Transaction Processing and Concurrency Control

- It is often the case that a database system will be accessed by many users simultaneously.

- If this access is read-only, then there are no serious integrity problems; only ones of performance.

- If the access includes writing the database, then serious problems will arise if the interaction is not regulated.

Example: Simplified bank transactions: Suppose that we have two users u_1 and u_2, who issue transactions:

- T_1: Compound 10% on account 15.
- T_2: Withdraw 2000 from account 15.

- Let R_i and W_i be local variables for transaction T_i.

- Read_Bal (Y) means read the balance of account Y into a local variable R_i for that transaction:

  $R_i \leftarrow \mathrm{Bal}(Y)$

- Write_Bal(Y) means write the value of the local variable W_I to the balance record of account Y.

  $\mathrm{Bal}(Y) \leftarrow W_i$

- Compound(X) means add X% interest to the local account variable.

  $W_i \leftarrow R_i \times ((100 + X) / 100)$

- Withdraw(X) means subtract X dollars from local account variable W_i.

  $W_i \leftarrow R_i - X$

- Represent T_1 as:

```
Begin transaction
     Read_Bal (15);
     Compound(10);
     Write_Bal(15);
End transaction
```

- Represent T_2 as:

```
Begin transaction
   Read_Bal(15);
   Withdraw(2000);
   Write_Bal(15);
End transaction
```

Both of the following *schedules* work just fine:

T_1:	T_2:	Bal(15):
Read_Bal (15): $R_1 \leftarrow$ BAL(15)		10000
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		10000
Write_Bal(15): Bal(15) $\leftarrow W_1$		11000
	Read_Bal (15): $R_2 \leftarrow$ Bal(15)	11000
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	11000
	Write_Bal(15): Bal(15) $\leftarrow W_2$	9000

T_1:	T_2:	Bal(15):
	Read_Bal (15): $R_2 \leftarrow$ Bal(15)	10000
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	10000
	Write_Bal(15): Bal(15) $\leftarrow W_2$	8000
Read_Bal (15): $R_1 \leftarrow$ BAL(15)		8000
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		8000
Write_Bal(15): Bal(15) $\leftarrow W_1$		8800

Notice that they do not produce the same result, though!

The Lost Update Problem:

- Suppose that the two transactions now interleave their operations.
- It is then possible that the result of one transaction will overwrite that of the other.
- This is called a *lost update.*
- The following example illustrates.

T_1:	T_2:	Bal(15):
Read_Bal (15): $R_1 \leftarrow BAL(15)$		10000
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		10000
	Read_Bal (15): $R_2 \leftarrow Bal(15)$	10000
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	10000
	Write_Bal(15): $Bal(15) \leftarrow W_2$	8000
Write_Bal(15): $Bal(15) \leftarrow W_1$		11000

T_1:	T_2:	Bal(15):
	Read_Bal (15): $R_2 \leftarrow Bal(15)$	10000
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	10000
Read_Bal (15): $R_1 \leftarrow BAL(15)$		10000
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		10000
Write_Bal(15): $Bal(15) \leftarrow W_1$		11000
	Write_Bal(15): $Bal(15) \leftarrow W_2$	8000

The Dirty Read Problem:

- If a transaction proceeds partway, but aborts before completion, another transaction may make use of the discarded results.

- This is known as a *dirty read.*

Example:

Suppose that the following two transactions run concurrently:

- T_1: Add 10% to accounts 1, 2, and 3.
- T_2: Withdraw 2000 from account 2.

- Suppose further that T_1 *aborts* after compounding the interest on accounts 1 and 2, yet T_2 has used this information.

This is illustrated in the following.

T_1:	T_2:	Bal(1,2,3):
Read_Bal (1): $R_1 \leftarrow$ BAL(1)		10000 10000 10000
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		Unchanged
Write_Bal(1): Bal(1) $\leftarrow W_1$		11000 10000 10000
Read_Bal (2): $R_1 \leftarrow$ BAL(2)		Unchanged
Compound(10): $W_1 \leftarrow R_1 \times (110 / 100)$;		Unchanged
Write_Bal(2): Bal(2) $\leftarrow W_1$		11000 11000 10000
	Read_Bal (2): $R_2 \leftarrow$ Bal(2)	Unchanged
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	Unchanged
Abort and Restore!!		10000 10000 10000
	Write_Bal(2): Bal(2) $\leftarrow W_2$	10000 9000 10000

The Unrepeatable Read Problem:

- Another update problem occurs when a transaction writes its results in stages, and another transactions reads between these writes.

