

Transaction Processing (Διαχείριση Δοσοληψιών)

- In *modern applications* databases are
 - *shared by more than one users at the same time*
 - who can *query* and *update* them
- It is *not possible* to provide each user with *their own copy of the database*
- A database management system must ensure that:
 - *concurrent access is provided*
 - *each user has a consistent view of the data*

Transaction Management

- The problems encountered in the development of large database applications led to the *development of transaction management techniques*
 - *Creation of inconsistent results (Consistency)*
 - the machine crashes in the middle of the execution process
 - *Errors in concurrent execution (Concurrency)*
 - *arbitrary concurrent execution* of processes lead to the *inconsistent views of data*
 - *Uncertainty as to when changes become permanent:*
 - can we be confident about the results residing in secondary storage even if processes have completed successfully?

The concept of a transaction was invented to solve these problems

Transaction Processing (Διαχείριση Δοσοληψιών)

- A *transaction* is a series of database operations (*reads* and *writes*) that form a *single logical entity* with respect to the application being modeled.
 - **Example:** a transfer of funds between accounts is considered a logical entity
- A *transaction commits* when it finishes *execution normally* otherwise it *aborts*
- *User transactions* appear to *execute in isolation*, although they may *execute concurrently*

Inconsistent view of Data (Ασυνέπεια στα Δεδομένα)

accounts

account#	lname	fname	type	balance
1234	Doe	John	Checking	900.00
5678	Doe	John	Savings	100.00
....				

- Process P1 transfers \$400 from account 1234 to account 5678
- Transfer is implemented by
 1. (S1) subtracting \$400 from the balance of account 1234
 2. (S2) adding \$400 to the balance of account 5678
- Accounts can be found in the following 3 states:

	Balance 1234	Balance 5678
Before P1	\$900	\$100
After S1	\$500	\$100
After S2	\$500	\$500

Inconsistent view of Data: Process Interleaving (Ασυνέπεια στα Δεδομένα: Παρεμβολές μεταξύ Διαδικασιών)

- Process P2 performs a *credit check* on the account holder and requires a *minimum of \$900 as the total balance* of the accounts to approve the issuance of a credit card
- P2 reads the *balance values* of the two accounts and computes their *sum*
- *P2 and P1 are running concurrently*
- *Execution is incorrect since the 'real' sum is 1000\$*

Process P1	Process P2
	sum:=0
subtract 400\$ from the balance of 1234 balance:=500	
	add balance of 1234 to sum sum:=sum+500 = 500
	add balance of 5678 to sum sum:=500 + 100 = 600
	reject
add \$400 to the balance of 5678	

Inconsistent view of Data: Process Interleaving

- It is equivalent to *serial executions* $P1, P2$
- This execution is **correct**
 - *both processes see the correct data*
- Transaction management must ensure that *only correct interleaving of processes takes place*

Process P1	Process P2
	sum:=0
	add balance of 1234 to sum sum:=900
subtract 400\$ from the balance of 1234 balance:=500	
	add balance of 5678 to sum sum:=900+ 100 = 1000
add \$400 to the balance of 5678	
	Issue approval

Transaction Management

- Transactions guarantee the following properties:
 - Atomicity (Ατομικότητα)
 - Consistency (Συνέπεια)
 - Isolation (Μεμονωμένη Εκτέλεση Διαδικασιών)
 - Durability (Διάρκεια)
- *Known as ACID Properties*

Transaction Management: ACID Properties

- *Atomicity*

- Transactions are considered atomic when considering their *effect* on the database:

- *all operations that make up the transaction are executed or none is*: the *set of operations* that make up the transaction is considered *indivisible*
- result of the transaction is *preserved* even when crashes occur:
 - a *database recovery procedure* performs a *rollback* to bring the database back to its state prior to transaction execution

Transaction Management: ACID Properties

- *Consistency*

- a transaction *should preserve a domain-specific consistency constraint* independently of whether it is *executed concurrently* with other transactions or in *isolation*.

