CS 5330/7330

Two phase locking

Table of contents

The story so far

- Isolation: one key requirement for transaction
- Isolation as conflict serializability
- Why conflict serializability
  - Cycles in serializability
  - Multiple transactions share the same objects
  - The contention goes in *both* directions

So …

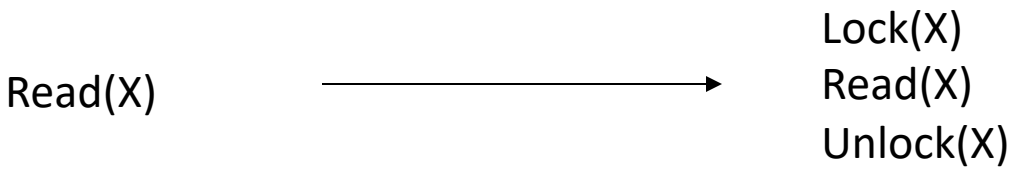
- Need to provide *protocol* (rules on how data item is accessed) to ensure conflict serializability
- Goal of protocol:
  - To allow access for data items that are not required by multiple transactions
  - For those data items required by multiple transaction, restrict access in some way, or limit it to exclusive access
- Balance between safety and efficiency
  - Too restrictive: little or no concurrency, ineffective
  - Too lenient: leads to inconsistency

Lock-based protocols

- “Exclusive” access $\Rightarrow$ locks
- Each database item is associated with locks
- Transaction must obtain locks before accessing the object
- Transaction must release lock when it finishes.

Lock-based protocols

- Example:



- Lock(X): check if object X is already locked
  - If not, obtain the lock
  - If so, wait or “do something” to handle the potential deadlock (like aborting)
- One does not have to read immediately after locking
- Unlock(X): release the lock on object X
- The addition of Lock(X) and Unlock(X) commands are done by the DBMS

Lock-based protocols: S and X locks

- How many transaction can obtain the lock to an item?
  - One
- Too restrictive?
- Consider the two transactions

T1	T2
1. Read(X)	1. Read(Y)
2. Read(Y)	2. Read(X)
3. Read(Z)	3. Read(Z)

- There seems to be no reason for one transaction to wait for the other

Lock-based protocols: S and X locks

- Two kinds of locks on each object
  - Shared locks (S-locks)
    - Requested before reading
    - Multiple transactions can hold a S-lock on an object simultaneously
  - Exclusive locks (X-locks)
    - Requested before writing
    - Only one transaction can hold an X-lock on an object at any given time
    - No other transaction can hold any lock (not even a S-lock) if some transaction has an X-lock

Lock-based protocols: S and X locks

- More on S and X locks
  - A transaction that holds an S-lock on an object can read the object
  - A transaction that holds an X-lock on an object can read and write the objects
- Lock-compatibility table

T1 Request \ T2 holds	S-lock	X-lock
S-lock	True	False
X-lock	False	False

Lock-based protocol

- Consider the following examples (again):

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

T1 (Transfer)

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 * 1.01
6. Write(Y, A2)

T2 (Dividend)

Lock-based protocol -- example

- With a lock-based protocol, one possible way T1 is transformed

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

T1 (Transfer)

1. S-lock(X)
2. A1 <- Read(X)
3. A1 <- A1 – k
4. X-lock(X)
5. Write(X, A1)
6. X-lock(Y)
7. A2 <- Read(Y)
8. A2 <- A2 + k
9. Write(Y, A2)
10. Unlock(X)
11. Unlock(Y)

Lock-based protocol -- example

- Notes from the previous example:
  - Locks must be obtained before read/write can begin
  - If a transaction want to read and write the same object, it can either
    - Obtain an X-lock before reading
    - Obtain an S-lock before reading, then obtain an X-lock before writing (it is not automatically granted)
  - A transaction does not have to release locks immediately after use
    - Good or bad?