- This is called an *unrepeatable read*, because it happens only by chance ordering of the operations.

Example:

- T_1: Sums the balances of accounts 1, 2, and 3, and just writes this result to a display. This transaction performs no writes of the database.

- T_2: Transfers 2000 from account 1 to account 2. It does this by first performing a withdrawal on account 1, and then a deposit to account 2.

T_1:	T_2:	Bal(1,2,3):
	Read_Bal (1): $R_2 \leftarrow$ Bal(1)	10000 10000 10000
	Withdraw(2000): $W_2 \leftarrow R_2 - 2000$;	Unchanged
	Write_Bal(1): Bal(1) $\leftarrow W_2$	8000 10000 10000
Read_Bal (1): $R_1 \leftarrow$ BAL(1)		Unchanged
Update Sum: $W_1 \leftarrow R_1$		Unchanged
Read_Bal (2): $R_1 \leftarrow$ BAL(2)		Unchanged
Update Sum: $W_1 \leftarrow W_1 + R_1$		Unchanged
Read_Bal (3): $R_1 \leftarrow$ BAL(3)		Unchanged
Update Sum: $W_1 \leftarrow W_1 + R_1$		Unchanged
Write_Sum: Write(W_1)		Unchanged
	Read_Bal (2): $R_2 \leftarrow$ Bal(2)	Unchanged
	Deposit(2000): $W_2 \leftarrow R_2 + 2000$;	Unchanged
	Write_Bal(2): Bal(2) $\leftarrow W_2$	8000 12000 10000

The sum displayed is 2000 low!

Schedules:

A schedule is a specification of the order in which the operations of a set of transactions are to be performed. Let us be more formal.

A *transaction* $T = \langle t_1, t_2, .., t_n \rangle$ is a finite sequence of steps, with each step t_i either a read action (denoted r(x)) or a write action (denoted w(x)). Here x is the database object which is read or written. It is usually assumed that a given object is read and written at most once in any transaction.

Example: T = r(x) w(x) r(y) is a transaction.

A schedule for a set of transactions is a specification of the order in which the steps will be executed. Formally, let $\mathbf{T} = \{T_1, T_2, .., T_m\}$ be a finite set of transactions, with

$$T_i = \langle t_{i1}, t_{i2}, .., t_{ini} \rangle.$$

A *schedule* S for $\mathbf{T} = \{T_1, T_2, .., T_m\}$ is any total ordering $\leq_S$ of the set

$$\{t_{ij} \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n_l\}$$

with the property that $t_{ij} \leq_S t_{ik}$ whenever $j \leq k$; (*i.e.* the order of elements within each T_l is preserved.)

A schedule S for $\mathbf{T} = \{T_1, T_2, .., T_m\}$ is *serial* if there is an ordering $\leq$ of $\mathbf{T}$ with the property that if $T_i \leq T_j$, then all elements of T_i occur before any element of T_j in S.

Example: Let $T_1 = r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$
$T_2 = r_2(z)\ w_2(z)\ w_2(y)$
$T_3 = r_3(z)\ w_3(z)\ r_3(x)\ w_3(x)$

Then

$r_2(z)\ w_2(z)\ w_2(y)\ r_3(z)\ w_3(z)\ r_3(x)\ w_3(x)\ r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

is the serial schedule corresponding to $T_2 < T_3 < T_1$, while

$r_1(x)\ r_1(y)\ r_3(z)\ w_3(z)\ r_2(z)\ w_1(x)\ w_1(y)\ w_2(z)\ w_2(y)\ r_3(x)\ w_3(x)$

is a non-serial schedule.

Serializability:

- A serial schedule is a "correct" one, in the sense that there is no undesirable intertwining of actions of different transactions.