- *Isolation (serializability)*

- serial schedule: when *transactions are executed one after the other*
- any *schedule of interleaved execution of transactions* is *equivalent* to some *serial schedule*

- *Durability*

- After a *transaction commits, it is guaranteed to be recoverable*
 - transactions are *durable to crashes*

Transaction Management (ACID Properties)

- *Atomicity* and *durability* are trivially satisfied by any *transaction* that performs *only read operations*
- Notation:
 - Transactions: $T_1, T_2, \dots, T_k$
 - $R_i(X)$: transaction T_i *reads* database item X
 - $R_i(X, u)$: transaction T_i *reads* database item X , u is the *value read*
 - $W_i(X)$: transaction T_i *writes* database item X
 - $W_i(X, u)$: transaction T_i *writes* database item X , u is the *value written*
 - C_i : transaction T_i *commits*
 - A_i : transaction T_i *aborts*

Transaction Management (ACID Properties)

- A *schedule* or *history* is an *interleaved sequence of operations*.
 - Transactions: T_1, T_2
 - Schedule : $R_2(A) \ W_2(A) \ R_1(A) \ R_1(B) \ R_2(B) \ W_2(B) \ C_1 \ C_2$
- A *schedule* is the *result of the translation of processes* - specified in some high-level language - into a *series of primitive operations*
- The *scheduler component* of the transaction processing component of a DBMS ensures that *only “correct” schedules are executed*

Transaction Management (ACID Properties)

- Given a set of *transaction specifications*, the *scheduler component* produces a *schedule that is equivalent to some serial execution of the transaction*
- If no such schedule is possible, the transaction manager *aborts* or *delays* some of the transactions
- The scheduler also detects *deadlocks*
 - Situations in which none of the transactions participating in the schedule can proceed unless one of them is aborted

Example: Scheduling

- Schedule $S = R_2(A) W_2(A) R_1(A) R_1(B) R_2(B) W_2(B) C_1 C_2$
 - involves transactions T_1, T_2
 - is *not equivalent* to any *serial execution* of the two transactions.
- *Interpretation* of the schedule
 - $T_1 = R_1(A), R_1(B), C_1$
 - $T_2 = R_2(A), W_2(A), R_2(B), W_2(B), C_2$

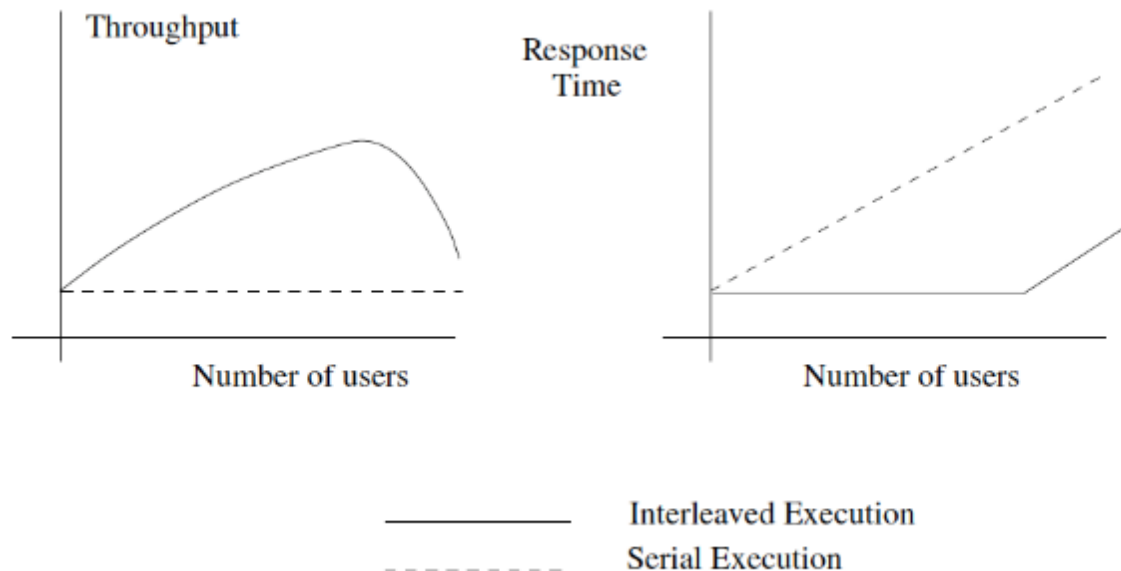
Example: Scheduling

- Schedule $S = R_2(A) \ W_2(A) \ R_1(A) \ R_1(B) \ R_2(B) \ W_2(B) \ C_1 \ C_2$
- $T_1 = R_1(A), R_1(B), C_1$
- $T_2 = R_2(A), W_2(A), R_2(B), W_2(B), C_2$
- S is correct *only if it is equivalent to one of the serial schedules T_1, T_2 or T_2, T_1*
 - Case 1: *serial schedule $S' = T_1, T_2$*
 - S: T_1 reads A *after* T_2 has modified it.
 - S' : the values of A and B *read* by T_1 *have not been modified* by T_2
 - Case 2: *serial schedule $S' = T_2, T_1$*
 - S: T_1 reads B before T_2 writes it.
 - S' : T_2 modifies the values of A and B, then T_1 reads it.