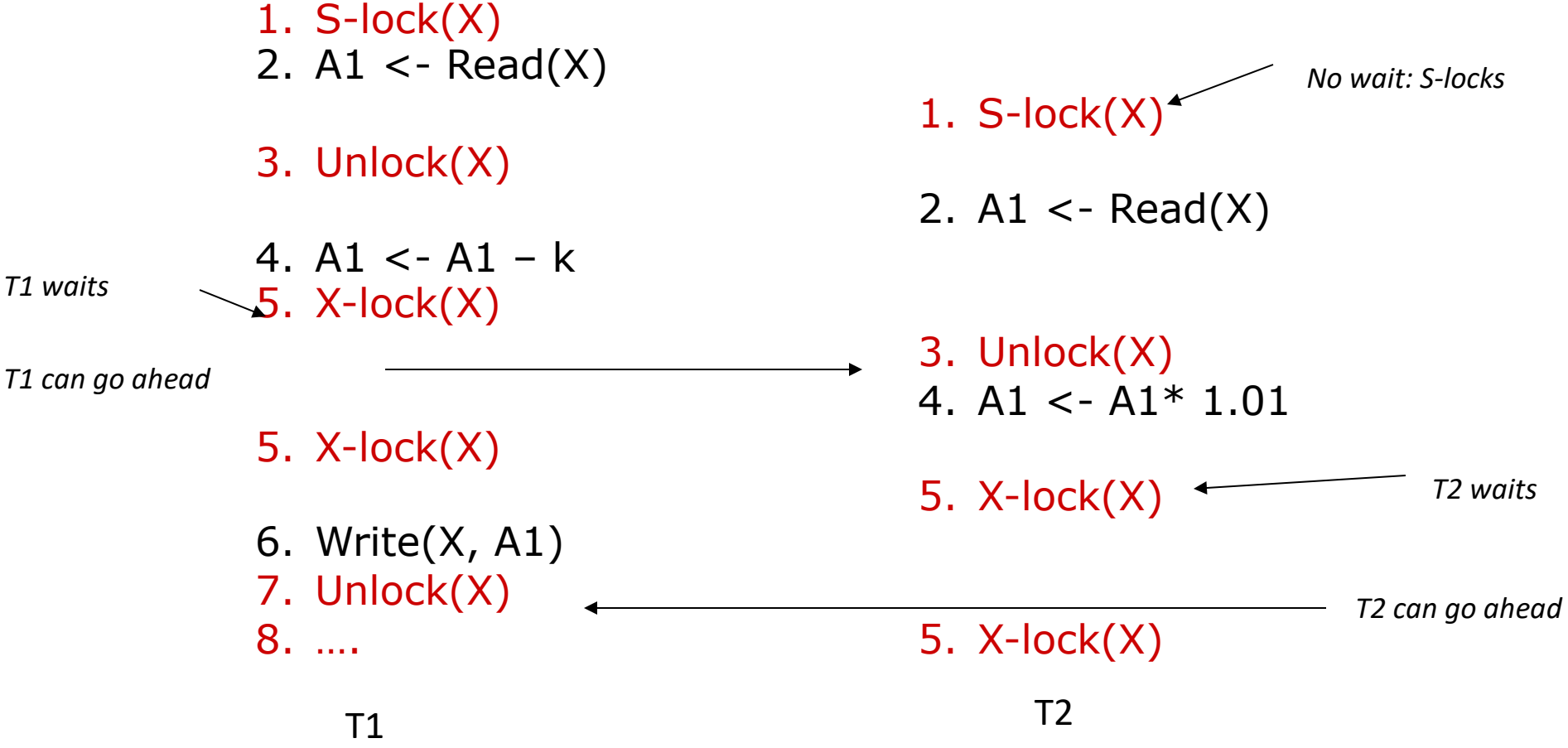
Lock-based protocol -- example

- Example of schedule with locks

T1	T2
1. S-lock(X)	1. S-lock(X)
2. A1 <- Read(X)	2. A1 <- Read(X)
3. Unlock(X)	3. Unlock(X)
4. A1 <- A1 – k	4. A1 <- A1* 1.01
5. X-lock(X)	5. X-lock(X)
6. Write(X, A1)	6. Write(X, A1)
7. Unlock(X)	7. Unlock(X)
8. S-lock(Y)	8. S-lock(Y)
9. A2 <- Read(Y)	9. A2 <- Read(Y)
10.Unlock(Y)	10.Unlock(Y)
11.A2 <- A2 + k	11.A2 <- A2 * 1.01
12.X-lock(Y)	12.X-lock(Y)
13.Write(Y, A2)	13.Write(Y, A2)
14.Unlock(Y)	14.Unlock(Y)

Lock-based protocol -- example

- Example of schedule with locks



Lock based protocols -- questions

- Does having locks this way guarantee conflict serializability?
- Is there any other requirements in the order/manner of accquiring/releasing locks?
- Does it matter when to acquire locks?
- Does it matter when to release locks?

Lock based protocols – need for a protocol

- Suppose we request a lock immediately before reading/writing
- We release a lock immediately after each read/write
- Does it guarantee serializability?

