Chapter 10

Update Transaction

ch10 Update Transaction

- **Def. 10.1 Transaction**
  - A transaction is a means to package together a number of database operations performed by a process, so the database system can provide several guarantees, called the ACID properties.

BEGIN TRANSACTION

op_1;

op_2;

.....

op_N;

END TRANSACTION

ch10 Update Transaction

- Begin a transaction
  - first database operation

- End a transaction by executing

  exec sql commit work;

  or

  exec sql rollback work;

BEGIN TRANSACTION

op_1;

op_2;

.....

op_N;

END TRANSACTION

What Does a Transaction Do?

1. **Return information from the database**
   - **RequestBalance transaction: Read customer's balance in database and output it**

2. **Update the database to reflect the occurrence of a real world event**
   - **Deposit transaction: Update customer's balance in database**

Transactions

- Many enterprises use databases to store information about their state
  - *e.g.*, Balances of all depositors at a bank
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - *e.g.*, Bank balance must be updated when deposit is made
- Such a program is called a transaction

Transactions

- The execution of each transaction must maintain the relationship between the database state and the enterprise state

- **Example:** banking applications with file system

- problems

1. Creating an inconsistent result
2. Errors of concurrent execution
3. Uncertainty as to when changes become permanent

❑Problems:

1. Creating an inconsistent result

① **Our application is transferring money from one account to another (different pages).**

② **One account balance gets out to disk (run out of buffer space)**

③ **and then the computer crashes.**

❑problems

2. Errors of concurrent execution

① **Teller 1 transfers money from Acct A to Acct B of the same customer, while Teller 2 is performing a credit check by adding balances of A and B.**

② **Teller 2 can see A after transfer subtracted, B before transfer added.**

❑problems

3. **Uncertainty as to when changes become permanent**

   - **At the very least, we want to know when it is safe to hand out money: don't want to forget we did it if system crashes, then only data on disk is safe.**

Transactions

- **Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs:**
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**

  } **ACID properties**

- **Atomicity （原子性）**
  - ① The set of record updates that are part of a transaction are indivisible (either they all happen or none happen).
  - ② This is true even in presence of a crash.
- **Consistency （一致性）**
  - If all the individual processes follow certain rules (money is neither created nor destroyed) and use transactions right,
  - then the rules won't be broken by any set of transactions acting together.

- **Isolation （隔离性）**
  - Means that operations of different transactions seem not to be interleaved in time
  - as if ALL operations of one Tx before or after all operations of any other Ty.
- **Durability （持久性）**
  - When the system returns to the logic after a Commit Work statement, it guarantees that all Tx Updates are on disk.
  - Now ATM machine can hand out money.

10.1 Transactional Histories

- **Actions in the database**
  - **Reads and Writes of data items**
    - **$R_i(A)$: transaction with identification number i (T_i) reads data item A.**
    - **$W_j(B)$: transaction T_j writes B.**
  - **A Update Statement will result in a lot of operations:**