Hence the schedule has different effects than any serial execution

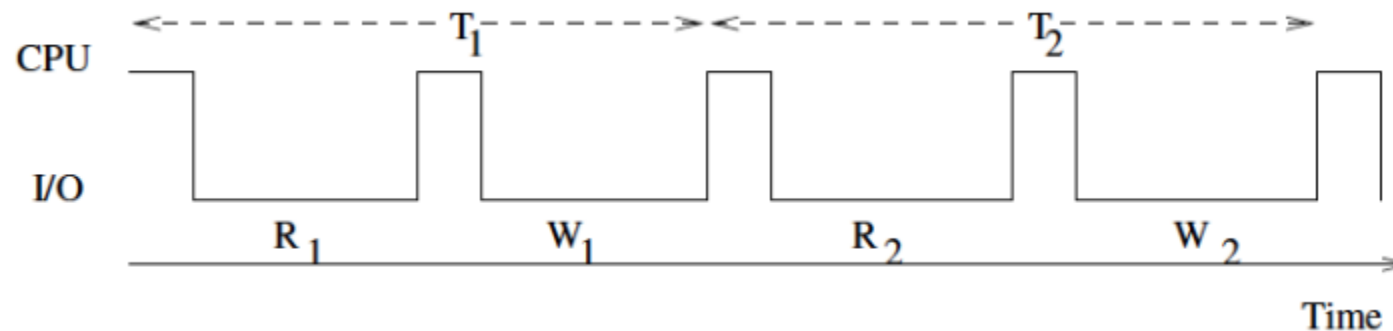
Interleaving of DB Operations

- Interleaving of database operations can yield *large performance gains*
- While some transaction is performing I/O, another transaction can use the CPU
- *System throughput*
 - the number of transactions that can finish execution in a given period of time) *increases* whereas *response time remains constant*



Serial vs Concurrent Execution (Example)

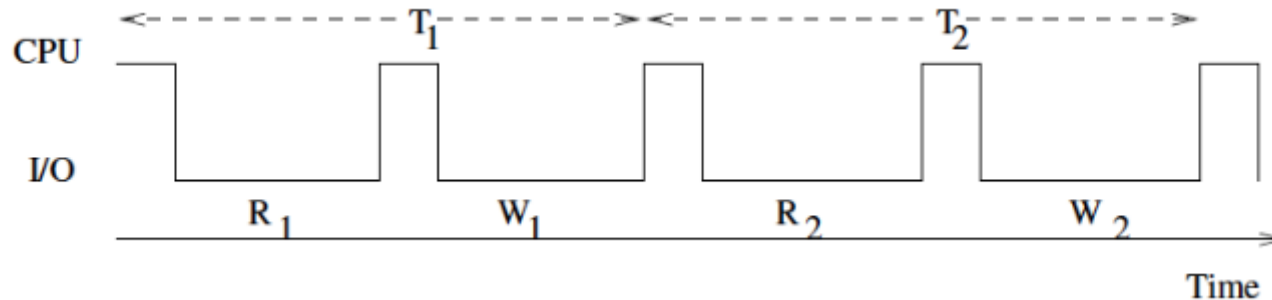
- *Transaction Manager* services database transactions
- Each transaction uses **both** CPU and I/O Resources
 - T_i : (cpu operation) $R_i()$ (cpu operation) $W_i()$ C_i
 - The system has a single CPU with a *5ms interval* and a *single disk*.
 - Each I/O operation requires *50ms of wait time*.
- *Serial Execution*: Resource usage



Serial vs Concurrent Execution (Example)