Lock based protocol – need for a protocol

- But with the following schedule

1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 * 1.01
6. Write(Y, A2)

4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)

X : 100 -> 50 -> 50.5; Y : 200 -> 202 -> 252; X+Y = 302.5

Not conflict serializable.

Lock based protocol – need for a protocol

S-lock (X)
1. A1 <- Read(X)

Unlock(X)
2. A1 <- A1 – k

X-lock (X)
3. Write(X, A1)

Unlock(X)

S-lock (X)
1. A1 <- Read(X)

Unlock(X)
2. A1 <- A1* 1.01
3. Write(X, A1)

Unlock(X), Slock(Y)
4. A2 <- Read(Y)

Unlock(Y)
5. A2 <- A2 * 1.01

X-lock (Y)
6. Write(Y, A2)

Unlock(Y)

S-lock (Y)
4. A2 <- Read(Y)

Unlock(Y)
5. A2 <- A2 + k

X-lock (Y)
6. Write(Y, A2)

Unlock(Y)

Not conflict serializable.

Lock based protocol – need for a protocol

- So the previous technique does not work.
- How about, only releasing a lock after all operations on the object is finished?

Lock based protocol – need for a protocol

S-lock (X)

1. A1 <- Read(X)
2. A1 <- A1 – k

X-lock (X)

3. Write(X, A1)

Unlock(X)

S-lock (X)

1. A1 <- Read(X)
2. A1 <- A1* 1.01
3. Write(X, A1)

Unlock(X), Slock(Y)

4. A2 <- Read(Y)
5. A2 <- A2 * 1.01

X-lock (Y)

6. Write(Y, A2)

Unlock(Y)

S-lock (Y)

4. A2 <- Read(Y)

X-lock (Y)

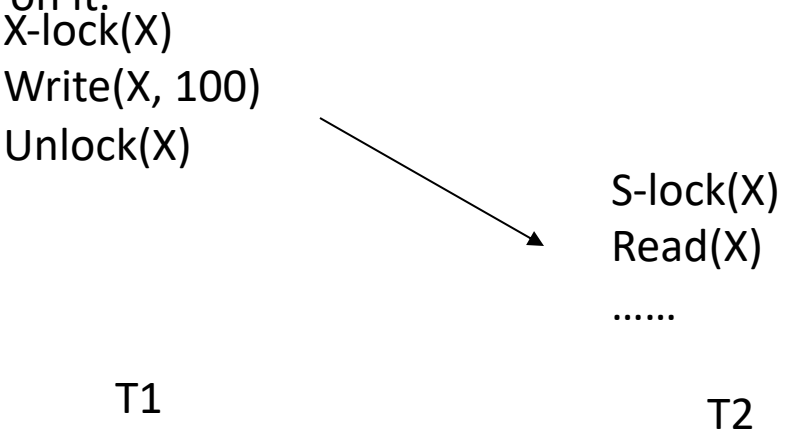
5. A2 <- A2 + k
6. Write(Y, A2)

Unlock(Y)

Not conflict serializable.

Two-phase locking -- motivation

- What is the problem?
- When a transaction release a lock on an object , that means other transactions can obtain a lock on it.



- In this case, there is contention from T1 to T2
- To ensure serializability, we must ensure there is no conflict from T2 back to T1
- How?

Two-phase locking -- motivation

- Ensure that T1 does not read/write anything that T2 read/write.
  - Unrealistic to check in real life
- What is a sufficient condition then?
- Ensure T1 does not read/write **anything** after releasing the lock!
- $\Rightarrow$ (basic) Two-phase locking

Two phase locking – definition

- The basic two-phase locking (2PL) protocol
  - A transaction *T* must hold a lock on an item *x* in the appropriate mode before *T* accesses *x*.
  - If a conflicting lock on *x* is being held by another transaction, *T* waits.
  - Once *T* releases a lock, it cannot obtain any other lock subsequently.
- Note: a transaction is divided into two phases:
  - A *growing phase* (obtaining locks)
  - A *shrinking phase* (releasing locks)
- Claim : 2PL ensures conflict serializability

Two phase locking – Notations

- If the operation is immaterial, we will simply use $\text{lock}_i[x]$ and $\text{unlock}_i[x]$ to denote the operations that T_i locks and unlocks x, respectively.
- $o_i[x]$: the execution of the operation o by T_i on x
- $<_s$: the ordering relationship between operations in the schedule S (We omit the subscript s if the context is clear)