$$R_j(B_1)\ W_j(B_1)\ R_j(B_2)\ W_j(B_2)\ \ldots\ R_j(B_n)\ W_j(B_n)$$

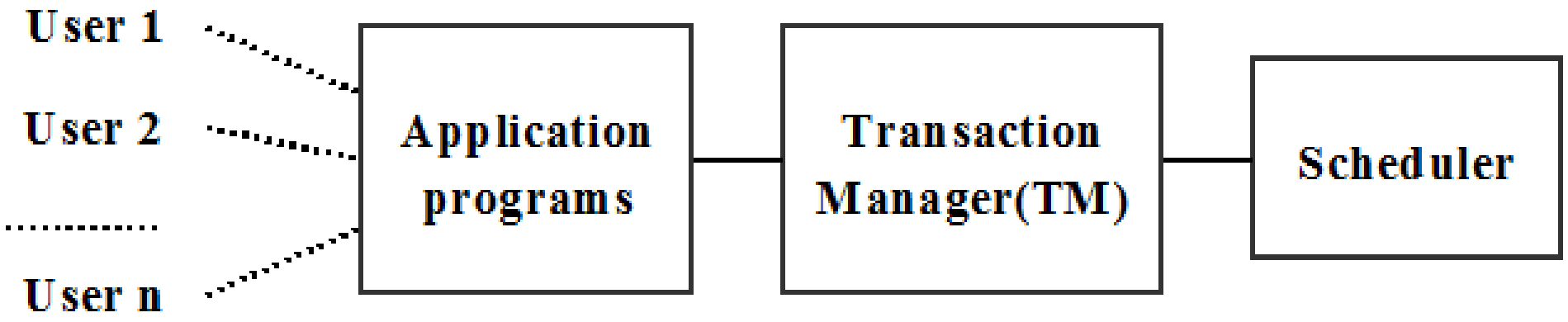
[10.1.2] history or schedule（调度）

Figure 10.1 The job of the Scheduler

- look at the history of operations as it comes in and

$R_2(A,50)W_2(A,20)R_1(A,20)R_1(B,50)R_2(B,50)W_2(B,80)\ C_1\ C_2$

- provide the Isolation guarantee, by sometimes delaying some operations, and occasionally insisting that some transactions be aborted.



10.1 Transactional Histories

- **serial schedule（串行调度）**
  - all operations of a T_x are performed in sequence with no interleaving with other transactions.

$T_1; T_2; T_3; \ldots\ldots; T_x;$

$Op_{1,1}; Op_{1,2}; \ldots\ldots; Op_{1,i1};$

$Op_{2,1}; Op_{2,2}; \ldots\ldots; Op_{2,i2};$

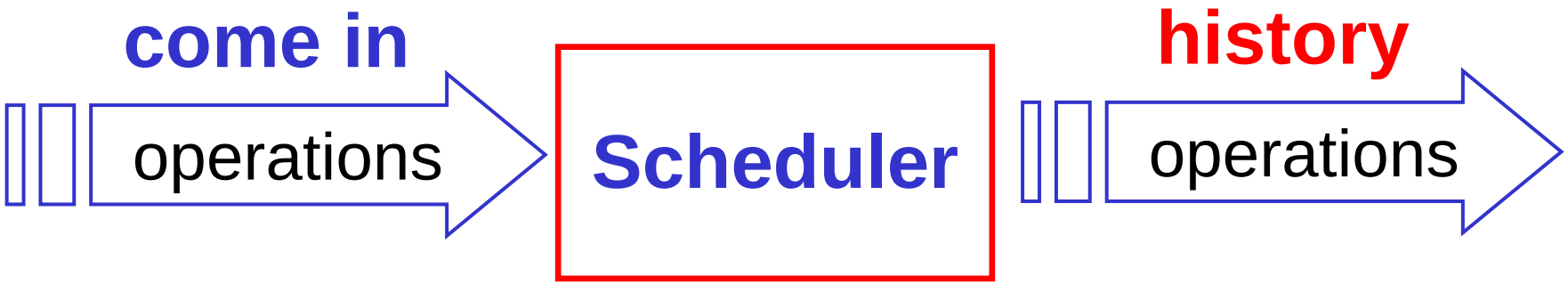
$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

$Op_{x,1}; Op_{x,2}; \ldots\ldots; Op_{x,ix};$

❑Scheduler（调度器）

- 隶属于同一个事务的访问操作，其到达顺序是固定的；
- 来自于不同事务的访问操作，它们到达的先后次序是不确定的。

10.1 Transactional Histories



- **Scheduler（调度器）**
  - **Scheduler assures that the sequence of operations in history is equivalent in effect to some serial schedule.**
  - **maybe**
    - **delaying some operations**
    - **some transactions be aborted**

Example 10.1.1

- The two elements A and B in (10.1.2) are account records with each having balance 50 to begin with.
  - T_1 is adding up balances of two accounts
  - T_2 is transferring 30 units from A to B

$R_2(A,50)W_2(A,20)R_1(A,20)R_1(B,50)R_2(B,50)W_2(B,80)\ C_1\ C_2$

  - result of transaction (schedule 10.1.2)
    - T_1: A+B = 70 (fails the credit check)
    - T_2: A = 20, B = 80

❑Example 10.1.1 (cont.)

	schedule 10.1.2		data item	
	T2	T1	account A	account B
1	R2(A, 50);		50	50
2	W2(A, 20);		20	50
3		R1(A, 20);	20	50
4		R1(B, 50);	20	50
5	R2(B, 50);		20	50
6	W2(B, 80);		20	80
7		C1	20	80
8	C2		20	80

10.1 Transactional Histories

- **Example 10.1.1 (cont.)**
  - But this could never have happened in a serial schedule, where all operation of T_1 occurred before or after all operations of T_2. (following)

$R_1(A,50)R_1(B,50)\ C_1\ R_2(A,50)W_2(A,20)\ R_2(B,50)W_2(B,80)\ C_2$

or

$R_2(A,50)W_2(A,20)\ R_2(B,50)W_2(B,80)\ C_2\ R_1(A,20)R_1(B,80)\ C_1$

ACID (1)

- **Atomic**
  - The set of updates contained in a transaction must succeed or fail as a unit.