- Serial Execution

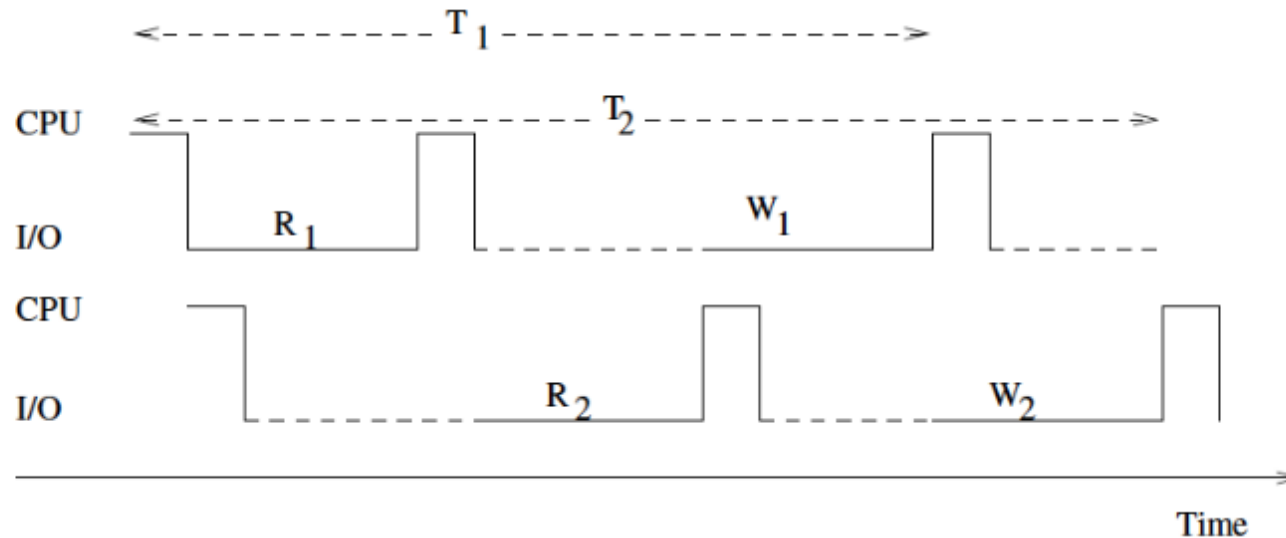
- a transaction needs 110ms
- *throughput* is 1 transaction per 110ms (9.09 transactions per second)
- CPU is *underutilized*: active 9.09% of the time



Interleaved execution of transactions can increase CPU utilization and thus the system throughput

Serial vs Concurrent Execution (Example)

- *Interleaved Execution*



- *throughput has increased*
- throughput will *increase* with the *number of transactions processes executed concurrently*
- *additional improvements:* more than one I/O devices are used

Testing Serializability

- Criteria to determine given a set of transactions S if
 - *interleaved schedules for S* are *equivalent* to some *serial execution for the transactions in S*
- *Conflicting database operations* when they
 - I. belong to *different transactions*
 - II. refer to the *same data item*
 - III. at least *one of them* is a *write operation*

a transaction reads an attribute and another tries to write its value

Properties of Schedules

- Two schedules are called *equivalent* if *for any initial state of the database, they result to the same database state.*
- Two schedules *are equivalent if all pairs of conflicting operations occur in the same order*
- A schedule is called *serializable* if it can *be shown to be equivalent to some serial execution of its transactions*
- Only *serializable schedules* are *acceptable*

- **Example:**

- $T_1 = R_1(A), R_1(B), W_1(A), C_1$

- $T_2 = W_2(A), R_2(A), C_2$

- $S = W_2(A) R_1(A) R_1(B) R_2(A) W_1(A)$

- *Is S serializable?*

- *Yes, it is equivalent to $T_2 T_1$*

There may be more than one serial schedules equivalent to some serializable schedule

Testing Schedule Serializability

- **Notation:** $op_i(X) \ll_S op_j(X)$ means that operation op_i of some transaction T_i on item X , precedes operation op_j of some transaction T_j on item X in schedule S
- **Cases:**
 - If $op_i(X) \ll_{S1} op_j(X)$ then $op_i(X) \ll_{S2} op_j(X)$ where ***S2*** is a ***serial schedule equivalent*** to ***S1***
 - If $op_i(X) \ll_{S1} op_j(X)$ and $op_j(Y) \ll_{S1} op_i(Y)$, then ***S1*** is ***not serializable***.
 - If it were, then, in the ***equivalent serial*** schedule ***S2***, transaction T_i should ***both precede*** and ***follow*** transaction T_j .

Testing Serializability: The lost update problem

- *The case in which two users want to update the same item in a database.*
 - Suppose transaction T_1 reads item A first : $R_1(A)$
 - Assume transaction T_2 reads item A: $R_2(A)$
 - T_2 writes immediately its value to A, before T_1 performs the update: $W_2(A)$
 - T_1 writes its value to A: $W_1(A)$
 - Hence any changes made by T_2 , are lost.

