactions in E are possibly concurrently executed according to some schedule S.
- Schedule S is equivalent to some *totally ordered* execution of $T1$, … Tn.

Serializability

BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION

(a)

BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION

(b)

BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

a) – c) Three transactions T_1, T_2, and T_3

d) Possible schedules

Serializability

Note: Because we're not concerned with the computations of each transaction, a transaction can be modeled as a *log* of **read** and **write** operations.

Two operations *Op(Ti, x)* and *Op(Tj, x)* on the same data item *x*, and from a set of logs may **conflict** at a data manager:

read-write conflict (rw): One is a read operation while the other is a write operation on *x*.

write-write conflict (ww): Both are write operations on *x*.

Basic Scheduling Theorem

Let **T** = *{T*1, … *Tn} be a set of transactions and let E be an execution of these transactions modeled by logs {L*1, … *Ln}*.

E is serializable
if there exists a total ordering of **T** *such that*
for each pair of conflicting operations Oi and Oj
from distinct transactions Ti and Tj (respectively),
*Oi precedes Oj in any log L*1, … *Ln,*
if and only if
Ti precedes Tj in the total ordering.

Basic Scheduling Theorem

Note: The important thing is that we process conflicting reads and writes in certain relative orders. This is what concurrency control is all about.

Note: It turns out that read-write and write-write conflicts can be synchronized *independently*, as long as we stick to a total ordering of transactions that is consistent with both types of conflicts.

Synchronization Techniques

Two-phase locking: Before reading or writing a data item, a lockmust be obtained. After a lock is given up, the transaction is not allowed to acquire any more locks.

Timestamp ordering: Operations in a transaction are timestamped, and data managers are forced to handle operations in timestamp order.

Optimistic control: Don't prevent things from going wrong, but correct the situation if conflicts actually did happen. Basic assumption: you can pull it off in most cases.

Two-phase Locking

Clients do only READ and WRITE operations within transactions.

Locks are **granted** and **released** only by scheduler.

Locking policy is to avoid **conflicts** between operations

Two-phase Locking

Rule 1: When client submits *Op(Ti,x)*, scheduler tests whether it conflicts with an operation *Op(Tj,x)* from some other client. If no conflict then grant *Op(Ti,x)*, otherwise delay execution of *Op(Ti,x)*.

Conflicting operations are executed in the same order as that locks are granted.

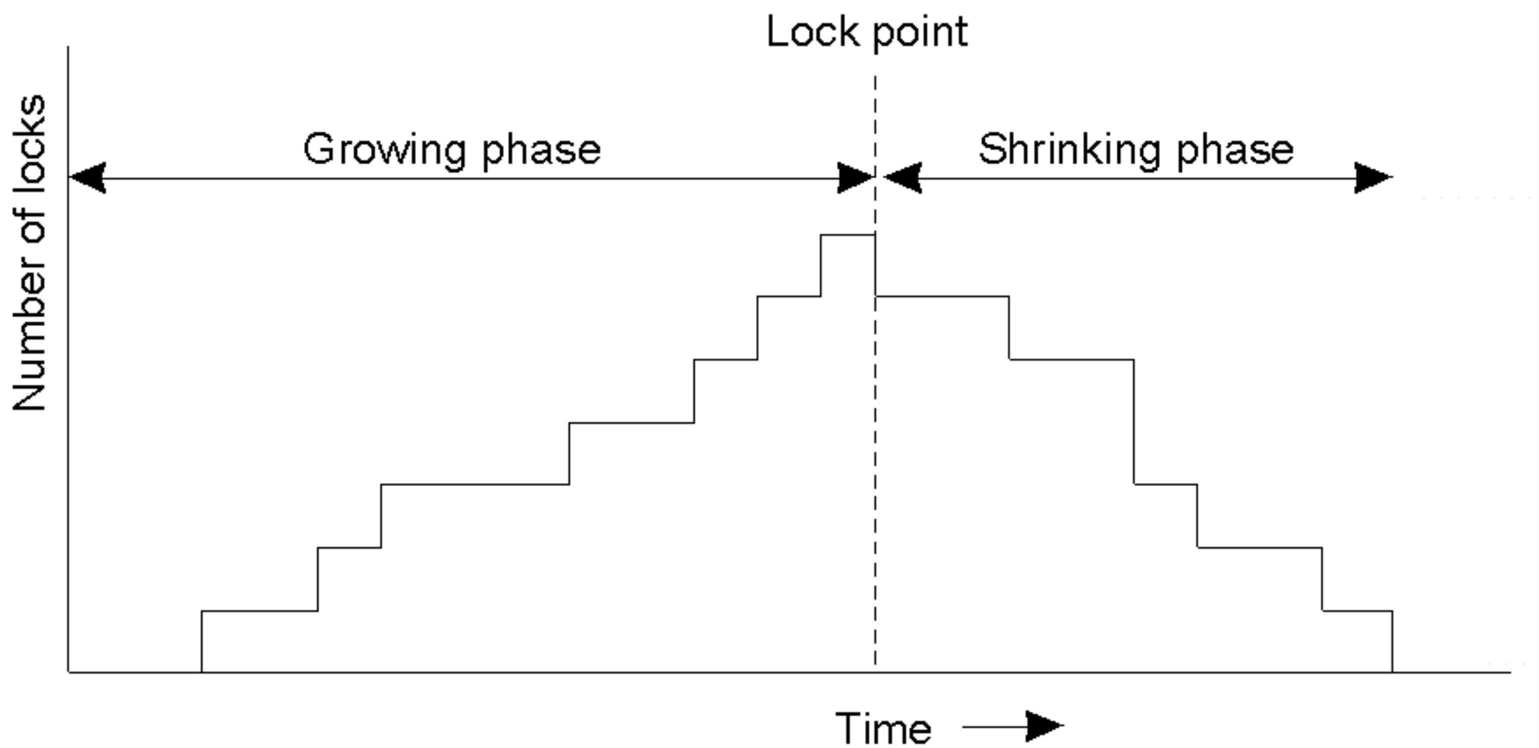
Rule 2: If Op(*Ti,x)* has been granted, do not release the lock until Op(*Ti,x)* has been executed by data manager.