- Allowing only serial schedules is very restrictive, and prohibits any sort of concurrency whatever.
  - Performance may be compromised greatly, particularly in systems with real-time human input.

- The solution is to allow *serializable* schedules; that is, ones which are operationally equivalent to serial schedules. In this fashion:
  - Parallelism is allowed.
  - The correctness of the transactions is not compromised.

- The obvious question is then, how one defines "serializable."

- It turns out that there are (at least) two reasonable definitions.

View Serializability: In view serializability, it is ensured that reads and subsequent writes occur in the same order as in some serial schedule.

Let **T** = $\{T_1, T_2, .., T_m\}$ be a set of transactions, and let S be a schedule for **T**. Let $r_i(x)$ occur in T_i and let $w_j(x)$ occur in T_j.

- It is said that $r_i(x)$ *reads from* $w_j(x)$ in S if $w_j(x) \leq_S r_i(x)$ and there is no $k \neq j$ for which $w_j(x) \leq_S w_k(x) \leq_S r_i(x)$.

- It is said that $r_i(x)$ is an *initial read* if there is there is no k for which $w_k(x) \leq_S r_i(x)$.

- It is said that $w_j(x)$ is a *final write* (*of* x) in S if there is no $k \neq j$ for which $w_j(x) \leq_S w_k(x)$.

Example: In

$r_1(x)\ r_1(y)\ r_3(z)\ w_3(z)\ r_2(z)\ w_1(x)\ w_1(y)\ w_2(z)\ w_2(y)\ r_3(x)\ w_3(x)$

- $r_2(z)$ reads from $w_3(z)$.
- $r_3(x)$ reads from $w_1(x)$.
- $r_1(x)$, $r_1(y)$, and $r_3(z)$ are initial reads.
- $w_2(z)$, $w_2(y)$, and $w_3(x)$ are final writes.

Let S and S′ be schedules for **T** = $\{T_1, T_2, .., T_m\}$. S and S′ are said to be *view equivalent*, written

$$S \approx_v S'$$

if the following conditions hold:

1. Every read action of the form $r_i(x)$ is either an initial read or else reads from the same write action $w_j(x)$ in both schedules.
2. Both schedules have the same final write steps.

Example: The following two schedules are view equivalent:

$r_1(x)\ r_1(y)\ r_3(z)\ w_3(z)\ r_2(z)\ w_1(x)\ w_1(y)\ w_2(z)\ w_2(y)\ r_3(x)\ w_3(x)$

$r_1(x)\ r_1(y)\ r_3(z)\ w_1(x)\ w_1(y)\ w_2(y)\ w_3(z)\ r_2(z)\ w_2(z)\ r_3(x)\ w_3(x)$

The following schedule is not view equivalent to either:

$r_1(x)\ r_1(y)\ r_3(z)\ w_3(z)\ r_3(x)\ r_2(z)\ w_1(x)\ w_1(y)\ w_2(z)\ w_2(y)\ w_3(x)$

A schedule S for a set **T** of transactions is said to be *view serializable* if there is a serial schedule S′ for **T** such that $S \approx_v S'$.

- The motivation for condition 1 above is quite clear.

- The motivation for 2 (same final writes) is less clear. Here is an example which will help to clarify.

  - $T_1 = w_1(x)\ w_1(y)$
  - $T_2 = w_2(x)\ w_2(y)$

- Note that, for any serial schedule, the first transaction has no effect.

- Note further that, since there are no reads, no schedule can violate condition 1.

- The following schedule is not equivalent to a serial schedule; it violates only condition 2. T_1 final-writes y, while T_2 final-writes x.

$$w_1(x)\ w_2(x)\ w_2(y)\ w_1(y)$$

Observation: If $S \approx_v S'$, then S and S' produce exactly the same updates. □

Complexity Problem: The problem of deciding whether or not an arbitrary schedule is view serializable is NP-complete. This means that the best known algorithm has exponential complexity in the worst case.

Question: Is there an alternative notion of serializability?

Conflict Serializability: In conflict serializability, it is ensured that “conflicting steps” occur in the same order.

