Example Problem

Time	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);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Concurrency will be serializable only if it gives the result of one of the serial schedules.

Concurrency Control Protocols

- Using binary locks or read/write locks do not guarantee serializability.
- To guarantee serializability, it is necessary to follow an additional protocol concerning the positioning of locking and unlocking operations in every transaction.

  - Two-Phase Locking – 2PL

Two Phase Locking

- A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- Such a transaction can be divided into two phases:
  - **Expanding or growing (first) phase**→ during which new locks on items can be acquired but none can be released.
  - **Shrinking (second) phase** → during which existing locks can be released but no new locks can be acquired.

Two Phase Locking (2PL)

- Lock conversion
  - upgrading of locks (from read-locked to write-locked) must be done during the expanding phase
  - downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase

Two Phase Locking

T1	T2	T3
	Lock-s(A)	
		Lock-s(A)
	Lock-x(B)	
	Unlock(A)	
		Lock-x(C)
	Unlock(B)	
Lock-s(B)		
		Unlock(A)
		Unlock(C)
Lock-x(A)		
Unlock(B)		
Unlock(A)		

The highlighted parts are the last locks taken, i. e. the serial schedule could be:

T2 -> T3 -> T1

- **If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable**, obviating the need to test for serializability of schedules.
- Guaranteeing serializability paid by reduce concurrency.

Limitations in 2PL

Using 2PL may cause the following problems.

- Does not permit *all possible* serializable schedules
- Cascading Rollback
- Deadlocks

2PL - Limitations

- Not all serializable schedules are allowed by 2PL.
- Example S1: T1:w1(x), **T3:w3(x),** T2:w2(y), T1:w1(y)
- The lock by T1 for y must occur after w2(y), so the unlock by T1 for x must also occur after w2(y) (according to 2PL).
- Due to 2PL, w3(x) cannot occur where shown in S1 because T1 holds the x lock at that point.
- However, S1 is serializable (equivalent to T2, T1, T3).

2PL – Limitations

Cascading Rollback : Recursively abort the transactions that read data written by aborted transactions.

T1	T2	T3
Lock-x(A)		
R(A)		
W(A)		
Lock-x(B)		
R(B)		
Unlock(A)		
	Lock(A)	
	R(A)	
	W(A)	
	Unlock(A)	
		Lock-s(A)
		R(A)

If transaction T1 rollbacks then transaction T2 and T3 has to rollback.

Cascading Rollback

Solutions to avoid Cascading Rollbacks :

- Strict Two Phase Locking Protocol
- Rigorous Two Phase Locking Protocol

2PL – Limitations

- Deadlocks : occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T’ in the set.
- Waiting queue is maintained.
- Both T and T’ are waiting until the other release the lock.

2PL – Limitations

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);

T_1'	T_2'

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

T1	T2
Read_lock(Y)	
Read_item(Y)	
	Read_lock(X)
	Read_item(X)
Write_lock(X)	
	Write_lock(Y)

T1 is waiting until the T2 lock on X is released.

T2 is waiting until the T1 lock on Y is released.

2PL Types

- Due to the issues with 2-PL such as Cascading rollbacks and Deadlocks some enhancements/modifications have been made on 2-PL to improve it. There are three categories:
  - Strict 2-PL
  - Rigorous 2-PL
  - Conservative 2-PL

2PL Types

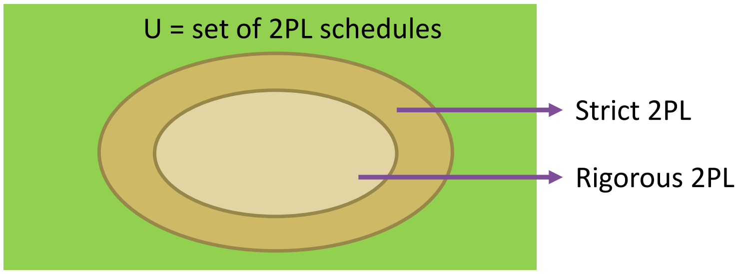
Strict 2PL (most popular)

- Does not release **exclusive (write) locks** until after it commits or aborts.
- Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule.
- Strict 2-PL ensures that the schedule is:
  - Recoverable
  - Cascadeless
- Not deadlock free.