Guarantees LOCK => Op => RELEASE *order.*

Rule 3: If RELEASE*(Ti,x)* has taken place, no more locks for *Ti* may be granted.

Combined with rule 1, guarantees that all pairs of conflicting operations of two transactions are done in the same order.

Two-Phase Locking



Centralized 2PL: A single site handles all locks

Primary 2PL: Each data item is assigned a primary site to handle its locks. Data is not necessarily replicated

Distributed 2PL: Assumes data can be replicated. Each primary is responsible for handling locks for its data, which may reside at remote data managers.

Two-phase Locking: Problems

Problem 1: System can come into a **deadlock**. *How?*

Practical solution: put a timeout on locks and abort transaction on expiration.

Problem 2: When should the scheduler actually release a lock:

(1) when operation has been executed

(2) when it knows that no more locks will be requested

No good way of testing condition (2) unless transaction has been committed or aborted.

Moreover: Assume the following execution sequence takes place:

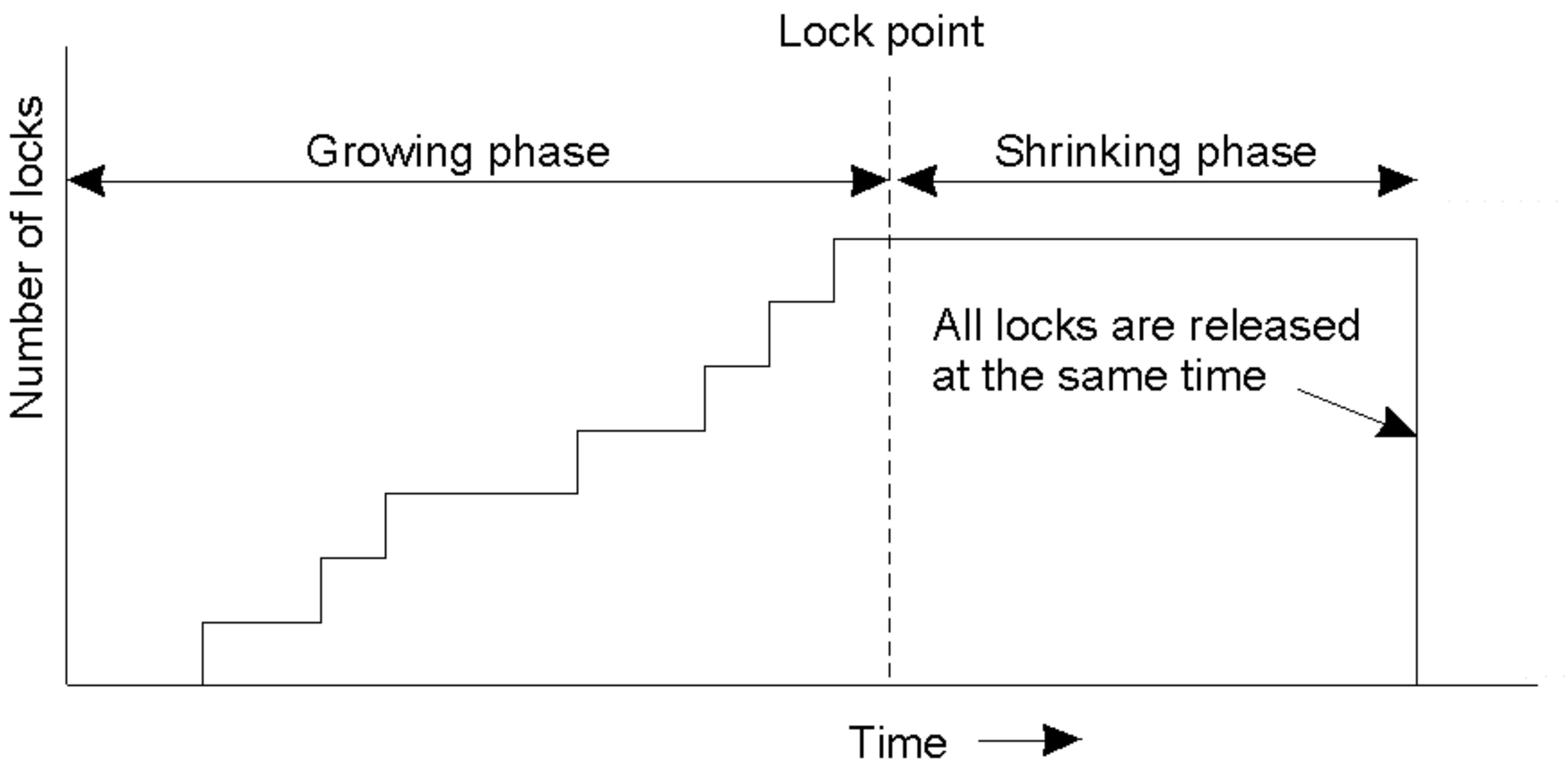
RELEASE*(Ti,x)* => LOCK*(Tj,x)* => ***ABORT****(Ti)*.

Consequence: scheduler will have to abort *Tj* as well (**cascaded aborts**).

Solution: Release *all* locks only at commit/abort time (**strict two-phase locking**).

Two-Phase Locking

Strict two-phase locking.



Timestamp Ordering

Basic idea:

Transaction manager assigns a unique timestamp TS*(Ti)* to each transaction *Ti*.

Each operation Op*(Ti,x)* submitted by the transaction manager to the scheduler is timestamped TS(Op*(Ti,x))* = TS*(Ti)*.

Scheduler adheres to following rule:

If Op*(Ti,x)* and Op*(Tj,x)* conflict
then data manager processes
Op*(Ti,x)* before Op*(Tj,x)*
iff TS(Op*(Ti,x))* < TS(Op*(Tj,x))*

Note: rather aggressive since
if a single Op*(Ti,x)* is rejected, *Ti* will have to be aborted.

Timestamp Ordering

Suppose: TS(Op*(Ti,x)* < TS(Op*(Tj,x))*, but that Op*(Tj,x)* has already been processed by the data manager.

Then: the scheduler rejects Op*(Ti,x)*, as it came in *too late*.

