Today:

- Conflict-serializability test, precedence graphs.
- The scheduler, a component of a DBMS that enforces serializability.
- Introduction to locking, a method to assure conflict serializability.
  - 2-phase locking.
  - Update locks.

Soon:

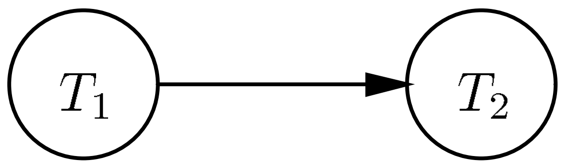
- Other locking schemes.
- View serializability: a more general condition assuring correctness.
- Non-locking ways to assure serializability: timestamps, validation.

Precedence Graphs

- Nodes = transactions.
- If $T_i <_S T_j$, put an arc $T_i \to T_j$.

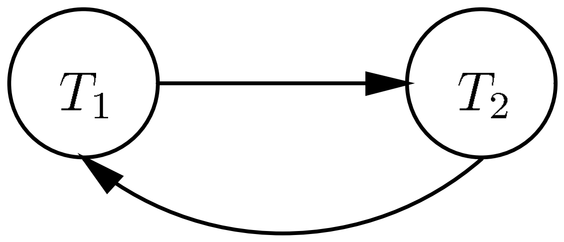
Example

$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$



Example

$r_1(A);\ r_2(B);\ w_1(A);\ r_2(A);\ w_2(B);\ w_2(A);\ r_1(B);\ w_1(B);$



If There is a Cycle?

Then there is no serial schedule conflict-equivalent to S.

- Why? Each arc represents a requirement on the order of transactions in a conflict-equivalent serial schedule.
  - A cycle puts too many requirements on any linear order of transactions.

If There is No cycle?

Then any topological order of the precedence graph for S suggests a conflict-equivalent schedule.

Proof

Induction on n, the number of transactions, that if the precedence graph is acyclic, we can swap actions to form a serial schedule consistent with any topological order.

- Basis: $n = 1$, already serial.
- Induction: let $T_1, T_2, \ldots, T_n$ be a topological order consistent with the precedence graph.
- There are no arcs into T_1, so we may swap all T_1-actions to the front.
  - Result is a schedule

    (T_1 actions)(other actions)

- The "other actions" have a precedence graph that is the same as the original, minus node T_1.
  - Thus, $T_2, \ldots, T_n$ is a topological order for this graph of $n - 1$ nodes.
- By induction, the other actions may be permutted to a serial schedule in which the order of transactions is $T_2, \ldots, T_n$.

Schedulers

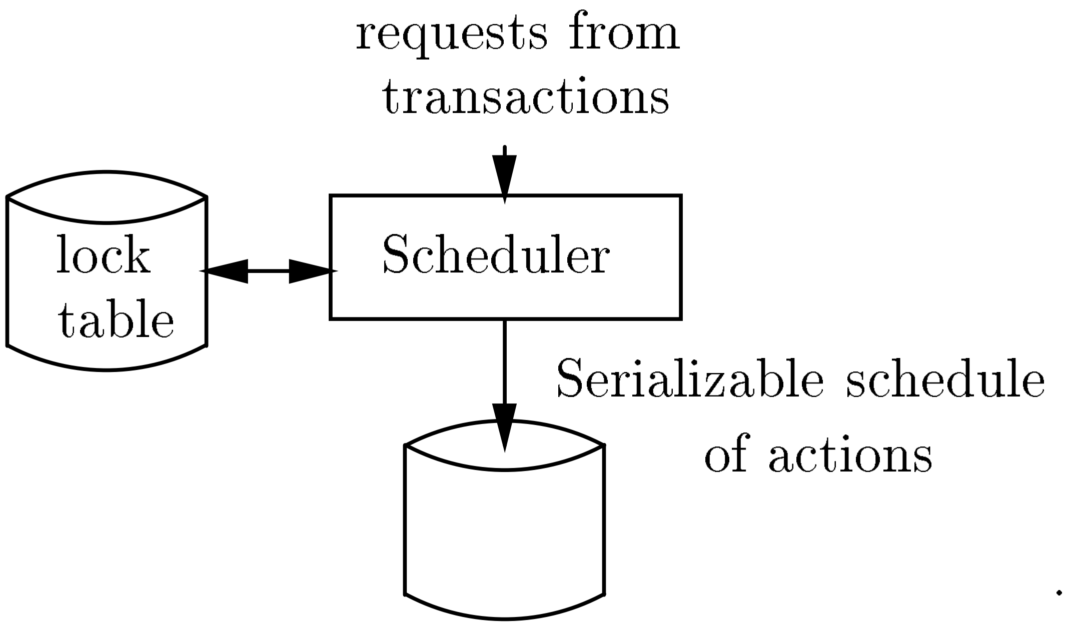
The *scheduler* sits between transactions and the storage system, deciding whether it is OK to forward a read or write request.