Two phase locking – Notations

- 2PL implies
  - o-lock$_i[x] < o_i[x] < o$-unlock$_i[x]$ (if T_i commits)
    - i.e. you obtain a lock, than read/write, then unlock
  - Given two conflicting operations $p_i[x]$ and $q_j[x]$, we have either p-unlock$_i[x] < q$-lock$_j[x]$, or q-unlock$_j[x] < p$-lock$_i[x]$
    - i.e. if there are two conflicting operations, they cannot both hold the lockas at the same time
  - For any p-lock$_i[x]$ and q-unlock$_i[y]$, p-lock$_i[x] < q$-unlock$_i[y]$
    - i.e. Every transaction is 2-phase (you cannot lock something after you unlock anything)

Two phase locking – Proof

- By contradiction
  - If $T_i \rightarrow T_j$ in SG(S), we have a $p_i[x]$ conflict with a $q_j[x]$ and that $p_i[x] < q_j[x]$ in S
  - $\Rightarrow$ p-unlock$_i[x] <$ q-lock$_j[x]$
  - If $T_i \rightarrow T_j \rightarrow T_k$ in SG(S) $\Rightarrow$ $\exists x, y$ such that unlock$_i[x] <$ lock$_j[x]$ and unlock$_j[y] <$ lock$_k[y]$
  - Since lock$_j[x] <$ unlock$_j$[y], we have unlock$_i[x] <$ lock$_k[y]$. By induction, for any path $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ in SG(S), T_1 performs an unlock before T_n has acquired certain lock.
  - Now, if SG(S) has a cycle: $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, we have T_1 performs an unlock before T_1 has acquired certain lock. A violation of the 2-phase rule!!

Two phase locking – Serializability

- Lock-point: the point where the transaction obtains all the locks
- With 2PL, a schedule is conflict equivalent to a a serial schedule ordered by the lock-point of the transactions

2-phase locking -- example

1. S-lock(X)
2. A1 <- Read(X)
3. A1 <- A1 – k
4. X-lock(X)

5. Write(X, A1)
6. S-lock(Y)
7. A2 <- Read(Y)
8. A2 <- A2 + k
9. X-lock(Y)
10. Write(Y, A2)
11. Unlock(X)

Lock point for T1

12. Unlock(Y)

T1

1. S-lock(X)

T2 waits

1. S-lock(X)
2. A1 <- Read(X)
3. A1 <- A1* 1.01
4. X-lock(X)
5. Write(X, A1)
6. S-lock(Y)

T2 waits

6. S-lock(Y)
7. A2 <- Read(Y)
8. A2 <- A2 * 1.01
9. X-lock(Y)

Lock point for T2

10. Write(Y, A2)
11. Unlock(Y)
12. Unlock(X)

T2

2 phase locking -- recoverability

- Recall definitions for recoverability
  - Recoverability
  - Avoid cascade aborts
  - Strict
- If a schedule is 2PL, does it guarantee any of the above?

2 phase locking -- recoverability

- Consider the following schedule

1. X-lock(X)
2. A1 <- Read(X)
3. A1 <- A1 * 10
4. Write(X, A1)
5. Unlock(X)

1. X-lock(X)
2. A2 <- Read(X)
3. A2 <- A2 + 1
4. Write(X, A2)
5. Unlock(X)
6. Commit

6. Abort!

Not recoverable!

2 phase locking – recoverability

- What's the problem?
- There is a gap between releasing locks and the decision to commit/abort
- Other transactions can still access data written by a uncomitted transaction
- How to solve this problem?

2 phase locking – recoverability

- Strict 2-phase locking:
  - 2-phase locking
  - X-locks can only be released when the transaction commits
- Question: does strict 2-phase locking ensure
  - Recoverability?
  - Avoid cascade aborts?
  - Strict?

2 phase locking – recoverability

- Rigorous 2-phase locking:
  - 2-phase locking
  - Any lock can only be released when the transaction commits
- Ensure serializability
  - Moreover, serial schedule is ordered by the time the transaction commits

Lock conversion (upgrades)

- Consider the transaction on the right
- Eventually transaction will need an X-lock on X
- But obtaining it at the beginning seems a waste of effort
- $\Rightarrow$ lock conversion
- Obtain an S-lock on X first, then obtain an X-lock only at the end
- Notice that X-lock is not automatically granted
- Similarly, downgrades are allowed (X-lock -> S-lock)

S-lock(X)

1. Read(X)
2. Read(Y)
3. Read(Z)
4. …

X-lock(X)