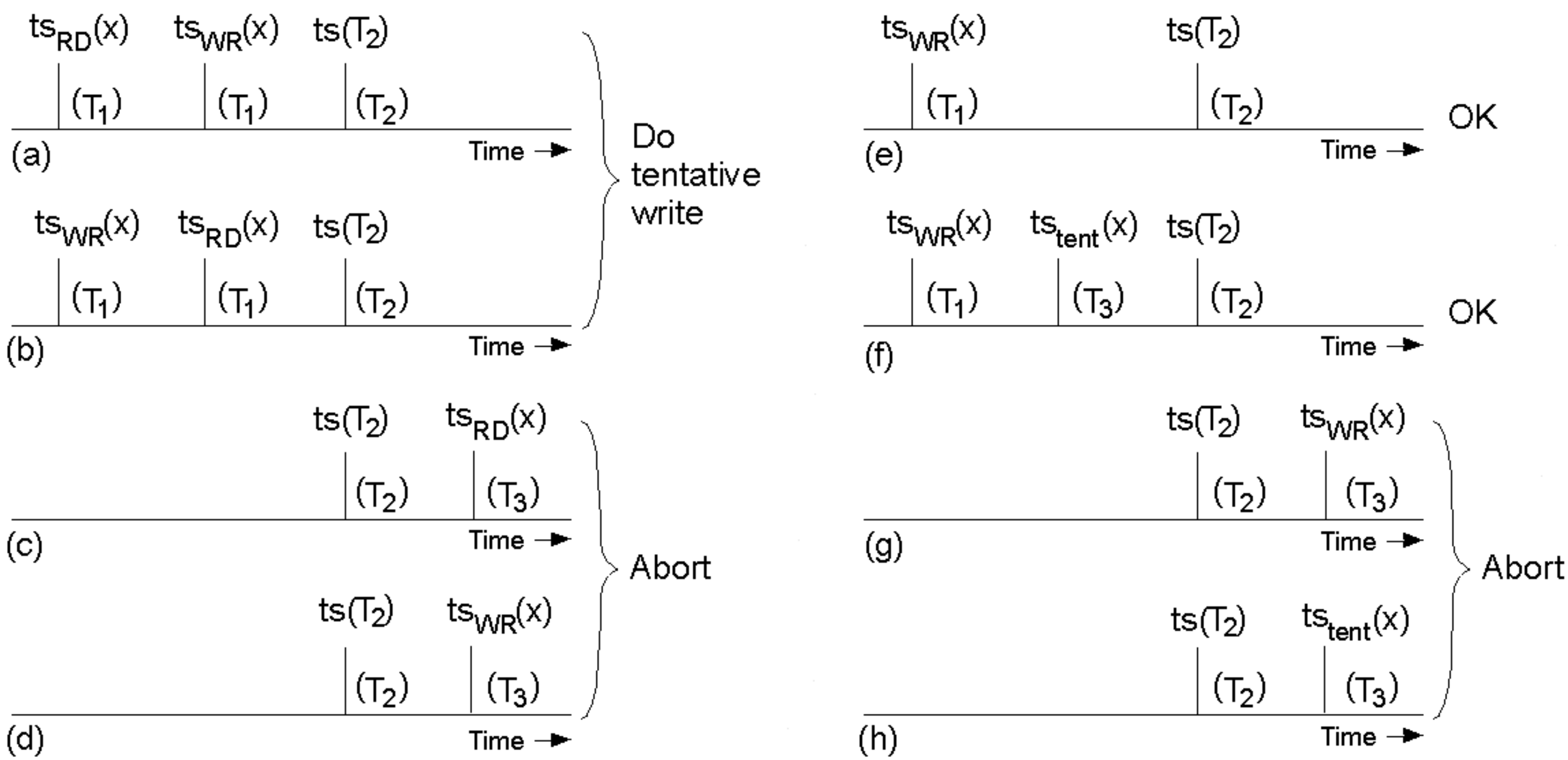
Suppose: TS(Op*(Ti,x))* < TS(Op*(Tj,x))*, and that Op*(Ti,x)* has already been processed by the data manager.

Then: the scheduler would submit Op*(Tj,x)* to data manager.

Refinement: hold back Op*(Tj,x)* until *Ti* commits or aborts.

Question: Why would we do this?

Pessimistic Timestamp Ordering



Optimistic Concurrency Control

Observation: (1) Maintaining locks costs a lot;
(2) In practice not many conflicts.

Alternative: Go ahead immediately with all operations, use tentative writes everywhere (shadow copies), and solve conflicts later on.

Phases: allow operations
tentatively validate effects
make updates permanent.

Validation: Check two basic rules for each pair of active transactions *Ti* and *Tj*:

Rule 1: *Ti* must not read or write data that has been written by *Tj*.

Rule 2: *Tj* must not read or write data that has been written by *Ti*.

If one of the rules doesn't hold: **abort** one of the transactions.