- Ideal: forward request only if consistency cannot be compromised.
- Real: forwarded requests cannot result in a violation of conflict-serializability.

Lock Actions

A transaction T_i requests a lock on a DB element X from the scheduler.

- The scheduler can either grant the lock to T_i or make T_i wait for the lock.
- If granted, T_i should eventually unlock (release) the lock on X.
- Shorthands:
  - $l_i(X)$ = "transaction T_i requests a lock on X."
  - $u_i(X)$ = "T_i releases the lock on X."


requests from transactions
lock table
Scheduler
Serializable schedule of actions

Consistency Conditions for Transactions

1. Read or write X only when you hold a lock.

   ❖ $r_i(X)$ or $w_i(X)$ must be preceded by some $l_i(X)$ with no intervening $u_i(X)$.

2. If you lock, you must eventually unlock.

   ❖ Every $l_i(X)$ must be followed by $u_i(X)$.

Legal Schedule

Locked means locked.

- Two transactions cannot hold a lock simultaneously on the same X.

  ❖ A schedule with $l_i(X)$ cannot have another $l_j(X)$ until $u_i(X)$ appears.

Example

$T_1 = l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
$T_2 = l_2(A)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $u_2(A)$; $l_2(B)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $u_2(B)$;

- Legal schedule does not by iteslf assure consistency ($A = B$).

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			
`B := B+100;`			
$w_1(B)$; $u_1(B)$;			150

Two-Phase Locking

Forbid a transaction to request a lock after it has unlocked anything.

Example

$T_1 = l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
$T_2 = l_2(A)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $l_2(B)$; $u_2(A)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $u_2(B)$;

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
$r_1(B)$;	$l_2(B)$ Denied		
`B := B+100;`			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		250

But Deadlocks Still Possible

Example

$T_1 = l_1(A)$; $r_1(A)$; `A := A+100;` $w_1(A)$; $l_1(B)$;
$u_1(A)$; $r_1(B)$; `B := B+100;` $w_1(B)$; $u_1(B)$;
$T_2 = l_2(B)$; $r_2(B)$; `B := B*2;` $w_2(B)$; $l_2(A)$;
$u_2(B)$; $r_2(A)$; `A := A*2;` $w_2(A)$; $u_2(A)$;

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100;`			
	`B := B*2;`		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$ `Denied`	$l_2(A)$ `Denied`		

- We'll return to deadlocks, but key idea: system must abort and restart one or more transactions.
  - Yet another reason we must be careful how we manage log and DB updates.

Why 2PL Works

Precisely: a legal schedule of 2PL transactions is conflict-serializable.

- Proof is an induction on n, the number of transactions.
- Remember, conflicts involve only read/write actions, not locks, but the legality of the transaction requires that the r/w's be consistent with the l/u's.

Basis

$n = 1$; nothing to do.

Induction

Find the first transaction, say T_i, to perform an unlock action, say $u_i(X)$.

