LOCKING PROTOCOLS

A lock is a variable associated with a data item that describe the statues of the item with respect to possible operations that can be applied to it. Locking is an operation which secures

(a) Permission to Read

(b) Permission to Write a data item for a transaction.

Example:

Lock (X). Data item X is locked in behalf of the requesting transaction. Unlocking is an operation which removes these permissions from the data item. Example:

Unlock (X): Data item X is made available to all other transactions. Lock and Unlock are Atomic operations.

	Read	Write
Read	Y	N
Write	N	N

Lock Manager:

- Managing locks on data items.

Lock table:

- Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list

Types of lock

- Binary lock
- Read/write(shared / Exclusive) lock

Binary lock –

It can have two states (or) values 0 and 1.

0 – unlocked

1 - locked

- Lock value is 0 then the data item can accessed when requested.
- When the lock value is 1, the item cannot be accessed when requested.

Binary Lock

Lock_item(x)

```
B : if lock(x) = 0 ( * item is unlocked * )
then lock(x)            //1
else begin
        wait ( until lock(x) = 0 )
        goto  B;
end;
```

Unlock_item(x)

```
B : if lock(x)=1 ( * item is locked * )
then unlock(x)                        \\ 0
else
        printf (_ already is unlocked _)
        goto  B;
end;
```

Read / write(shared/exclusive) lock

Read_lock

- Its also called shared-mode lock
- If a transaction Ti has obtain a shared-mode lock on item X, then Ti can read, but cannot write , X.
- Outer transactions are also allowed to read the data item but cannot write.

Read_lock(x)

```
B : if lock(x) = "unlocked"  then                    (1)
begin
    lock(x)            //"read_locked" )
    read(x)            //1
else if
    lock(x) = "read_locked"  then                    (2)
    read(x)                //no_of_read(x) +1
else begin
    wait (until lock(x) = "unlocked")
    goto  B;
    end;
```

Write_lock(x)

```
B : if lock(x) = "unlocked" then                                  (1)
begin
        lock(x)         //"write_locked"
else if
        lock(x) = "write_locked"                                  (2)
        wait ( until lock(x) = "unlocked")
else begin
        lock(x)="read_locked" then                            (3)
        wait ( until lock(x) = —unlocked  )
end;
```

Unlock(x)

```
If lock(x) = "write_locked" then
begin
        unlock(x)               \\"unlocked"
else if
        lock(x) = "read_locked" then
begin
        read(x)                 \\no_of_read(x) - 1
        if ( no_of_read(x) = 0 ) then
begin
        unlock(x)               \\"unlocked"
end
```

TWO PHASE LOCKING PROTOCOL

This protocol requires that each transaction issue lock and unlock request in two phases

- Growing phase
- Shrinking phase

Growing phase

During this phase new locks can be occurred but none can be released

Shrinking phase

During which existing locks can be released and no new locks can be occurred

Let's see a transaction implementing 2-PL.

	T_1	**T_2**
1	**lock-S(A)**	
2		**lock-S(A)**
3	**lock-X(B)**	
4	**…….**	**……**
5	**Unlock(A)**	
6		**Lock-X(C)**
7	**Unlock(B)**	
8		**Unlock(A)**
9		**Unlock(C)**
10	**…….**	

This is just a skeleton transaction which shows how unlocking and locking works with 2-PL. Note for:

Transaction T_1:

- Growing Phase is from steps 1-3.
- Shrinking Phase is from steps 5-7.
- Lock Point at 3

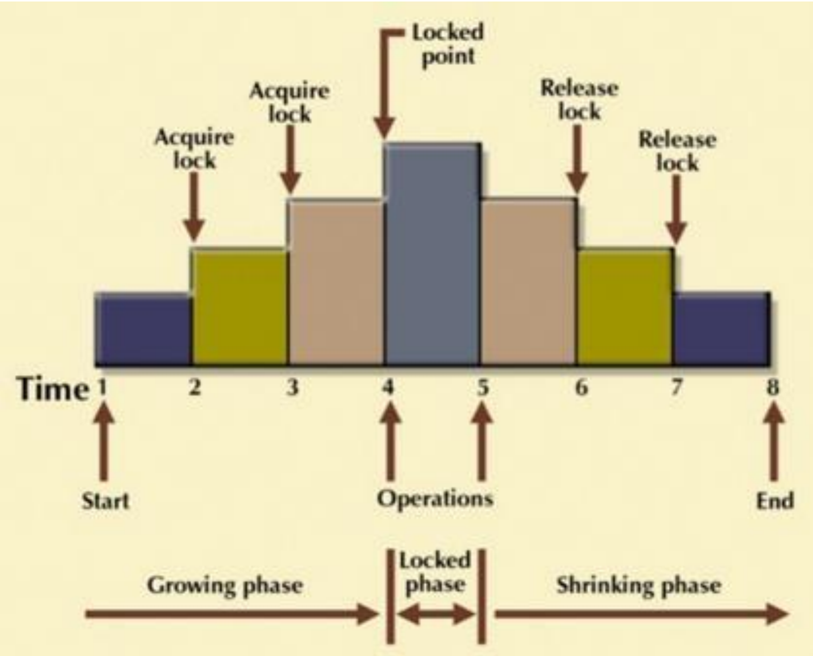
Transaction T_2:

- Growing Phase is from steps 2-6.
- Shrinking Phase is from steps 8-9.
- Lock Point at 6

What is **LOCK POINT** ? The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work.

Types of two phase protocol

- Strict two phase locking protocol
- Rigorous two phase locking protocol



Strict two phase locking protocol

This protocol requires not only that locking be two phase, but also all exclusive locks taken by a transaction be held until that transaction commits

Rigorous two phase locking protocol

This protocol requires that all locks be held until all transaction commits.