Formally, let $\mathbf{T} = \{T_1, T_2, .., T_m\}$ be a set of transactions, and let S be a schedule for **T**. Two steps p and q from S are said to be *in conflict* if the following conditions hold:

- They are from distinct transactions.
- They operate on the same database object.
- At least one is a write.

Now let S and S′ each be schedules for T. They are said to be *conflict equivalent*, denoted

$$S \approx_c S'$$

If for each pair (p,q) of conflicting elements,

$$(p \leq_S q) \Leftrightarrow (p \leq_{S'} q)$$

In other words, conflicting elements occur in the same order in each schedule.

A schedule S for **T** is said to be *conflict serializable* if there is a serial schedule S′ for **T** with the property that $S \approx_c S'$.

Complexity: There is a simple test for conflict serializability.

For a schedule S, define the *conflict graph* to be the directed graph which has transactions as nodes, with an edge from T_i to T_j precisely in the case that a step p in T_i is in conflict with a step q in T_j, with p preceding q in S.

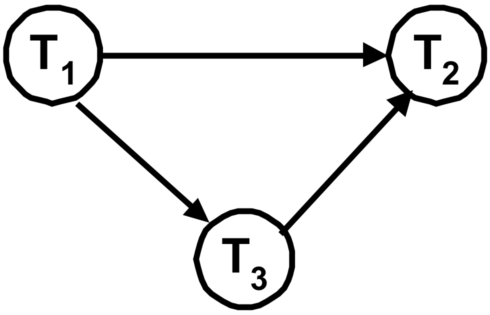
Example:

The conflict graph for both of the schedules

$r_1(x)\ r_1(y)\ r_3(z)\ w_3(z)\ r_2(z)\ w_1(x)\ w_1(y)\ w_2(z)\ w_2(y)\ r_3(x)\ w_3(x)$

$r_1(x)\ r_1(y)\ r_3(z)\ w_1(x)\ w_1(y)\ w_2(y)\ w_3(z)\ r_2(z)\ w_2(z)\ r_3(x)\ w_3(x)$

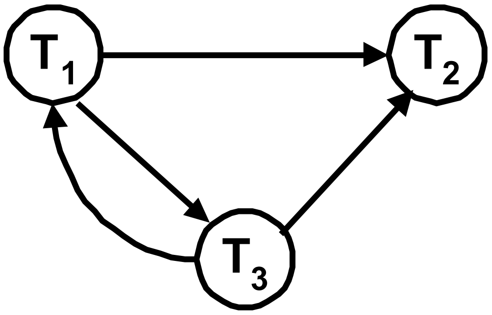
is



The conflict graph for the schedule

$r_1(x)$ $r_1(y)$ $r_3(z)$ $w_3(z)$ $r_3(x)$ $r_2(z)$ $w_1(x)$ $w_1(y)$ $w_2(z)$ $w_2(y)$ $w_3(x)$

is



This provides a computationally tractable test for conflict serializability:

Theorem: The schedule S is conflict serializable iff its conflict graph is acyclic. □

The Relationship between View Serializability and Conflict Serializability:

Theorem: If a schedule S is conflict serializable, then it is also view serializable. However, the converse is not the case. □

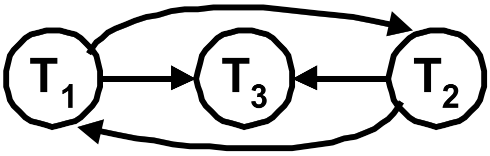
Example:

$T_1 = r_1(x)\ w_1(x)$ $T_2 = w_2(x)$ $T_3 = w_3(x)$

The schedule

$$S = r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$$

is not conflict serializable, since the conflict graph is



However, S is clearly view equivalent to

$$S = r_1(x)\ w_1(x)\ w_2(x)\ w_3(x),$$

and so it is view serializable.

- Schedules which are view but not conflict serializable involve so-called *blind writes,* in which a value is written but never read.

- Even though it is overly restrictive, conflict serializability is often used because of the simplicity of the conflict-graph test.

Other types of serializability:

- Other formalizations of serializability may be found in the literature.
- Perhaps the most frequently seen is *final-state serializability.*
- These are studied mostly from a theoretical point of view to motivate the need for either view serializability or else conflict serializability, since these alternatives have drawbacks which make them undesirable.
- They will not be discussed here.

