Concurrency Control Technique

Two-Phase Locking Techniques for Concurrency Control

- Lock
  - Variable associated with a data item describing status for operations that can be applied
  - One lock for each item in the database
- Binary locks
  - Two states (values)
    - Locked (1)
      - Item cannot be accessed
    - Unlocked (0)
      - Item can be accessed when requested

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Transaction requests access by issuing a lock_item(X) operation

```
lock_item(X):
B:  if LOCK(X) = 0                (*item is unlocked*)
        then LOCK(X) ←1           (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                  (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Lock and unlock operations for binary locks

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock table specifies items that have locks
- Lock manager subsystem
  - Keeps track of and controls access to locks
  - Rules enforced by lock manager module
- At most one transaction can hold the lock on an item at a given time
- Binary locking too restrictive for database items

Binary Locking Rules

1. A transaction *T* must issue the operation lock_item(*X*) before any read_item(*X*) or write_item(*X*) operations are performed in *T.*
2. A transaction *T* must issue the operation unlock_item(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T.*
3. A transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*
4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Shared/exclusive or read/write locks
  - Read operations on the same item are not conflicting
  - Must have exclusive lock to write
  - Three locking operations
    - read_lock(X)/share-locked
    - write_lock(X)/exclusive-locked
    - unlock(X)

Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) –1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

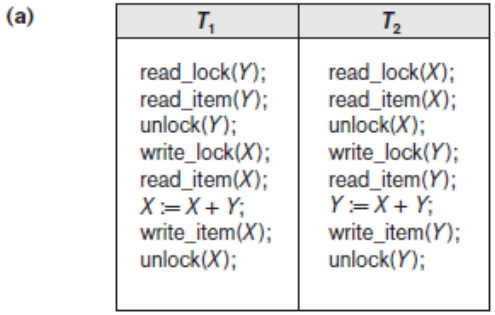
Rules for Shared/exclusive or read/write locks

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T.*
2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T.*
3. A transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T.*
4. A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. This rule may be relaxed for downgrading of locks
5. A transaction *T* will not issue a write_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*. This rule may also be relaxed for upgrading of locks
6. A transaction *T* will not issue an unlock(*X*) operation unless t already holds a read (shared) lock or a write (exclusive) lock on item *X*.

