CSE 344

AUGUST 8TH

SCHEDULING

ADMINISTRIVIA

- **WQ7 due Monday**

- **HW8 due Wednesday**
  - uses JDBC API
  - (should be easy to figure out
    see the example code provided)

TRANSACTIONS

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries & updates

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others
- (Note: we won’t talk about rows much here...
  transactions are a broader concept than databases)

KNOW YOUR TRANSACTIONS: ACID

Atomic

- State shows either all the effects of txn, or none of them

Consistent

- Txn moves from a DBMS state where integrity holds, to another where integrity holds

Isolated

- Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

Durable

- Once a txn has committed, its effects remain in the database

SCHEDULES

A schedule is a sequence of interleaved actions from all transactions

SERIAL SCHEDULE

A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

Fact: nothing can go wrong if the system executes txns serially

- (rather, whatever does go wrong is the app's fault)
- But DBMS don't do that because we want better overall system performance

HOW DO WE KNOW IF A SCHEDULE IS SERIALIZABLE?

Notation:

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Key Idea: Focus on *conflicting* operations
(I.e., where changing order can change result)

CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction T_i:

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

CONFLICT SERIALIZABILITY

A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swaps of adjacent non-conflicting actions

Every conflict-serializable schedule is serializable

The converse is not true (why?)

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

⇩

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CONFLICT SERIALIZABILITY

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$

….

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

TESTING FOR CONFLICT-SERIALIZABILITY

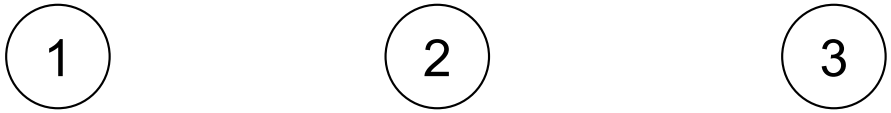
Precedence graph:

- **A node for each transaction T_i,**
- **An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j**

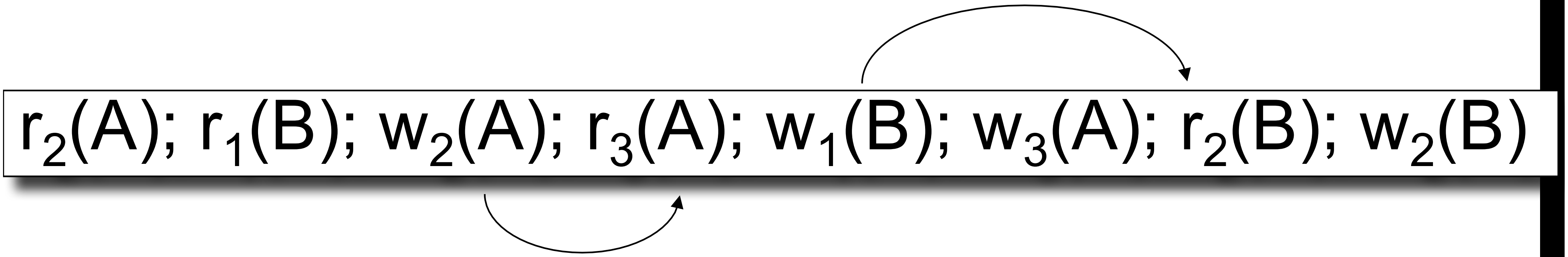
The schedule is conflict-serializable iff the precedence graph is acyclic

EXAMPLE 1

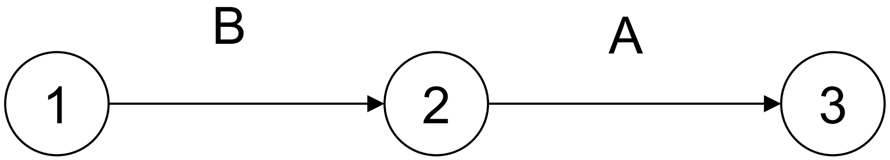
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



EXAMPLE 1



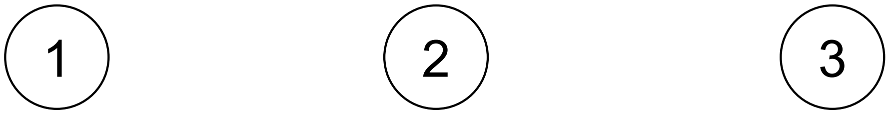
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



