CONCURRENCY CONTROL

CHAPTER 21-22.1, 23 (6/E)

CHAPTER 17-18.1, 19 (5/E)

LECTURE OUTLINE

- Goal: Preserve *Isolation* of ACID properties
- Need to constraint how transactions interleave
  - Serializability
- Two-phase locking

TRANSACTION NOTATION

T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

T_2
read_item(X);
$X := X + M$;
write_item(X);

- Focus on read and write operations
  - T_1: b_1; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; e_1;
  - T_2: b_2; $r_2(Y)$; $w_2(Y)$; e_2;
- b_i and e_i specify transaction boundaries (begin and end)
- *i* specifies a unique transaction identifier (Tid)
  - $w_5(Z)$ means *transaction 5 writes out the value for data item Z*

SCHEDULE

- Sequence of interleaved operations from several transactions

	at ATM window #1	at ATM window #2
1	read_item(savings);	
2	savings = savings - $100;	
3		read_item(checking);
4	write_item(savings);	
5	read_item(checking);	
6		checking = checking - $20;
7		write_item(checking);
8	checking = checking + $100;	
9	write_item(checking);	
10		dispense $20 to customer;

$\equiv$ b_1; $r_1(s)$; b_2; $r_2(c)$; $w_1(s)$; $r_1(c)$; $w_2(c)$; $w_1(c)$; e_1; e_2;

SERIAL SCHEDULES

- A schedule *S* is **serial** if *no interleaving* of operations from several transactions
  - For every transaction *T*, all the operations of *T* are executed consecutively
- Assume consistency preservation (ACID property):
  - Each transaction, if executed on its own (from start to finish), will transform a consistent state of the DB into another consistent state
  - Hence, each transaction is correct on its own
  - Thus, any serial schedule will produce a correct result
- Although any serial schedule will produce a correct result, they might not all produce the *same* result.
  - If two people try to reserve the last seat on a plane, only one gets it. The serial order determines which one. The two orderings have different results, but either one is correct.
  - There are *n*! serial schedules for *n* transactions; any of them gives a correct result.

SERIAL SCHEDULES (CONT'D)

- Serial schedules are not feasible for performance reasons:
  - Long transactions force other transactions to wait
  - When a transaction is waiting for disk I/O or any other event, system cannot switch to other transaction
  - Solution: allow *some* interleaving (without sacrificing correctness)

ACCEPTABLE INTERLEAVINGS

- Executing some operations in another order causes a different outcome
  - …$r_1(X)$; $w_2(X)$… *vs.* …$w_2(X)$; $r_1(X)$…
    - T1 will read a different value for X
  - …$w_1(Y)$; $w_2(Y)$… *vs.* …$w_2(Y)$; $w_1(Y)$...
    - DB value for Y after both operations will be different
- Different execution order for two read operations is *not* a problem
  - …$r_1(Z)$; $r_2(Z)$… *vs.* …$r_2(Z)$; $r_1(Z)$...
    - both transactions read the same values of Z
- Two operations **conflict** if:
  1. They access the same data item X
  2. They are from two different transactions
  3. At least one is a write operation
     - Read-Write conflict : … r_1 (X); …; $w_2(X)$; …
     - Write-Write conflict : … $w_1(Y)$; …; $w_2(Y)$; …
- Two schedules are **conflict equivalent** if the relative order of *any two conflicting* operations is the same in both schedules

SERIALIZABLE SCHEDULES

- A schedule *S* with *n* transactions is **serializable** if it is conflict equivalent to *some* serial schedule of the same *n* transactions
- Serializable schedule “correct” because equivalent to some serial schedule, and any serial schedule acceptable
  - It will leave the database in a consistent state
  - Interleaving such that
    - transactions see data as if they were executed serially
    - transactions leave DB state as if they were executed serially
    - efficiency achievable through concurrent execution

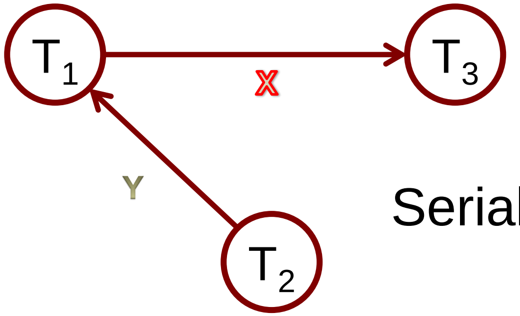
TESTING SERIALIZABILITY

- Consider all read_item and write_item operations in a schedule

1. Construct **serialization** graph
   - Node for each transaction T
   - Directed edge from T_i to T_j if some operation in T_i appears before a conflicting operation in T_j
2. The schedule is serializable if and only if the serialization graph has no cycles

- Is the following schedule serializable?

b_1; $r_1(X)$; b_2; $r_2(Y)$; $w_1(X)$; b_3; $w_2(Y)$; e_2; $r_1(Y)$; $r_3(X)$; e_3; $w_1(Y)$; e_1;



Serializable; equivalent to: $T_2 \rightarrow T_1 \rightarrow T_3$

b_2; $r_2(Y)$; $w_2(Y)$; e_2; b_1; $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; e_1; b_3; $r_3(X)$; e_3;

DATABASE LOCKS

- Use **locks** to ensure that conflicting operations cannot occur
  - **exclusive** lock for writing; **shared** lock for reading
  - cannot read item without first getting shared or exclusive lock on it
  - cannot write item without first getting write (exclusive) lock on it
- Request for lock might cause transaction to **block** (wait) because write lock is exclusive
  - Any lock on X (read or write) cannot be granted if some other transaction holds write lock on X
  - Write lock cannot be granted on X if some other transaction holds *any* lock on X

T1 \ T2	holds read (shared) lock	holds write (exclusive) lock
requests read lock	OK	block T1
requests write lock	block T1	block T1

- Blocked transactions are unblocked and granted the requested lock when conflicting transaction(s) release their lock(s)

ENFORCING SERIALIZABLE SCHEDULES

- **Rigorous two-phase locking (2PL)**:
  - If transaction will read X, obtain read lock on X
  - If transaction will write X, obtain write lock on X (or promote read lock to write lock)
  - Release all locks at end of transaction
    - whether commit or abort

- Rigourous 2PL ensures serializability of the resulting schedule

T1	T2
request_read(A);	
read_lock(A);	
read_item(A);	
A := A + 100;	
request_write(A);	
write_lock(A);	
write_item(A);	
	request_read(A);
request_read(B);	
read_lock(B);	
read_item(B);	
B := B -10;	
request_write(B);	
write_lock(B);	
write_item(B);	
commit; /*unlock(A,B)*/	
	read_lock(A);
	read_item(A);
	…

POTENTIAL PROBLEMS WITH RIGOROUS 2PL

- **Deadlock**: T_1 waits for T_2 waits for … waits for T_n waits for T_1
  - Requires assassin

- **Starvation**: T waits for write lock and other transactions repeatedly grab read locks before all read locks released
  - Requires scheduler

LECTURE SUMMARY

- Characterizing schedules based on serializability
  - Serial and non-serial schedules
  - Conflict equivalence of schedules
  - Serialization graph
- Two-phase locking
  - Guarantees serializability of resulting schedules
  - Deadlock and starvation