1000 operations not involving X

1005.Write(X)

Lock conversion (upgrades)

- Why lock conversion?
  - Allow more concurrency
  - If the lock operations are generated automatically by CC, we have no information about whether a transaction that reads *x* will or will not write *x* later.
- Upgrades are consider equivalent as obtaining a new lock (must be in growing phase)
- Downgrades are consider equivalent as releasing a lock (must be in shrinking phase)

Deadlocks

- 2 phase locking is a blocking protocol (transaction has to wait if it cannot obtain a lock)
- Probability of a deadlock
- Example:

1. S-lock(X)
2. A1 <- Read(X)
3. A1 <- A1 – k

Can't proceed T2 has S-lock on X

4. X-lock(X)

1. S-lock(X)
2. A1 <- Read(X)
3. A1 <- A1* 1.01

4. X-lock(X)

Can't proceed T1 has S-lock on X

Deadlock!

Deadlocks

- How to deal with deadlocks?
  - Ostrich
    - Pretend nothing happens and wait for user to hit the <ctrl-alt-del> key
  - Timeout
    - Assume deadlock after there is no progress for some time, and do something about it
  - Detection and recovery
    - Wait until a deadlock occurs and do something about it
  - Avoidance
    - Wait until a deadlock can occur if certain operations is executed and do something about it
  - Prevention
    - Set up the system such that there is never a chance of deadlocks

Deadlocks -- timeout

- When transaction waits too long for a lock, it is aborted and restarted again from the beginning
- The timeout period should be:
  - long enough so that most transactions that are aborted are actually deadlocked;
  - short enough so that deadlocked transactions don’t wait too long for their deadlocks to be broken.

Deadlocks -- Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlocks -- Detection

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock. (but tricky to implement)
- Drawback:
  - maintaining wait-for-graph is not cheap
  - Detecting cycles is an overhead

Deadlocks -- Starvation

- Sometimes the same transaction may keep being aborted
- This leads to starvation (really no blocking, but one transaction never get progressed)
- Solution: incorporate number of times being rolled-back as part of the cost

Deadlocks -- Avoidance

- Allow normal operation, but when there is a chance that deadlock will occur, then do something about it
- Recall that deadlock cannot occur if the wait-for-graph have no cycles.
- If we numbered each transaction (e.g. timestamp), then
  - If every edge in the wait-for graph points from $T_i \rightarrow T_j$ such that $i < j$, then deadlock never occurs
  - Why?
  - Works also if for every edge, $i > j$

Deadlocks -- Avoidance

- When a transaction T_i request a lock on an object, but T_j currently have the lock
- Two options:
  - Wait-die (non pre-emptive)
    - If i < j then wait
    - else T_i aborts
  - Wound-wait (pre-emptive)
    - If i > j then wait
    - Else T_j aborts
- If i and j represent time (i.e. small number = older transactions)
  - Wait-die: older transactions wait for younger transactions;
  - Wound-wait: younger transaction wait for older transactions

Deadlocks -- Avoidance

- How to avoid starvation?
- Note that in both scheme, at any given time, at least one transaction is never going to be aborted
  - The one with the smallest i
- Thus, to avoid starvation, ensure that when a transaction is aborted, it is restarted with the SAME i
  - Eventually it will becomes the one with the smallest i

Deadlock -- prevention

- Select a scheme such that deadlock will never occurs
- Conservative 2PL
  - 2 phase locking
  - A transaction must request ALL the locks in the beginning, and either all the locks or none of the locks are granted
- Labelling database objects
  - Transaction can only request locks in order of the database objects

Locks -- implementation

- Various support need to implement locking
  - OS support – lock(X) must be an atomic operation in the OS level
    - i.e. support for critical sections
  - Implementation of read(X)/write(X) – automatically add code for locking
  - Lock manager – module to handle and keep track of locks

Locks -- implementation

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

if T_i has a lock on *D*
then
read(*D*)
else
begin
if necessary wait until no other
transaction has a **X-lock** on *D* (or apply any deadlock avoidance rules)
grant T_i a **S-lock** on *D*;
read(*D*)
end