- **Consistent**
  - Complete transactional transformations on data elements bring the database from one consistent state to another.

ACID (2)

- **Isolated**
  - Even though transactions execute concurrently, it appears to each successful transaction that it has executed in a serial schedule with the others.

- **Durable**
  - Once a transaction commits, the changes it has made to the data will survive any machine or system failures.

10.2 Interleaved Read/Write Operations

- If a serial history is always consistent, why don't we just enforce serial histories?

- Figure 10.3
  - T_x has relatively small CPU bursts and then I/O during which CPU has nothing to do.

- What do we want to do?
  - Figure 10.4: Let another T_y run during slack CPU time.

10.2 Interleaved Read/Write Operations

- **Figure 10.5 ~ 6**
  - **If we have many disks in use, we can keep the CPU 100% occupied.**
  - **When one thread does an I/O, want to find another thread with completed I/O ready to run again.**

10.3 Serializability and the Precedence Graph

- **Serializability Schedule（可串行化调度）**
  - **The series of operations is EQUIVALENT to a Serial schedule**
- **Scheduler（调度器）**
  - **find a set of rules for the Scheduler to allow operations by interleaved transactions and guarantee *Serializability*.**
  - **How can we guarantee this?**

10.3 Serializability and the Precedence Graph

- **First**
  - **If two transactions never access the same data items, so**
    - **We can commute operations in the history of requests permitted by the scheduler until all operations of one T_x are together (serial history).**
    - **The operations don't affect each other, and order doesn't matter.**

- **If we have operations by two different transactions that do affect the same data item, what then?**
  - **There are only four possibilities**
    1. **... $R_1(A)$ ..... $R_2(A)$ ...**
       - **This can be commuted.**
       - **If there is a third transaction T3, where:**
         ... $R_1(A)$ ... $W_3(A)$ ... $R_2(A)$ ...
         then the reads (R_1 and R_2) cannot be commuted.
    2. **... $R_1(A)$ ..... $W_2(A)$ ...**
    3. **... $W_1(A)$ ..... $R_2(A)$ ...**
    4. **... $W_1(A)$ ..... $W_2(A)$ ...**
       - **These two operations cannot commute.**

10.3 Serializability and the Precedence Graph

- **Def. 10.3.1 Conflicting Operations（冲突动作）**
  - Two operations $X_i(A)$ and $Y_j(B)$ in a history are said to *conflict* (i.e., the order matters) if and only if the following three conditions hold:
    1) $A \equiv B$
    2) $i \neq j$
    3) One of the two operations X or Y is a write (W).
       - Other can be R or W.

Def. 10.3.1 Conflicting Operations（冲突动作）

- Two operations $X_i(A)$ and $Y_j(B)$ in a history

1) $A \equiv B$
   - Operations on distinct data items never conflict.
2) $i \neq j$
   - Operations conflict only if they are performed by different transactions.
     - Two operations of the SAME transaction also cannot be commuted in a history, why ?
3) One of the two operations X or Y is a write (W).
   - Other can be R or W.

10.3 Serializability and the Precedence Graph

- **Figure 10.7 The Three Types**

1) $R_i(A) \rightarrow W_j(A)$

   - meaning, in a history, $R_i(A)$ followed by $W_j(A)$

2) $W_i(A) \rightarrow R_j(A)$

3) $W_i(A) \rightarrow W_j(A)$

10.3 Serializability and the Precedence Graph

- **Def. 10.3.2 An interpretation of an arbitrary history H consists of 3 parts.**
  1) **A description of the purpose of the logic being performed.**
  2) **Specification of precise values for data items being read and written in the history.**
  3) **A consistency rule, a property that is obviously preserved by isolated transactions of the logic defined in 1).**

10.3 Serializability and the Precedence Graph

- **Example 10.3.1**

H_1: $R_2(A)$ $W_2(A)$ $R_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$ C_1 C_2

- **Here is an interpretation**
  - T_1 is doing a credit check, adding up the balances of A and B.
  - T_2 is transferring money from A to B.
  - The Consistency Rule: Neither transaction creates or destroys money.