2PL Types

Rigorous 2PL

- A transaction T does not release **any of its locks** (exclusive or shared) until after it commits or aborts.
- Ensures that the schedule is Recoverable and Cascadeless.
- Deadlocks are possible.
- Easier to implement than strict 2PL.



2PL Types

Conservative 2PL/Static 2PL

- Requires a transaction to lock all the items it accesses before the transaction begins execution, by pre declaring its read-set and write-set. Hence this protocol does not have a Growing Phase.
- If any item can not be locked, does not lock any.
- Deadlock free; difficult to use.
- Releasing locks has no restrictions.
- Still face drawbacks like Cascading Rollbacks.

2PL Types

T1	T1	T1
Lock-s(A)	**Lock-s(A)**	**Lock-s(A)**
Read(A)	Read(A)	**Lock-x(B)**
Lock-x(B)	**Lock-x(B)**	Read(B)
Unlock(A)	Read(B)	Write(B)
Read(B)	Write(B)	Read(A)
Write(B)	commit	**Unlock(A)**
commit	**Unlock(B)**	commit
Unlock(B)	**Unlock(A)**	**Unlock(B)**

Strict 2PL: Exclusive locks are unlocked after commit.

Rigorous: unlock all the locks after commit.

Conservative: obtain all the locks before starting the transaction.

	T1	T2
1.	Lock-X(X)	
2.	Lock-X(Y)	
3.	Read(X)	
4.	Write(X)	
5.	Unlock(X)	
6.		Lock-X(X)
7.		Read(x)
8.		Write(x)
9.		Unlock(x)
10.	Read(Y)	
11.	Write(Y)	
12.	Unlock(Y)	
13.	Commit	
14.		Commit

Follows Conservative 2-PL and does not meet the requirements of Strict and Rigorous 2-PL, since X and Y are unlocked before the transaction commits.

Deadlocks

Time ↓

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Schedule

Handling Deadlocks

Two main methods

- Deadlock prevention
- Deadlock detection & recovery

Deadlock Detection

- Can be used to detect any deadlock situation in advance.
- Wait for Graphs
  - Track if any deadlock situation may arise.
  - For each transaction entering into the system, a node is created.
  - When a transaction T_i requests for a lock on an item X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j.
  - If T_j releases item X, the edge between them is dropped and T_i locks the data item.
  - The system keeps checking if there's any cycle in the graph.

Deadlock Detection

Precedence graph

- Each transaction is a vertex
- Arcs from T1 to T2 if
  - T1 reads X before T2 writes X
  - T1 writes X before T2 reads X
  - T1 writes X before T2 writes X

Wait-for Graph

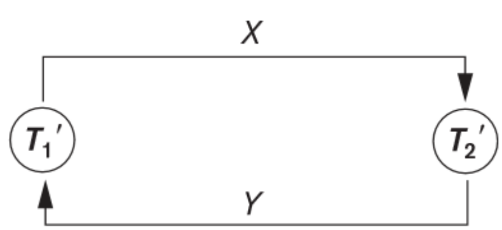
- Each transaction is a vertex
- Arcs from T2 to T1 if
  - T1 read-locks X then T2 tries to write-lock it
  - T1 write-locks X then T2 tries to read-lock it
  - T1 write-locks X then T2 tries to write-lock it

Deadlock Detection

Time

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

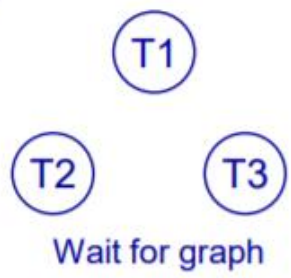
Schedule

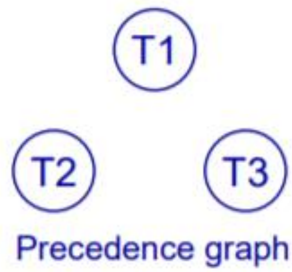


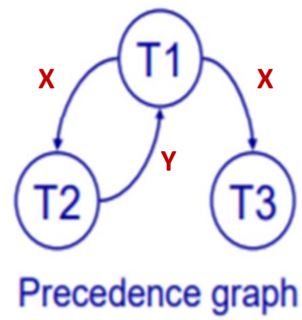
Wait for Graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)