Two-Phase Locking:

Given that conflict-free schedules are an acceptable compromise, it remains to determine how to generate them. One of the most popular strategies is known as two-phase locking.

Assumptions:

- For a transaction to use a database object, it must request and be granted an appropriate *lock* on that object.

- There are two kinds of locks:

  - A *write lock* permits a transaction both to read and to write a database object. Only one transaction may hold a write lock on a given object at any point in time.

  - A *read lock* permits a transaction to read a database object, but not to write it. Any number of transactions may hold simultaneous read locks on an object, but read and write locks may not coexist on the same object.

- The following operations exist:

  - rlock(x): Request a read lock on object x. This request may be granted provided there are no current write locks on x.

  - wlock(x): Request a write lock on x. This request may be granted only in the case that there are no locks on x, save that the transaction requiring the lock may already have a read lock on x.

  - upgrade(x): Convert a read lock to a write lock. This request may only be granted in the case that the requesting transaction already holds a read lock on x, and no other transaction holds such a lock on x.

  - downgrade(x): Convert a write lock to a read lock. This request may only be granted in the case that the requesting transaction already holds a write lock on x.

  - unlock(x): Dissolve all locks on x. This request dissolves any lock *which the requesting transaction holds* on object x.

- The following constraints exist on transactions:
  - Before an object is read, a lock (read or write) must be requested and granted.
  - Before an object is written, a write lock must be requested and granted.
  - All reads must be performed before the corresponding lock is released.
  - All writes must be performed before the corresponding write lock is downgraded or released.
  - It is also generally assumed that transactions do not request redundant locks. (This makes analyses simpler.)

A scheduler operates according to a *locking protocol* just in case these conventions are followed.

Example: $r_1(y)$ $r_3(z)$ $w_3(z)$ $r_2(z)$ $w_1(x)$ $w_1(y)$ $w_2(z)$ $w_2(y)$ $r_3(x)$ $w_3(x)$

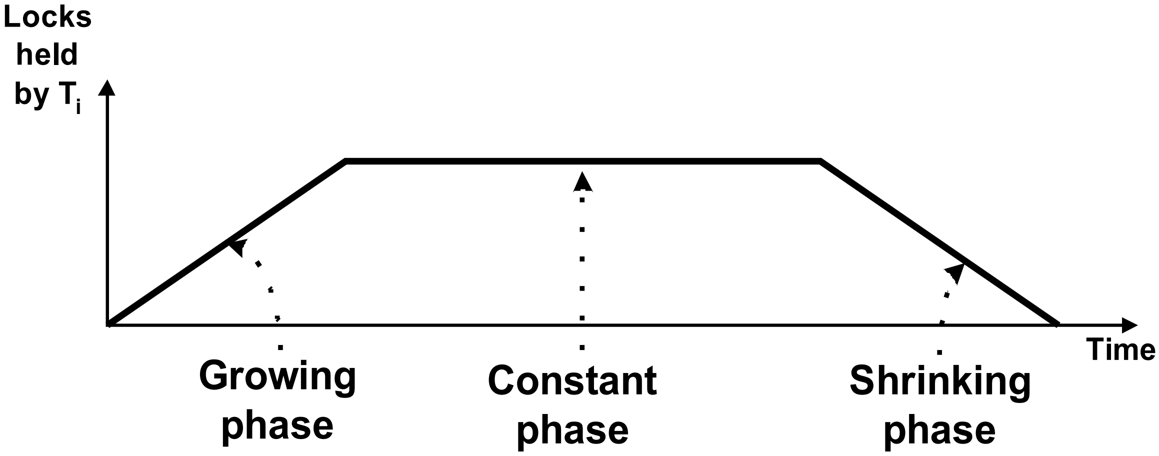
Here is one possible lock schedule:

Schedule	Lock x	Lock y	Lock z
Rlock$_1$(y)		R1	
r_1(y)		R1	
Wlock$_3$(z)		R1	W3
r_3(z)		R1	W3
w_3(z)		R1	W3
Unlock$_3$(z)		R1	
Rlock$_2$(z)		R1	R2
r_2(z)		R1	R2
Wlock$_1$(x)	W1	R1	R2
w_1(x)	W1	R1	R2
Unlock$_1$(y)	W1		R2
Wlock$_1$(y)	W1	W1	R2
w_1(y)	W1	W1	R2
Upgrade$_2$(z)	W1	W1	W2
w_2(z)	W1	W1	W2
Unlock$_1$(y)	W1		W2
Wlock$_2$(y)	W1	W2	W2
w_2(y)	W1	W2	W2
Downgrade$_1$(x)	R1	W2	W2
Rlock$_3$(x)	R1 R3	W2	W2
r_3(x)	R1 R3	W2	W2
Unlock$_1$(x)	R3	W2	W2
Unlock$_3$(x)		W2	W2
Wlock$_3$(x)	W3	W2	W2
w_3(x)	W3	W2	W2
Unlock$_3$(x)		W2	W2
Unlock$_2$(y)			W2
Unlock$_2$(z)			

The *two-phase locking protocol* (2PL) requires that the following condition be met:

- For each transaction, after the first downgrade or unlock operation, no subsequent rlock, wlock, or upgrade operations are allowed.

This situation may be envisioned graphically as follows:



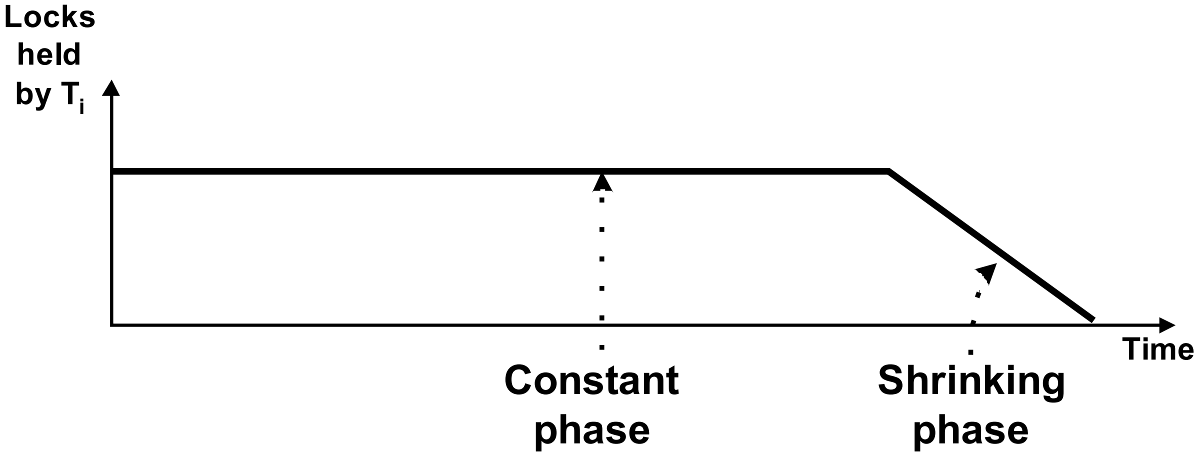
Note that the preceding schedule is not 2PL.