❑ Example 10.3.1

H_1: $R_2(A)$ $W_2(A)$ $R_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$ C_1 C_2

❑ The schedule H_1 is not serializability (SR) because

① If schedule H_1 is equivalent to serializability schedule $S(H_1)$, then

- $W_2(A)$ & $R_1(A)$ is conflicting operations, and $W_2(A) <<_{H1} R_1(A)$, so $W_2(A) <<_{S(H1)} R_1(A)$
  - ➢ It means that $T_2 <<_{S(H1)} T_1$
- $R_1(B)$ & $W_2(B)$ is conflicting operations, and $R_1(B) <<_{H1} W_2(B)$, so $R_1(B) <<_{S(H1)} W_2(B)$
  - ➢ It means that $T_1 <<_{S(H1)} T_2$

② so, schedule H_1 is not serializability. (non-SR)

10.3 Serializability and the Precedence Graph

- **Example 10.3.2 lost update (dirty write)**

  H_2: $R_1(A)$ $R_2(A)$ $W_1(A)$ $W_2(A)$ C_1 C_2

  - operations 3 and 4 imply that $T_1 <<_{S(H2)} T_2$
  - operations 2 and 3 imply that $T_2 <<_{S(H2)} T_1$

  ☙ so, H_2 is non-SR

- **an example of H_2**

  $R_1(A,100)$ $R_2(A,100)$ $W_1(A,140)$ $W_2(A,150)$ C_1 C_2

schedule	H_2	T_1 & T_2	T_2 & T_1
value of A	150	190	190

10.3 Serializability and the Precedence Graph

- **Example 10.3.3 Blind Writes**

  H_3: $W_1(A)$ $W_2(A)$ $W_2(B)$ $W_1(B)$ C_1 C_2

  - operations 1 and 2 imply that $T_1 <<_{S(H3)} T_2$
  - operations 3 and 4 imply that $T_2 <<_{S(H3)} T_1$

  ☙ so, H_3 is non-SR

- **an example of H_3**

  $W_1(A,50)$ $W_2(A,80)$ $W_2(B,20)$ $W_1(B,50)$ C_1 C_2

schedule	H_3	T_1 & T_2	T_2 & T_1
value of A	80	80	50
value of B	50	20	50

10.3 Serializability and the Precedence Graph

Def. 10.3.3. The Precedence Graph

- A precedence graph for a history H is a directed graph denoted by PG(H).
  - The vertices of PG(H) correspond to the transactions that have COMMITTED in H
    - Uncommitted transactions don't count.
  - An edge $T_i \rightarrow T_j$ exists in PG(H) whenever two conflicting operations X_i and Y_j occur in that order in H.
    - Thus, $T_i \rightarrow T_j$ should be interpreted to mean that T_i must precede T_j in any equivalent serial history S(H).

Theorem 10.3.4 The Serializability Theorem

- A history H has an equivalent serial execution S(H) iff the precedence graph PG(H) contains no circuit.

Proof.

- Assume there are m transactions involved, and label them T_1, T_2, . . ., T_m.
- Because In any directed graph with no circuit there is always a vertex with no edge entering it, thus there is a vertex, or transaction T_k, with no edge entering it. We choose T_k to be $T_i(1)$.

Proof.(cont.)

- Note that since $T_i(1)$ has no edge entering it, there is no conflict in H that forces some other transaction to come earlier.
- Now remove this vertex, $T_i(1)$, from PG(H) and all edges leaving it. Call the resulting graph $PG^1(H)$ with no circuit.
- Continue in this fashion, removing $T_i(2)$ and all it's edges from $PG^1(H)$, and so on, choosing $T_i(3)$ from $PG^2(H)$, . . ., $T_i(m)$ from $PG^{m-1}(H)$.