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock conversion
  - Transaction that already holds a lock allowed to convert the lock from one state to another
- Upgrading
  - Issue a read_lock operation then a write_lock operation
- Downgrading
  - Issue a read_lock operation after a write_lock operation

Transactions that do not obey two-phase locking (a) Two transactions *T*1 and *T*2 (b) Results of possible serial schedules of *T*1 and *T*2 (c) A nonserializable schedule *S* that uses locks



(a)

T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);
write_lock(*X*);
read_item(*X*);
X := *X* + *Y*;
write_item(*X*);
unlock(*X*);
	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	Y := *X* + *Y*;
	write_item(*Y*);
	unlock(*Y*);

(b) Initial values: *X*=20, *Y*=30

Result serial schedule T_1 followed by T_2: *X*=50, *Y*=80

Result of serial schedule T_2 followed by T_1: *X*=70, *Y*=50

(c)

T_1	T_2

read_lock(*Y*);
read_item(*Y*);
unlock(*Y*);	

	read_lock(*X*);
	read_item(*X*);
	unlock(*X*);
	write_lock(*Y*);
	read_item(*Y*);
	Y := *X* + *Y*;
	write_item(*Y*);
	unlock(*Y*);

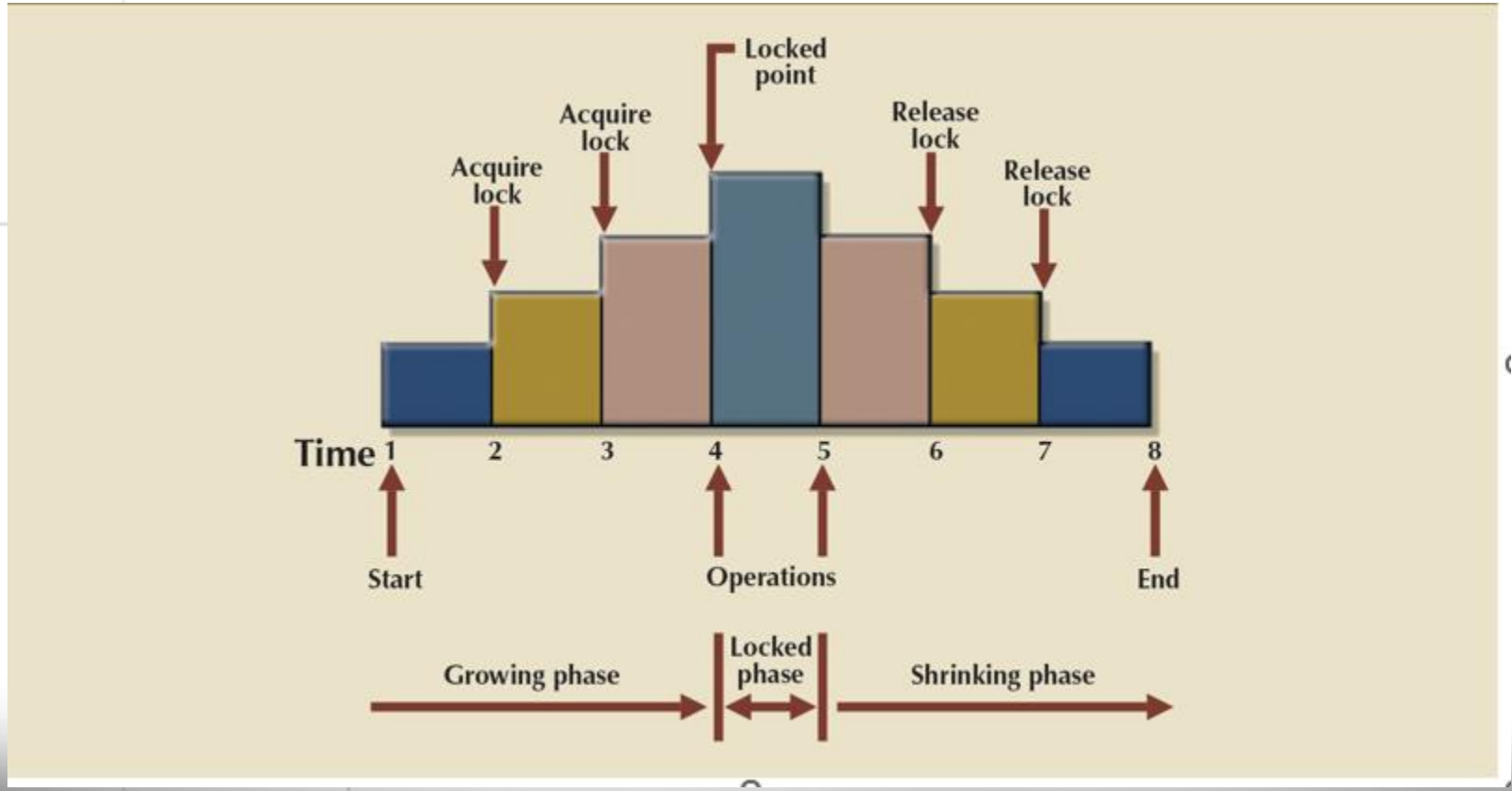
write_lock(*X*);
read_item(*X*);
X := *X* + *Y*;
write_item(*X*);
unlock(*X*);	

Time

Guaranteeing Serializability by Two-Phase Locking

- Two-phase locking protocol
  - All locking operations precede the first unlock operation in the transaction
  - Phases
    - Expanding (growing) phase
      - New locks can be acquired but none can be released
      - Lock conversion upgrades must be done during this phase
    - Shrinking phase
      - Existing locks can be released but none can be acquired
      - Downgrades must be done during this phase

Guaranteeing Serializability by Two-Phase Locking



T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Guaranteeing Serializability by Two-Phase Locking

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

Variations of Two-Phase Locking

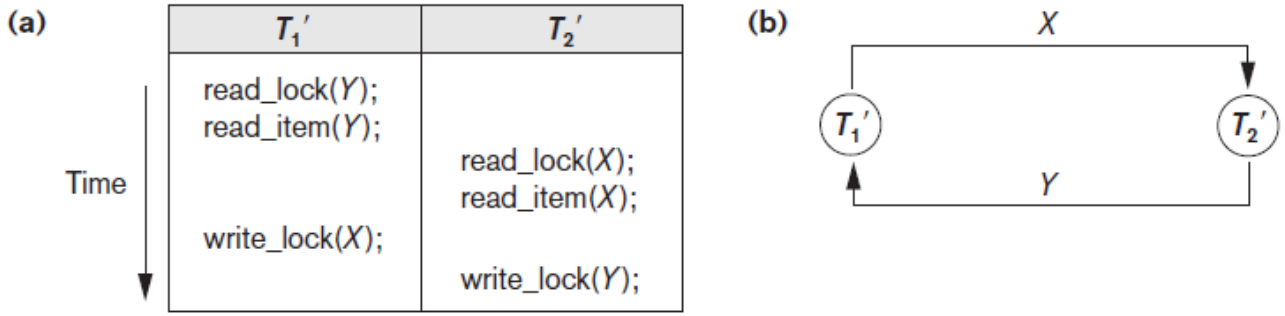
- Basic 2PL
- Conservative (static) 2PL
  - Requires a transaction to lock all the items it accesses before the transaction begins
    - Predeclare read-set and write-set
  - Deadlock-free protocol
- Strict 2PL
  - Transaction does not release exclusive locks until after it commits or aborts
- Rigorous 2PL
  - Transaction does not release any locks until after it commits or aborts

Variations of Two-Phase Locking (cont'd.)