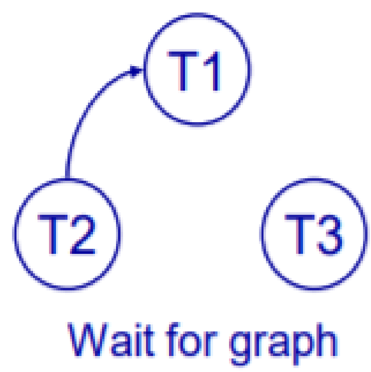






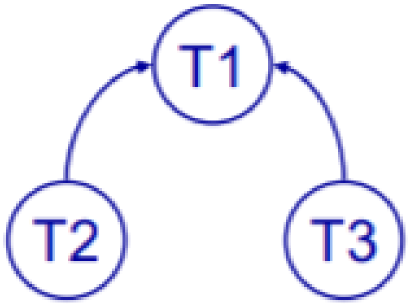
Wait-for Graph

T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) **tries read-lock(X)**
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait-for Graph

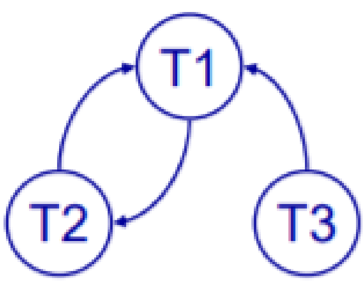
T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) **tries read-lock(X)**
T1 Write(Y)



Wait for graph

Wait-for Graph

T1 Read(X) read-locks(X)
T2 Read(Y) **read-locks(Y)**
T1 Write(X) write-lock(X)
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) tries read-lock(X)
T1 Write(Y) **tries write-lock(Y)**



Wait for graph

Deadlock Detection

- One problem with this approach is the matter of determining when the system should check for a deadlock.
  - When a new edge is added to the graph
- If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.
- Choosing which transactions to abort is known as **victim selection**.
  - avoid selecting transactions that have been running for a long time and that have performed many updates.
  - select transactions that have not made many changes (younger transactions).

Deadlock Detection

Timeouts

- If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.
- Do not check whether a deadlock actually exists or not.

Issues in Deadlock Recovery

I. **Issue of choosing a victim** – determine which transaction(s) among a set of deadlocked transactions to roll back to break the deadlock.

II. **Issue of rollback operation** - determine how far the chosen victim transaction should be rolled backed (total or partial).

III. **Issue of starvation** - avoid a situation where some transaction may always be chosen as the victim due to selections based on cost factors. This may prevent the transaction from ever completing its job.

Starvation

- Occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- **Reasons**

  - if the waiting scheme for locked items is unfair, giving priority to some transactions over others.

    - a first-come-first-served queue
    - allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.

Starvation

- The algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
  - The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.
  - The ***wait-die*** and ***wound-wait*** schemes (under timestamp ordering) avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is less.

Deadlock Prevention Protocols

- Conservative two-phase locking
  - Requires that every transaction lock all the items it needs in advance (generally not a practical assumption)
  - If any of the items cannot be obtained, none of the items are locked. The transaction waits and then tries again to lock all the items it needs.
- Timestamp ordering mechanism
- No waiting (NW) algorithm
- Cautious waiting (CW) algorithm

Timestamp ordering

- These techniques use the concept of transaction timestamp TS(T),which is a unique identifier assigned to each transaction.
- The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then $TS(T_1) < TS(T_2)$.
- Schemes that prevent deadlock are
    - wait-die
    - wound-wait

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***.

- An older transaction is allowed to wait on a younger transaction,
- Younger transaction requesting an item held by an older transaction is aborted and restarted.

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.

- Younger transaction is allowed to wait on an older one,
- Older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it.