10.4 Locking Ensures Serializability

- Two-Phase Locking (2PL)
- 封锁（lock）
- 封锁类型
  - 常用的两种类型封锁
  - 锁相容矩阵

10.4 Locking Ensures Serializability

- 封锁（lock）
  - 使用封锁技术的前提
    - 在一个事务访问数据库中的数据时，必须先获得被访问的数据对象上的封锁，以保证数据访问操作的正确性和一致性。
  - 封锁的作用
    - 在一段时间内禁止其它事务在被封锁的数据对象上执行某些类型的操作（*由封锁的类型决定*）
    - 同时也表明：持有该封锁的事务在被封锁的数据对象上将要执行什么类型的操作（*由系统所采用的封锁协议来决定*）

10.4 Locking Ensures Serializability

- 封锁类型
  - 常用的封锁类型有两种

1. 排它锁 (eXclusive lock，简称X锁)
   - 又被称为 ‘写封锁’ (WL – Write Lock)

2. 共享锁 (Sharing lock，简称S锁)
   - 又被称为 ‘读封锁’ (RL – Read Lock)

10.4 Locking Ensures Serializability

- 排它锁（X锁）
  1. 特性
     - 只有当数据对象A没有被其它事务封锁时，事务T才能在数据对象A上施加‘X锁’；
     - 如果事务T对数据对象A施加了’X锁’，则其它任何事务都不能在数据对象A上再施加任何类型的封锁。

10.4 Locking Ensures Serializability

- 排它锁（cont.）

2. 作用
   - 如果一个事务T申请在数据对象A上施加‘X锁’并得到满足，则：事务T自身可以对数据对象A作读、写操作，而其它事务则被禁止访问数据对象A
   - 这样可以让事务T独占该数据对象A，从而保证了事务T对数据对象A的访问操作的正确性和一致性

3. 缺点：降低了整个系统的并行性

4. ‘X锁’必须维持到事务T的执行结束

10.4 Locking Ensures Serializability

- 共享锁（S锁）
  1. 特性
     - 如果数据对象A没有被其它事务封锁，或者其它事务仅仅以 ‘S锁’ 的方式来封锁数据对象A时，事务T才能在数据对象A上施加‘S锁’；

❑共享锁（cont.）

2. 作用

- 如果一个事务T申请在数据对象A上施加‘S锁’并得到满足，则：事务T可以‘读’数据对象A，但不能‘写’数据对象A
- 不同事务所申请的‘S锁’可以共存于同一个数据对象A上，从而保证了多个事务可以同时‘读’数据对象A，有利于提高整个系统的并发性
- 在持有封锁的事务释放数据对象A上的所有‘S锁’之前，任何事务都不能‘写’数据对象A

3. ‘S锁’不必维持到事务T的执行结束(依封锁协议而定)

10.4 Locking Ensures Serializability

- “排它锁”与“共享锁”的相互关系可以用如下图所示的“锁相容矩阵”来表示。

		其它事务已持有的锁		
		X锁	S锁	-
当前事务申请的锁	X锁	No	No	Yes
	S锁	No	Yes	Yes

锁相容矩阵

10.4 Locking Ensures Serializability

- 合适(well formed)事务
  - 如果一个事务在访问数据库中的数据对象A之前按照要求申请对A的封锁，在操作结束后释放A上的封锁，这种事务被称为合适事务。

- ‘合适事务’是保证并发事务的正确执行的基本条件。

10.4 Locking Ensures Serializability

- **Def. 10.4.1 Two-Phase Locking (2PL)**
  - Locks taken in released following three rules.

1) 锁申请规则
2) 锁申请的等待规则
3) 单个事务的锁申请与释放规则(2PL)

10.4 Locking Ensures Serializability

- **Def. 10.4.1 Two-Phase Locking (cont.)**

  1) Before T_i can read a data item, $R_i(A)$, scheduler attempts to Read Lock the item on it's behalf, $RL_i(A)$;

  before $W_i(A)$, try Write Lock, $WL_i(A)$.

- **Def. 10.4.1 Two-Phase Locking (cont.)**

  2) If conflicting lock on item exists, requesting T_i must WAIT.