Consider the two transaction T1 and T2

T1 :

```
        read(a1);
        read(a2);
        …….
        read(an);
    write(a1);
```

T2:

```
    read(a1);
    read(a2);
```

display(a1+a1);

Lock conversion

- Lock Upgrade
- Lock Downgrade

Lock upgrade:

- ➢ Conversion of existing read lock to write lock
- ➢ Take place in only the growing phase

if Ti has a read-lock (X) and Tj has no read-lock (X) (i != j)

then convert read-lock (X) to write-lock (X)

else

force Ti to wait until Tj unlocks X

Lock downgrade:

- ➢ Conversion of existing write lock to read lock
- ➢ Take place in only the shrinking phase

Ti has a write-lock (X) (*no transaction can have any lock on X*)

convert write-lock (X) to read-lock (X)

T_1	T_2
Lock-S(a_1)	
	Lock-S(a_1)
Lock-S(a_2)	
	Lock-S(a_1)
Lock-S(a_3)	
Lock-S(a_4)	
	Unlock(a_1)
	Unlock(a_2)
Lock-S(a_1)	
Upgrade(a_1)	

Log

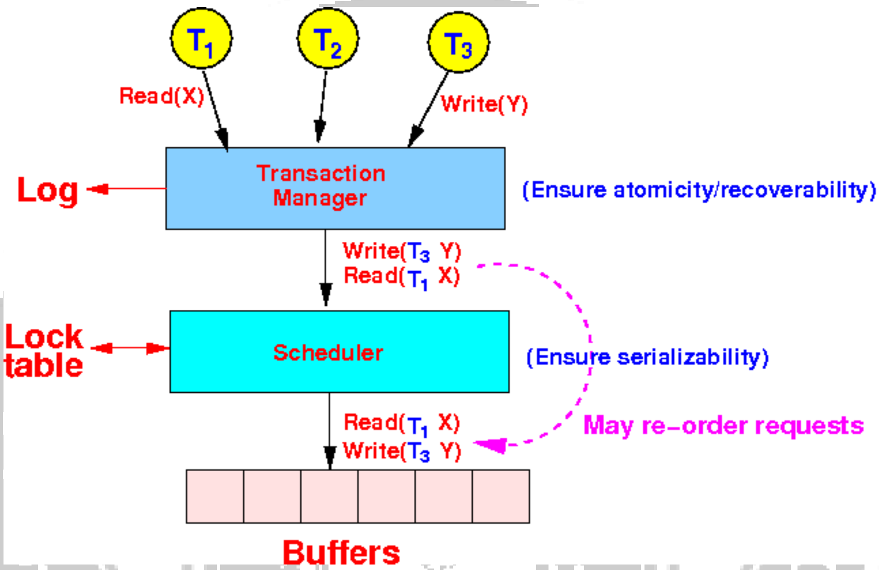
- ➢ Log is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures.
- ➢ Physically, a log is a file of updates done to the database, stored in stable storage.

Log rule

– A log records for a given database update must be physically written to the log, before the update physically written to the database.

– All other log record for a given transaction must be physically written to the log, before the commit log record for the transaction is physically written to the log.

– Commit processing for a given transaction must not complete until the commit log record for the transaction is physically written to the log.



System log

– [Begin transaction ,T]

– [write_item , T, X , oldvalue,newvalue]

– [read_item,T,X]

– [commit,T]

– [abort,T]

- Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let *T* be a transaction initiated at site *Si,* and let the transaction coordinator at *Si* be *Ci*

Phase 1: Obtaining a Decision (prepare)

- Coordinator asks all participants to *prepare* to commit transaction *Ti*.

  - Ci adds the records <prepare *T*> to the log and forces log to stable storage
  - sends prepare *T* messages to all sites at which *T* executed

- Upon receiving message, transaction manager at site determines if it can commit the transaction
    - if not, add a record <no *T*> to the log and send abort *T* message to *Ci*
    - if the transaction can be committed, then:
    - add the record <ready *T*> to the log
    - force *all records* for *T* to stable storage
    - send ready *T* message to C*i*

Phase 2: Recording the Decision (commit)

- *T* can be committed of *Ci* received a ready *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <commit *T*> or <abort *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site *Si* recovers, it examines its log to determine the fate of transactions active at the time of the failure.

• Log contain <commit *T*> record: site executes redo (*T*)

• Log contains <abort *T*> record: site executes undo (*T*)

• Log contains <ready *T*> record: site must consult C*i* to determine the fate of *T*.

– If *T* committed, redo (*T*)

– If *T* aborted, undo (*T*)

• The log contains no control records concerning *T* replies that Sk failed before responding to the prepare *T* message from Ci

  – since the failure of *Sk* precludes the sending of such a response *Ci* must abort *T*

  – *Sk* must execute undo (*T*)

Handling of Failures- Coordinator Failure

If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:

1. If an active site contains a <commit *T*> record in its log, then *T* must be committed.

2. If an active site contains an <abort *T*> record in its log, then *T* must be aborted.

3. If some active participating site does not contain a <ready *T*> record in its log, then the failed coordinator *Ci* cannot have decided to commit *T*. Can therefore abort *T*.

4. If none of the above cases holds, then all active sites must have a <ready *T*> record in their logs, but no additional control records (such as <abort *T*> of <commit *T*>). In this case active sites must wait for *Ci* to recover, to find decision.

• Blocking problem : active sites may have to wait for failed coordinator to recover**.**

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
- No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
- Again, no harm results