Theorem: Every schedule produced by a 2PL scheduler is conflict serializable. □

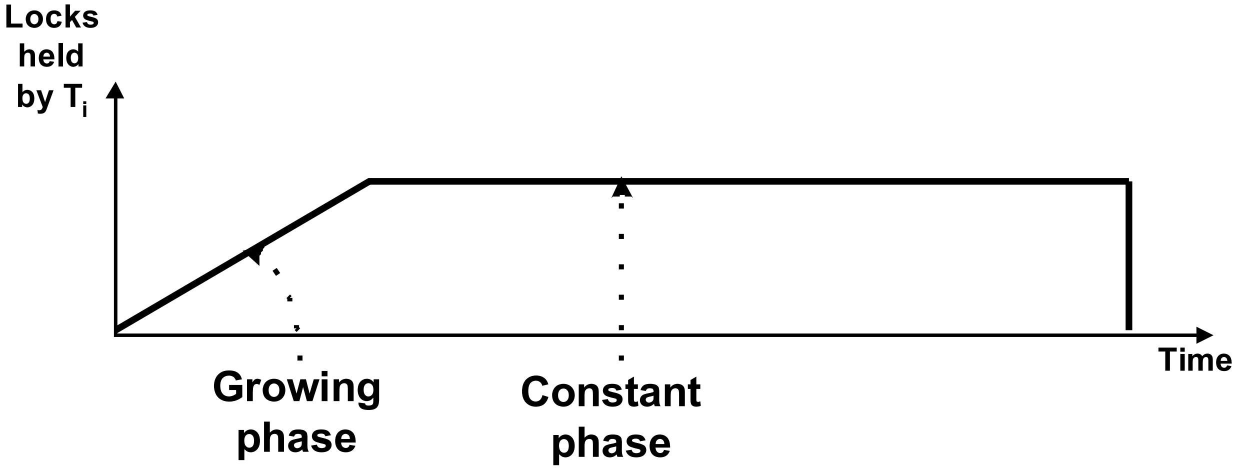
Theorem: Every serial schedule may be (trivially) converted to a 2PL schedule. □

Some common restrictions of 2PL:

In *conservative 2PL*, all locks must be requested at the beginning of a transaction:



In *rigorous 2PL*, all locks must be held until the end of the transaction:



In *strict 2PL*, all write locks must be held until the end of the transaction:

Each of these strategies has a use, as we shall see.

Deadlock Detection and Prevention:

- General schedules produced by 2PL are subject to deadlock.

Example: T_1 = r(x) r(y) w(x)
T_2 = r(y) r(x) w(y)

Suppose we adopt the following schedule:

T_1	T_2	Status
Rlock(x)		
r(x)		
	Rlock(y)	
	r(y)	
Rlock(y)		
r(y)		
	Rlock(x)	
	r(x)	
Wlock(x) (wait)	Wlock(y) (wait)	Deadlock!!

- Then each transaction must wait for the other to release something with which it is not finished.
- This is a *deadlock*.
- It *might* be the case that one of the transactions can release the needed resource, but the scheduler cannot know this, so it must respect the lock requests!
- Such early releases may also be inconsistent with recovery protocols. (More later).

Dealing with Deadlock:

There are two fundamental ways to deal with deadlock:

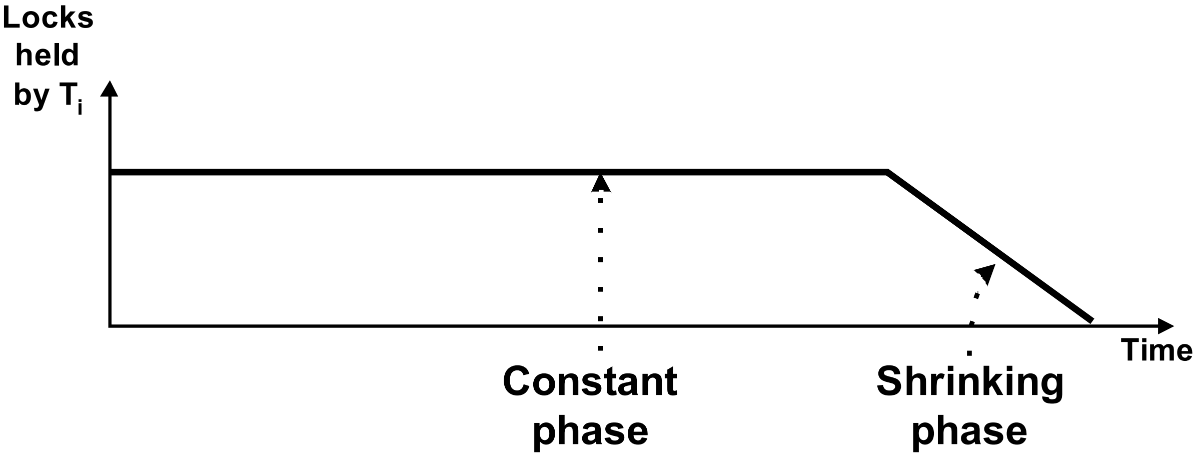
Pessimistic strategy: Use a scheduling algorithm which ensures that deadlock cannot occur.