  - **Conflicting locks corresponding to conflicting operations:**
    - **two locks on a data item conflict if they are attempted by different T_is and at least one of them is a WL.**

10.4 Locking Ensures Serializability

3) There are two phases to locking

- **phase Ⅰ : the growing phase (增长阶段)**
- **phaseⅡ: the shrinking phase (收缩阶段)**
  - **when locks are released: $RU_i(A)$;**
  - **The scheduler must ensure that can't shrink (drop a lock) and then grow again (take a new lock).**

- **Rule 3) implies can release locks before Commit;**
- **More usual to release all locks at once on Commit, and we shall assume this in what follows.**

10.4 Locking Ensures Serializability

- Note that a transaction can never conflict with its own locks!
  - If T_i holds RL on A, can get WL on A so long as no other T_x holds a lock on A (must be RL).
  - a T_x with a WL doesn't need a RL
    - WL more powerful than RL

10.4 Locking Ensures Serializability

- **Clearly locking is defined to guarantee that a circuit in the Precedence Graph can never occur.**
  - **The first T_x to lock an item forces any other T_x that gets to it second to "come later" in any SG.**
  - **But what if other T_x already holds a lock the first one now needs?**
    - **This would mean a circuit, but in the WAIT rules of locking it means NEITHER T_x CAN EVER GO FORWARD AGAIN.**
    - **This is a DEADLOCK.**

❑ Example 10.4.1

	H4
1	$R_1(A)$
2	$R_2(A)$
3	$W_2(A)$
4	$R_2(B)$
5	$W_2(B)$
6	$R_1(B)$
7	C_1
8	C_2

non-SR

Serializability

	Schedule	Explanation	A	B

❑ Example 10.4.1

	H_4
1	$R_1(A)$
2	$R_2(A)$
3	$W_2(A)$
4	$R_2(B)$
5	$W_2(B)$
6	$R_1(B)$
7	C_1
8	C_2

non-SR

	Schedule
	$RL_1(A)$
1	$R_1(A, 50)$
	$RL_2(A)$
2	$R_2(A, 50)$
	$WL_2(A)$
	$RL_1(B)$
6	$R_1(B, 50)$
7	C_1
3	$W_2(A, 20)$
	$RL_2(B)$
4	$R_2(B, 50)$
	$WL_2(B)$
5	$W_2(B, 80)$
8	C_2

	H_4'
1	$R_1(A)$
2	$R_2(A)$
6	$R_1(B)$
7	C_1
3	$W_2(A)$
4	$R_2(B)$
5	$W_2(B)$
8	C_2

Serializability

❑ Example 10.4.2

	H_5
1	$R_1(A)$
2	$R_2(B)$
3	$W_2(B)$
4	$R_2(A)$
5	$W_2(A)$
6	$R_1(B)$
7	C_1
8	C_2

non-SR

	Schedule		A	B
	T_1	T_2		
	$RL_1(A)$		50	50
1	$R_1(A, 50)$		50	50
		$RL_2(B)$	50	50
2		$R_2(B, 50)$	50	50
		$WL_2(B)$	50	50
3		$W_2(B, 80)$	50	80
		$RL_2(A)$	50	80
4		$R_2(A, 50)$	50	80
		$WL_2(A)$	50	80
		T_2 waiting		
	$RL_1(B)$	.....		
	T_1 waiting			
	.....			

Deadlock !

10.4 Locking Ensures Serializability

- **Theorem. 10.4.2 Locking Theorem**
  - **A history of transactional operations that follows the 2PL discipline is SR (*Serializability*).**

10.5 Levels of Isolation

- 隔离级别 (level of isolation)
  - 每一个事务都可以自主选择，它自己与其它并发事务之间的隔离级别。
  - 一个事务所选择的隔离级别，决定了它在运行过程中(调度器)所采用的封锁策略。

10.5 Levels of Isolation

- 四种不同的隔离级别及其设置命令

```
SET TRANSACTION ISOLATION LEVEL
    READUNCOMMITTED |
    READCOMMITTED |
    READREPEATABLE |
    SERIALIZABLE
```

10.5 Levels of Isolation

- **READUNCOMMITTED：未提交读**
  - 在该方式下，当前事务不需要申请任何类型的封锁，因而可能会‘读’到未提交的修改结果
  - 禁止一个事务以该方式去执行对数据的‘写’操作，以避免‘写’冲突现象。

- **READCOMMITTED：提交读**
  - 在‘读’数据对象A之前需要先申请对数据对象A的‘共享性’封锁，在‘读’操作执行结束之后立即释放该封锁。
  - 以避免读取未提交的修改结果。

10.5 Levels of Isolation

- **READREPEATABLE：可重复读**
  - 在‘读’数据对象A之前需要先申请对数据对象A的‘共享性’封锁，并将该封锁维持到当前事务的结束。
  - 可以避免其它的并发事务对当前事务正在使用的数据对象的修改。