Locks -- implementation

- **write***(D)* is processed as:

  if T_i has a **X-lock** on *D*

  then

  write(*D*)

  else

  begin

  if necessary wait until no other trans. has any lock on *D* (or apply any deadlock avoidance rules),

  if T_i has a **S-lock** on *D*

  then

  upgrade lock on *D* to **X-lock**

  else

  grant T_i a **X-lock** on *D*

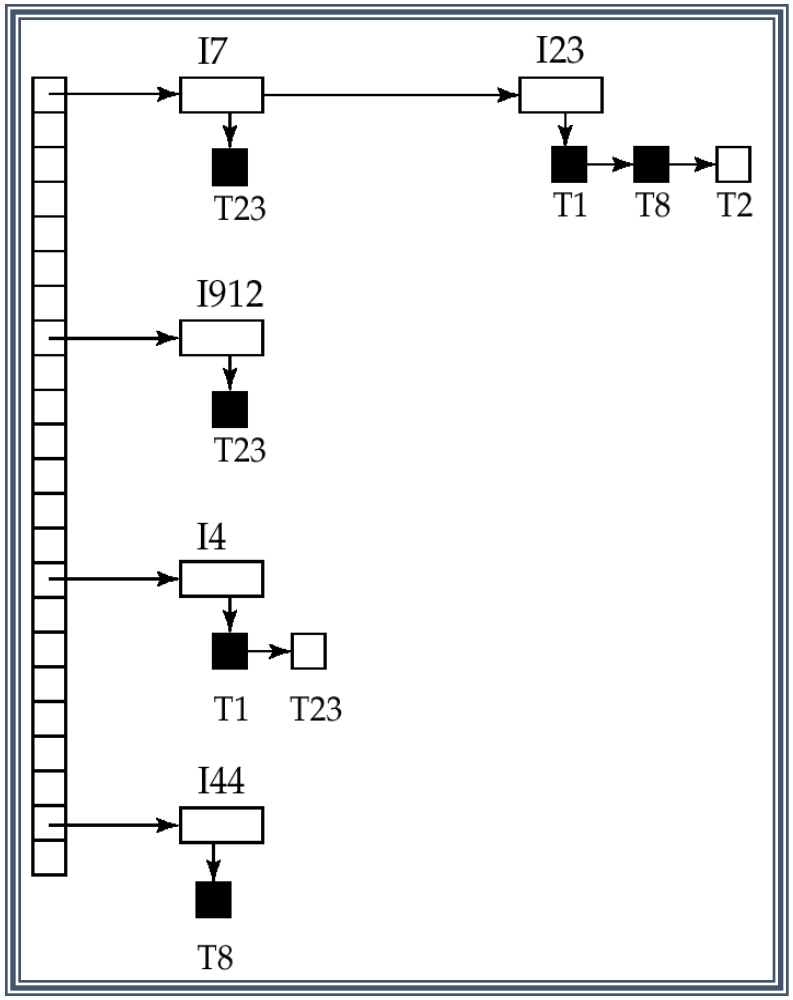
  write(*D*)

  end;
- All locks are released after commit or abort

Locks – Lock manager

- A **Lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Locks – lock manager



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Insertion & Deletion

- Does delete(X) conflict with read(X)/write(X)?
  - Yes. Wrong order leads to errors
- Thus delete(X) should be treated like a write operation
  - Request X-locks
- Similar to insert(X) operation.
  - X-lock is given to the newly created tuple.

The phantom menace

- Consider the following 2 transactions

1. Select sum(salary)
   From faculty
   Where dept = "CS"
2. Select sum(salary)
   From faculty
   Where dept = "Math"

T1

a. Insert into faculty
   values ("Lin", "CS", 1000)
b. Insert into faculty
   values ("Lam", "Math", 5000)

T2

- There does not seems to be a conflict (in terms of tuple)
- Assume initially CS faculty have total salary 1,000,000 and Math faculty have total salary 2,000,000
- Then T1 -> T2 will imply the select statements return 1,000,000 and 2,000,000
- T2 -> T1 will imply the select statements return 1,001,000 and 2,005,000

The phantom menace

- But consider the following schedule

a. Insert into faculty
values (“Lin”, “CS”, 1000)

1. Select sum(salary)
   From faculty
   Where dept = “CS”
2. Select sum(salary)
   From faculty
   Where dept = “Math”

b. Insert into faculty
values (“Lam”, “Math”, 5000)

T1

T2

- The output will be 1,000,100 and 2,000,000
- Not conflict serializable!

The phantom problem

- This is known as *the phantom problem*
- Why does it occur?
  - No tuples are in conflict
  - However, conflict occurs for tuples that satisfy a certain condition (dept = “CS”, dept = “Math”)
    - T1 require access for ALL tuples satisfying the condition
    - However, T2 changes the number of tuples satisfying the condition
- No quick solution: index-locking as a possibility (next lecture)