UNIT 4 TRANSACTIONS

Transaction Concepts –ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – **Deadlocks** – **Transaction Recovery** – Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery

Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock Handling

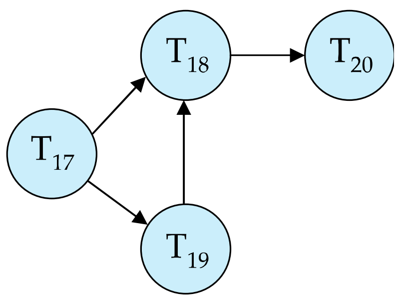
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - Impose **partial ordering of all data items** and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

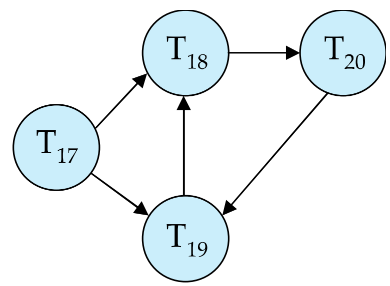
- **wait-die** scheme — non-preemptive
- **wound-wait** scheme — preemptive
- In both schemes, a rolled back transactions is restarted with its original timestamp.
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the **wait-for graph has a cycle.**
- Invoke a **deadlock-detection algorithm periodically** to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage

Recovery Algorithms

- Suppose transaction T_i **transfers $50 from account *A* to account *B***
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires **updates to A and B to be output to the database.**
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - **Modifying the database without ensuring that the transaction will commit** may leave the database in an **inconsistent state**
  - **Not modifying the database may result in lost updates** if failure occurs just after transaction commits

Recovery Algorithms

- Recovery algorithms have two parts
  1. Actions taken during **normal transaction processing** to **ensure enough information exists to recover from failures**
  2. Actions taken **after a failure to recover the database contents** to a **state that ensures atomicity, consistency and durability**

Storage Structure

- **Volatile storage**:
  - **Does not survive system crashes**
  - Examples: main memory, cache memory
- **Nonvolatile storage**:
  - **Survives system crashes**
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of **storage that survives all failures**
  - Approximated by **maintaining multiple copies** on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to **protect against disasters** such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - **Successful completion**
  - **Partial failure: destination block has incorrect information**
  - **Total failure: destination block was never updated**

Stable-Storage Implementation

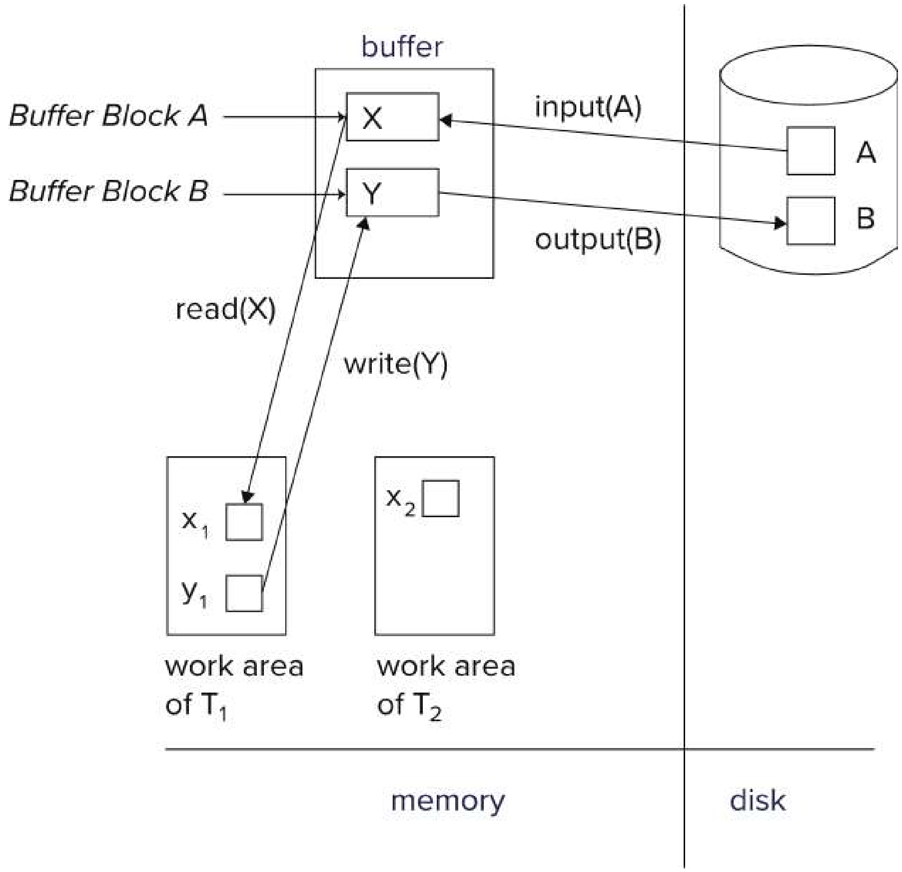
- **Protecting storage media** from failure during data transfer
- Execute output operation as follows ***(assuming two copies of each block):***

1. Write the **information onto the first physical block.**
2. When the **first write successfully completes, write the same information onto the second physical block.**
3. The output is completed only after the second write successfully completes.

Protecting storage media from failure

- Copies of a block may differ due to failure during output operation.
- To recover from failure:
  1. **First find inconsistent blocks:**
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Flash, Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems

Data Access



buffer
Buffer Block A
Buffer Block B
X
Y
input(A)
output(B)
A
B
read(X)
write(Y)
x1
y1
x2
work area
of T1
work area
of T2
memory
disk

Thank you