- **SERIALIZABLE：可序列化(可串行化)**
  - 并发事务以一种可串行化的调度策略实现其并发执行，以避免它们相互之间的干扰现象。

10.5 Levels of Isolation

- 不管采用何种隔离级别，在事务 ‘写’ 数据对象 **A** 之前需要先申请对数据对象 **A** 的 ‘排它性’ 封锁**(write-lock)**，并将该封锁维持到当前事务的结束。

10.5 Levels of Isolation

	Read 操作		Write 操作	
	锁类型	封锁时间	锁类型	封锁时间
READUNCOMMITTED 未提交读	No Lock	—	不允许执行 Write 操作	
READCOMMITTED 提交读	共享锁	读操作	排它锁	事务
READREPEATABLE 可重复读	共享锁	事务		
SERIALIZABLE 可串行化	共享锁	事务		

事务的隔离级别与封锁策略之间的关系

10.5 Levels of Isolation

- ***short-term locks***
  - A lock is taken prior to the operation (R or W) and released IMMEDIATELY AFTERWARD.

- ***long-term locks***
  - A lock are held until EOT (End Of Trans.).

10.5 Levels of Isolation

Levels of Isolation

Types of Locks

Levels of Isolation	Write locks	Read Locks (row)	Read Locks (Predicates)	Problem
Read Uncommitted	No (Read Only)	No	No	Dirty Reads
Read Committed	Yes	short-term	short-term	Lost Update
Repeatable Read	Yes	long-term	short-term	Update Anomaly
Serializable	Yes	long-term	long-term	No

10.6 Transactional Recovery

- **Logs**
  - **The log entries contain "Before Images" and "After Images" of every update made by a Transaction.**
    - **Before Image**
    - **After Image**

10.6 Transactional Recovery

- **Before Image**
  - In recovery, we can back up an update that shouldn't have gotten to disk (the transaction didn't commit) by applying a Before Image.
- **After Image**
  - In recovery, we can apply After Images to correct for disk pages that should have gotten to disk (the transaction did commit) but never made it.

10.6 Transactional Recovery

- **Three types of Logs**
  - **UNDO Logs**
    - only Before Image
    - Recovery for the transaction didn't commit.
  - **REDO Logs**
    - only After Image
    - Recovery for the transaction did commit.
  - **UNDO/REDO Logs**
    - Before Image & After Image
    - Recovery for all transactions.

Figure 10.13 (pg. 485)

Operation	Log entry	
$R_1(A, 50)$	(S, 1)	*
$W_1(A, 20)$	(W,1,A,50,20)	
$R_2(C, 100)$	(S, 2)	*
$W_2(C, 50)$	(W,2,C,100,50)	
C_2	(C, 2)	
$R_1(B, 50)$		
$W_1(B, 80)$	(W,1,B,50,80)	
C_1	(C, 1)	

- *: Log entry for Start transaction

Figure 10.13 (pg. 673)

Operation	Log entry	
$R_1(A, 50)$	(S, 1)	*
$W_1(A, 20)$	(W,1,A,50,20)	
$R_2(C, 100)$	(S, 2)	*
$W_2(C, 50)$	(W,2,C,100,50)	
C_2	(C, 2)	
$R_1(B, 50)$		*
$W_1(B, 80)$	(W,1,B,50,80)	
C_1	(C, 1)	

- *: no log entry is written for a Read operation.

Figure 10.13 (pg. 673)

Operation	Log entry	
$R_1(A, 50)$	(S, 1)	
$W_1(A, 20)$	(W,1,A,50,20)	*
$R_2(C, 100)$	(S, 2)	
$W_2(C, 50)$	(W,2,C,100,50)	*
C_2	(C, 2)	
$R_1(B, 50)$		
$W_1(B, 80)$	(W,1,B,50,80)	*
C_1	(C, 1)	

- *:
  - a) Write log for update operation.
  - b) with Before Image (BI) & After Image (AI)

Figure 10.13 (pg. 673)

Operation	Log entry	
$R_1(A, 50)$	(S, 1)	
$W_1(A, 20)$	(W,1,A,50,20)	
$R_2(C, 100)$	(S, 2)	
$W_2(C, 50)$	(W,2,C,100,50)	
C_2	(C, 2)	*
$R_1(B, 50)$		
$W_1(B, 80)$	(W,1,B,50,80)	
C_1	(C, 1)	*