Optimistic strategy: Take no measures to prevent deadlock. When deadlock is detected, execute a “repair” strategy.

The choice of strategy depends upon the kinds of transactions and the type of performance required.

Deadlock-free Scheduling:

Use a conservative 2PL locking strategy:



- It is clear that deadlock cannot occur, since a transaction must lock all resources which it needs at the time which it commences.

- Deadlock can only occur when a transaction locks a resource, and later requests a second resource which another process holds.

Another way to schedule transactions in a deadlock-free fashion is with

Timestamping:

- Each transaction is assigned a unique time-stamp.
- There are two main strategies:

Wait-Die:

```
If $T_i$ requests a resource held by $T_j$
   then if $T_i$ is older than $T_j$
     then allow $T_i$ to wait
     else  abort $T_i$, allowing $T_j$ to continue.
```

Wound-Wait:

```
If $T_i$ requests a resource held by $T_j$
   then if $T_i$ is older than $T_j$
     then abort $T_j$, allowing $T_i$ to continue
     else allow $T_i$ to wait.
```

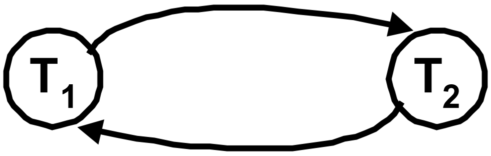
To understand why this strategy works, we introduce a new concept.

- The *wait-for graph* has an edge from a transaction holding a given resource, to one requesting the same resource.
- If it contains a cycle, there is a deadlock situation.

Previous example: T_1 = r(x) r(y) w(x)
T_2 = r(y) r(x) w(y)

T_1	T_2	Status
Rlock(x)		
r(x)		
	Rlock(y)	
	r(y)	
Rlock(y)		
r(y)		
	Rlock(x)	
	r(x)	
Wlock(x) (wait)	Wlock(y) (wait)	Deadlock!!

Here is the wait-for graph.



Cycles cannot occur when either the wait-die or the wound-wait strategy is implemented:

- In wait-die, there cannot be an edge from an older transaction to a younger one.
- In wound-wait, there cannot be an edge from a younger transaction to an older one.

- Remember that these strategies do not look for cycles in the graph, but only for edges whose nodes have particular forms of time stamps.
- The graph need not be and is not constructed!

Remark:

- The wait-for graph is complicated somewhat with read locks, since several transactions may hold such a lock on the same resource.
- In this situation, one must work with sets of transactions. The details are straightforward but omitted.

Livelock:

- *Livelock* occurs when a transaction is repeatedly aborted because of a deadlock situation.
- Livelock may be avoided with wait-die and wound-wait, provided that an aborted transaction is restarted with its original timestamp, and not a new one.

Deadlock Detection:

- To implement an optimistic deadlock-handling protocol, a method for detecting deadlock must be in place.

- The wait-for graph may be used.

- Cycle-detection algorithms for directed graphs are well-known, and reasonably efficient.

Optimistic and Pessimistic Concurrency Control:

- Optimistic and pessimistic strategies have been identified for deadlock management.

- These terms apply more generally to concurrency-control strategies.

  - A *pessimistic* strategy is one which attempts to avoid scheduling problems before they happen.
    - This strategy is appropriate when conflicts are likely to occur frequently.
    - Implies a higher overhead of producing acceptable schedules.

  - An *optimistic* strategy is one lets scheduling proceed rather freely, and then “repairs” situations in which conflicts occur.
    - This strategy is appropriate when conflicts are not likely to occur frequently.
    - Implies a lower overhead for producing acceptable schedules, but a higher overhead of detecting and correcting problems.

- There are numerous other strategies which embrace these principles.

Locking Granularity:

The size of an object to be locked by a single lock request is open to decision:

- A field of a tuple or record.
- A single tuple or record.
- A physical disk block.
- An entire relation.
- An entire database.

Generally, the larger the object to be locked:

- The less the possible concurrency.
- The easier it is to manage the concurrency.

- Access locks 2Kb. physical blocks.