- Must show we can permute all the r/w actions of T_i in front of the other transactions.
- Consider some action such as $w_i(Y)$. Can it be preceded by some conflicting action $w_j(Y)$ or $r_j(Y)$?
    - If so, then $u_j(Y)$ must intervene, as $w_j(Y) \cdots u_j(Y) \cdots l_i(Y) \cdots w_i(Y)$.
    - Since T_i is first to unlock, $u_i(X)$ appears before $u_j(Y)$.
    - But then $l_i(Y)$ appears after $u_i(X)$, contradicting 2PL.
- Conclusion: $w_i(Y)$ can slide forward in the schedule without conflict.
    - Similar argument for a $r_i(Y)$ action.

Shared/Exclusive Locks

Problem: while simple locks + 2PL guarantee conflict-serializability, they do not allow two readers of DB element X at the same time.

- But multiple readers is not a problem for conflict-serializability (read actions commute).
- Solution: Two kinds of locks:
  1. *Shared lock* ("read lock") $sl_i(X)$ allows T_i to read, but not write X. It must prevent other transactions from writing X without preventing them from reading X.
  2. *Exclusive lock* ("write lock") $xl_i(X)$ allows T_i to read and/or write X; no other transaction may read or write X.

Consistency Conditions for Transactions

A read action $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$.

2PL Condition

A transaction must not have a $sl_i(X)$ or $xl_i(X)$ after any $u_i(Y)$.

Legal Schedules

- No two exclusive locks.
  - ❖ If $xl_i(X)$ appears in a schedule, then there cannot be a $xl_j(X)$ until after a $u_i(X)$ appears.
- No exclusive and shared locks.
  - ❖ If $xl_i(X)$ appears, there can be no $sl(j(X)$ until after $u_i(X)$.
  - ❖ If $sl_i(X)$ appears, there can be no $wl(j(X)$ until after $u_i(X)$.

Scheduler Rules

When there is more than one kind of lock, the scheduler needs a rule that says "if there is already a lock of type T on DB element X, can I grant a lock of type S on X?"

- The *compatibility matrix* answers the question in row T, column S.

Compatibility Matrix for Shared/Exclusive Locks

	S	X
S	yes	no
X	no	no

Example

T_1: $sl_1(A); r_1(A); xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B);$

T_2: $sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B);$

T_1	T_2
$sl_1(A);\ r_1(A);$	
	$sl_2(A);\ r_2(A);$
	$sl_2(B);\ r_2(B);$
$xl_1(B)$ Denied	
	$u_2(A);\ u_2(B)$
$xl_1(B);\ r_1(B);\ w_1(B);$	
$u_1(A);\ u_2(B);$	

- Notice that in the conflict-equivalent serial schedule, T_2 precedes T_1, even though T_1 started first.

Upgrading Locks

Instead of taking an exclusive lock immediately, a transaction can take a shared lock, read, and then *upgrade* to exclusive so it can write.

Example

T_1: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$ Denied	
	$u_2(A)$; $u_2(B)$
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_2(B)$;	

- Allows T_1 to compute with A and B while waiting for the exclusive lock on B.

Update Locks

Problem: when we allow upgrades, it is easy to get into a deadlock situation.

- Example: T_1 and T_2 each read X and later write X.

T_1	T_2
$sl_1(X)$	
	$sl_2(X)$
$xl_1(X)$ Denied	
	$xl_2(X)$ Denied

- Solution: *update* lock $ul_i(X)$ with assymetric compatibility matrix.
    - ❖ Only an update (not read) can be upgraded to write.
    - ❖ Legal schedules: read action permitted when there is either a shared or update lock.
    - ❖ An update can be granted while there is a shared-lock, but the scheduler will not grant a shared-lock when there is an update.

Compatability Matrix for Shared, Exclusive, and Update Locks

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

- Example: T_1 and T_2 each read X and later write X.

T_1	T_2
$ul_1(X);\ r_1(X);$	
	$ul_2(X)$ Denied
$xl_1(X);\ w_1(X);\ u_1(X);$	
	$ul_2(X);\ r_2(X);$
	$xl_2(X);\ w_2(X);\ u_2(X);$