- *:
  - a) Log entry for commit transaction
  - b) write Log Buffer to Log File

Operation	Log entry	Explanation
$R_1(A, 50)$	(S, 1)	Start transaction T_1 - no log entry is written for a Read operation.
$W_1(A, 20)$	(W,1,A,50,20)	T_1 Write log for update of A.balance. The value 50 is the Before Image (BI) for A.balance column in row A, 20 is the After Image (AI) for A.balance
$R_2(C, 100)$	(S, 2)	Another start transaction log entry.
$W_2(C, 50)$	(W,2,C,100,50)	Another Write log entry.
C2	(C, 2)	Commit T_2 log entry. (*Write Log Buffer to Log File*)
$R_1(B, 50)$		No log entry
$W_1(B, 80)$	(W,1,B,50,80)	
C1	(C, 1)	Commit T_1 (*Write Log Buffer to*

10.7 Recovery in Detail: Log Formats

- Assume that a System Crash occurred immediately after the $W_1(B,80)$ operation.

Operation	Log entry	
$R_1(A, 50)$	(S, 1)	
$W_1(A, 20)$	(W,1,A,50,20)	
$R_2(C, 100)$	(S, 2)	
$W_2(C, 50)$	(W,2,C,100,50)	
C_2	(C, 2)	
$R_1(B, 50)$		
$W_1(B, 80)$	(W,1,B,50,80)	
crash !!!		

	Log entry	ROLL BACK ACTION PERFORMED
5	(S, 1)	Make a note that T_1 is no longer "Active". Now that no transactions were active, we can end the ROLL BACK phase.
4	(W,1,A,50,20)	Transaction T_1 has never committed (it's last operation was a Write). Therefore, *the system performs UNDO of this update by Writing the Before Image value (50) into data item A*. Put T_1 into "Uncommitted List".
3	(S, 2)	Make a note that T_2 is no longer "Active"
2	(W,2,C,100,50)	Since T_2 is on "Committed List", we do nothing.
1	(C, 2)	Put T_2 into "Committed List"
	Failure	

	Log entry	ROLL FORWARD ACTION PERFORMED
6	(S, 1)	No action required.
7	(W,1,A,50,20)	T_1 is Uncommitted — No action required.
8	(S, 2)	No action required.
9	(W,2,C,100,50)	Since T_2 is on Committed List, we REDO this update by writing After Image value (50) into data item C.
10	(C, 2)	No action required.
11		We note that we have rolled forward through all log entries and terminate Recovery.

10.8 Checkpoints (检查点)

- **Def. 10.8.1: The Commit-Consistent Checkpoint**

1) No new transactions can start until the checkpoint is complete.
2) Database operation processing continues until all existing transactions commit, and all their log entries are written to disk.
3) The current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffers have been written out to disk.
4) When steps 1-3 have been performed, the system writes a special log entry, (CKPT), to disk, and the checkpoint is complete.

10.8 Checkpoints (检查点)

- **Def. 10.8.2: The Cache-Consistent Checkpoint**

1) No new transactions are permitted to start.
2) Existing transactions are not permitted to start any new operations.
3) The current log buffer is written out to disk, and after this the system ensures that all dirty pages in cache buffers have been written out to disk.
4) Finally a special log entry, (CKPT, List), is written out to disk, and we say that the checkpoint is complete. The List in this log entry contains a list of active transactions at the time the checkpoint is taken.

❑ Def. 10.8.3: The Fuzzy Checkpoint

① Prior to checkpoint start, the remaining pages dirty as of the previous checkpoint are forced out to disk (but the rate of writes should leave I/O capacity to support current transactions in progress; there is no critical hurry in doing this).

② At the start of the checkpoint process, no new transactions are permitted to start. Existing transactions cannot start any new operations.

❑ Def. 10.8.1: The Fuzzy Checkpoint (cont.)

③ The current log buffer is written out to disk with an appended log entry, ($CKPT_N$, List), as in the cache-consistent checkpoint procedure.

④ The set of pages in buffer that are dirty since the last checkpoint log, $CKPT_{N-1}$, is noted. This will probably be accomplished by special flags on the buffer directory. There is no need for this information to be made disk resident, since it is used only to perform the next checkpoint, not in case of recovery. The checkpoint is now complete.

10.8 Checkpoints

- **Examples of Checkpoints** 

- **10.9 Media Recovery**
- **10.10 Performance: The TPC-A Benchmark**