Strict 2PL	Rigorous 2PL
Holds write-lock until it commits	Holds read and write lock
Must lock all of it items before it start, it is in shrinking phase	Does not unlock any of its item until after it terminates, so expanding state

Dealing with Deadlock and Starvation

- Deadlock
  - Occurs when each transaction T in a set is waiting for some item locked by some other transaction *T'*
  - Both transactions stuck in a waiting queue



Illustrating the deadlock problem (a) A partial schedule of *T*1' and *T*2' that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

How a Deadlock Condition is Created

HOW A DEADLOCK CONDITION IS CREATED				
TIME	TRANSACTION	REPLY	LOCK STATUS	
			DATA X	DATA Y
0			Unlocked	Unlocked
1	T1:LOCK(X)	OK	Locked	Unlocked
2	T2:LOCK(Y)	OK	Locked	Locked
3	T1:LOCK(Y)	WAIT	Locked	Locked
4	T2:LOCK(X)	WAIT	Locked	Locked
5	T1:LOCK(Y)	WAIT	Locked	Locked
6	T2:LOCK(X)	WAIT	Locked	Locked
7	T1:LOCK(Y)	WAIT	Locked	Locked
8	T2:LOCK(X)	WAIT	Locked	Locked
9	T1:LOCK(Y)	WAIT	Locked	Locked
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....

Dealing with Deadlock and Starvation (Deadlock prevention protocols)

- Deadlock prevention protocols
  - Every transaction locks all items it needs in advance (conservative 2PL)
  - Ordering all items in the database
    - Transaction that needs several items will lock them in that order
  - Both approaches impractical
- Protocols based on a timestamp
  - Some of these techniques use the concept of **transaction timestamp** TS(*T'*), which is a unique identifier assigned to each transaction.
    - Wait-die
    - Wound-wait

Timestamp method (Wait and wound)

- **Wait-die.** If Ts(T_i) < Ts(T_j), then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp.*
- **Wound-wait.** If Ts(T_i) < Ts(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

Aspect	Wait-Die	Wound-Wait
Priority	Older transactions are prioritized.	Older transactions are prioritized.
Action for Older Transactions	Allowed to wait if blocked by younger.	Aborts younger transactions (wounds them).
Action for Younger Transactions	Aborted if blocked by older.	Must wait if blocked by older.
Deadlock Prevention	By restricting waiting to older transactions.	By preventing older transactions from waiting.
Behavior	Conservative: Older transactions wait.	Aggressive: Older transactions abort younger ones.

WAIT/DIE AND WOUND/WAIT CONCURRENCY CONTROL SCHEMES

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. • T2 is rescheduled using the same timestamp.
T2 (19562545)	T1 (11548789)	• T2 dies (rolls back). • T2 is rescheduled using the same times tamp.	• T2 waits until T1 is completed and T1 releases its locks.

Dealing with Deadlock and Starvation (Deadlock prevention protocols)

- No waiting algorithm
  - If transaction unable to obtain a lock, immediately aborted and restarted later.
  - However, this scheme can cause transactions to abort and restart needlessly.
- Cautious waiting algorithm
  - Deadlock-free
  - Reduce the number of needless aborts/restarts.
  - If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

Dealing with Deadlock and Starvation (Deadlock detection)

- Deadlock detection
  - System checks to see if a state of deadlock exists
  - Wait-for graph
- Victim selection
  - Deciding which transaction to abort in case of deadlock
  - Generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).
- Timeouts
  - If system waits longer than a predefined time, it aborts the transaction
- Starvation
  - Occurs if a transaction cannot proceed for an indefinite period of time while other transactions continue normally
  - Solution: first-come-first-served queue

Concurrency Control Based on Timestamp Ordering

- Timestamp
  - Unique identifier assigned by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction start time
- Concurrency control techniques based on timestamps do not use locks
  - Deadlocks cannot occur

Concurrency Control Based on Timestamp Ordering

- Enforce the equivalent serial order on the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**.

1. **read_TS(*X*).** The **read timestamp** of item *X* is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.
2. **write_TS(*X*).** The **write timestamp** of item *X* is the largest of all the timestamps of transactions that have successfully written item *X*—that is, write_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully. Based on the algorithm, *T* will also be the last transaction to write item *X*,

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock
- General approach
  - Enforce equivalent serial order on the transactions based on their timestamps

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Timestamp ordering (TO)
  - Allows interleaving of transaction operations
  - Must ensure timestamp order is followed for each pair of conflicting operations
- Each database item assigned two timestamp values
  - read_TS(X)
  - write_TS(X)

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Basic TO algorithm
  - If conflicting operations detected, later operation rejected by aborting transaction that issued it
  - Schedules produced guaranteed to be conflict serializable
  - Starvation may occur
- Strict TO algorithm
  - Ensures schedules are both strict and conflict serializable
  - A transaction *T* issues a read_item(*X*) or write_item(*X*) such that TS(*T*) > write_TS(*X*) has its read or write operation *delayed* until the transaction *T'* that *wrote* the value of *X* (hence TS(*T'*) = write_TS(*X*)) has committed or aborted.

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Thomas's write rule
  - Modification of basic TO algorithm
  - Does not enforce conflict serializability
  - Rejects fewer write operations by modifying checks for write_item(X) operation as follows:
    1. If read_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation.
    2. If write_TS(*X*) > TS(*T*), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(*T*)—and hence after *T* in the timestamp ordering—has already written the value of *X*. Thus, we must ignore the write_item(*X*) operation of *T* because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
    3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).