Testing Serializability: The lost update problem

- **Schedule:** $S1 = R_1(A) R_2(A) W_2(A) W_1(A) C_1 C_2$
- **Conflicting Operations:**
 - $R_1(A), W_2(A)$
 - $R_2(A), W_1(A)$
- *Assume* there is a *serial schedule S2 equivalent to S1*.
- $S1: R_1(A) <<_{s1} W_2(A) \rightarrow S2: R_1(A) <<_{s2} W_2(A)$
 - T1 must precede T2
- $S1: R_2(A) <<_{s1} W_1(A) \rightarrow S2: R_2(A) <<_{s2} W_1(A)$
 - T2 must precede T1
- The schedule is non-serializable

Testing Serializability: The blind write problem

- Occurs when a *transaction writes a value before reading it*
- **Schedule:** $S1 = W_1(A) W_2(A) W_2(B) W_1(B) C_1 C_2$
- **Conflicting Operations:**
 - $W_1(A) W_2(A)$
 - $W_2(B) W_1(B)$
- **Assume** there is a **serial schedule S2 equivalent to S1**.
- $S1: W_1(A) <<_{s1} W_2(A) \rightarrow S2: W_1(A) <<_{s2} W_2(A)$
 - T1 must precede T2
- $S1: W_2(B) <<_{s1} W_1(B) \rightarrow S2: W_2(B) <<_{s2} W_1(B)$
 - T2 must precede T1
- The schedule is non-serializable

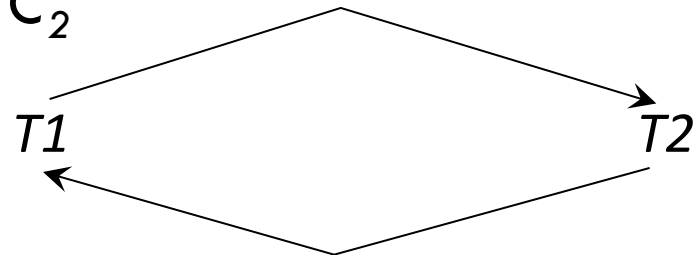
Testing Serializability: Precedence Graphs

- Given a *schedule* S , a *precedence graph* $PG(S)$ for S is a *directed graph* whose
 - vertices* correspond to the *transactions* T in the schedule and
 - set of edges* consists of *an edge* $T_i \rightarrow T_j$ whenever there exist two *conflicting operations* op_i, op_j in S and $op_i <<_S op_j$
- Example:**

■ $S1 = R_1(A) R_2(A) W_1(A) W_2(A) C_1 C_2$

$PG(S1)$

$PG(S2)$



■ Schedule $S2 = W_1(A) W_2(A) W_2(B) W_1(B) C_1 C_2$

Serializability

- **Theorem:** A schedule S is serializable if and only if the precedence graph $PG(S)$ contains no cycle
- **Lemma 1:** In any finite directed acyclic graph G , there is always a vertex u with no incoming edges
- **Proof:**
 - **Case 1:** If $PG(S)$ has no cycles, S is serializable
 - Assume that there are m transactions $T_1, T_2, \dots, T_m$ in S . We need to find a reordering $T_{i1}, T_{i2}, \dots, T_{im}$ of the transactions in order to construct an *equivalent serial schedule*
 - By Lemma 1, in the precedence graph $PG(S)$ there will be some vertex T_k with *no incoming edges*. Let T_{i1} be T_k .

Serializability

- Since T_k has no incoming edges in $PG(S)$, there is no pair of conflicting operations of T_k and some other transaction T_i such that the operation of T_i should precede that of T_k . Hence in the equivalent serial schedule, T_k should be the first to be executed.
- Remove T_k from $PG(S)$ along with all its incident edges. The resulting graph is still acyclic. Hence we can find a vertex T_l that has no incoming edges. Let T_{i2} be T_l . Then T_l should follow T_k in the serial schedule.
- Continue this process until the precedence graph contains one vertex. The corresponding transaction is the last one in the serial schedule.

■ **Case (2):** *If S is serializable, then $PG(S)$ is acyclic.*

- Let $PG(S)$ contain a cycle: $T_1 \ll_s T_2 \ll_s T_3 \dots \ll T_k \ll_s T_1$
(contradiction)