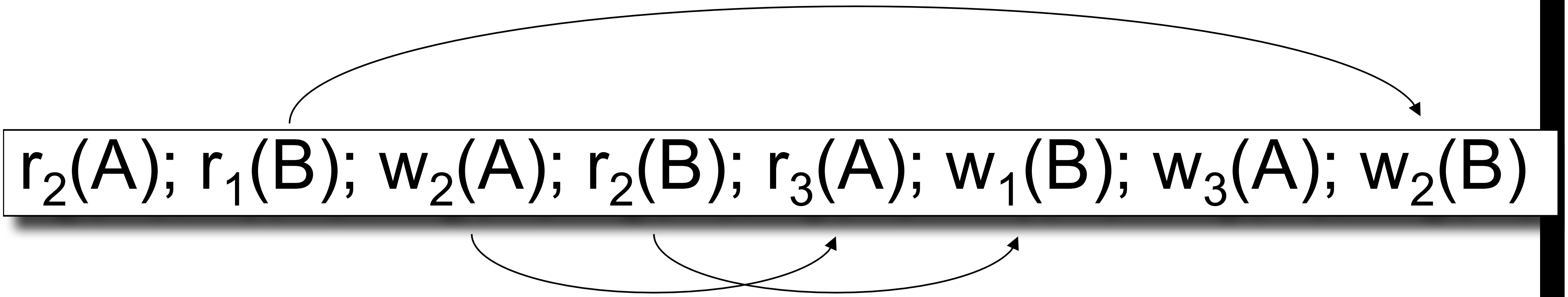
This schedule is conflict-serializable

EXAMPLE 2

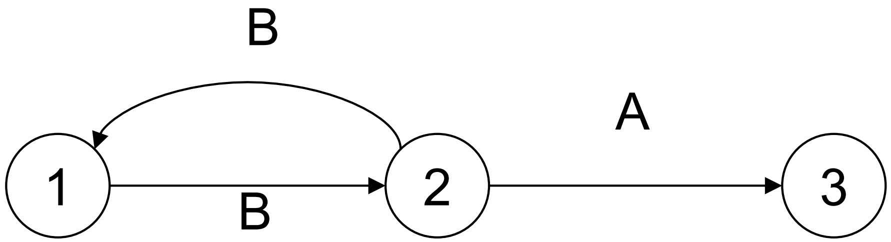
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



EXAMPLE 2



$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

SCHEDULER

Scheduler = the module that schedules the transaction’s actions, ensuring serializability

Also called **Concurrency Control Manager**

We discuss next how a scheduler may be implemented

IMPLEMENTING A SCHEDULER

Major differences between database vendors

Locking Scheduler

- Aka “pessimistic concurrency control”
- SQLite, SQL Server, DB2

Multiversion Concurrency Control (MVCC)

- Aka “optimistic concurrency control”
- Postgres, Oracle: Snapshot Isolation (SI)

We discuss only locking schedulers in this class

LOCKING SCHEDULER

Simple idea:

Each element has a unique lock

Each transaction must first acquire the lock before reading/writing that element

If the lock is taken by another transaction, then wait

The transaction must release the lock(s)

By using locks scheduler ensures conflict-serializability

WHAT DATA ELEMENTS ARE LOCKED?

Major differences between vendors:

Lock on the entire database

- SQLite

Lock on individual records

- SQL Server, DB2, etc

CASE STUDY: SQLITE

SQLite is very simple

More info: http://www.sqlite.org/atomiccommit.html

Lock types

- READ LOCK (to read)
- RESERVED LOCK (to write)
- PENDING LOCK (wants to commit)
- EXCLUSIVE LOCK (to commit)

SQLITE

Step 1: when a transaction begins

Acquire a **READ LOCK** (aka "SHARED" lock)

All these transactions may read happily

They all read data from the database file

If the transaction commits without writing anything, then it simply releases the lock

SQLITE

Step 2: when one transaction wants to write

Acquire a RESERVED LOCK

May coexists with many READ LOCKs

Writer TXN may write; these updates are only in main memory; others don't see the updates

Reader TXN continue to read from the file

New readers accepted

No other TXN is allowed a RESERVED LOCK

SQLITE

Step 3: when writer transaction wants to commit,
it needs *exclusive lock*, which can't coexists with *read locks*

Acquire a PENDING LOCK

May coexists with old READ LOCKs

No new READ LOCKS are accepted

Wait for all read locks to be released

Why not write
to disk right now?

SQLITE

Step 4: when all read locks have been released

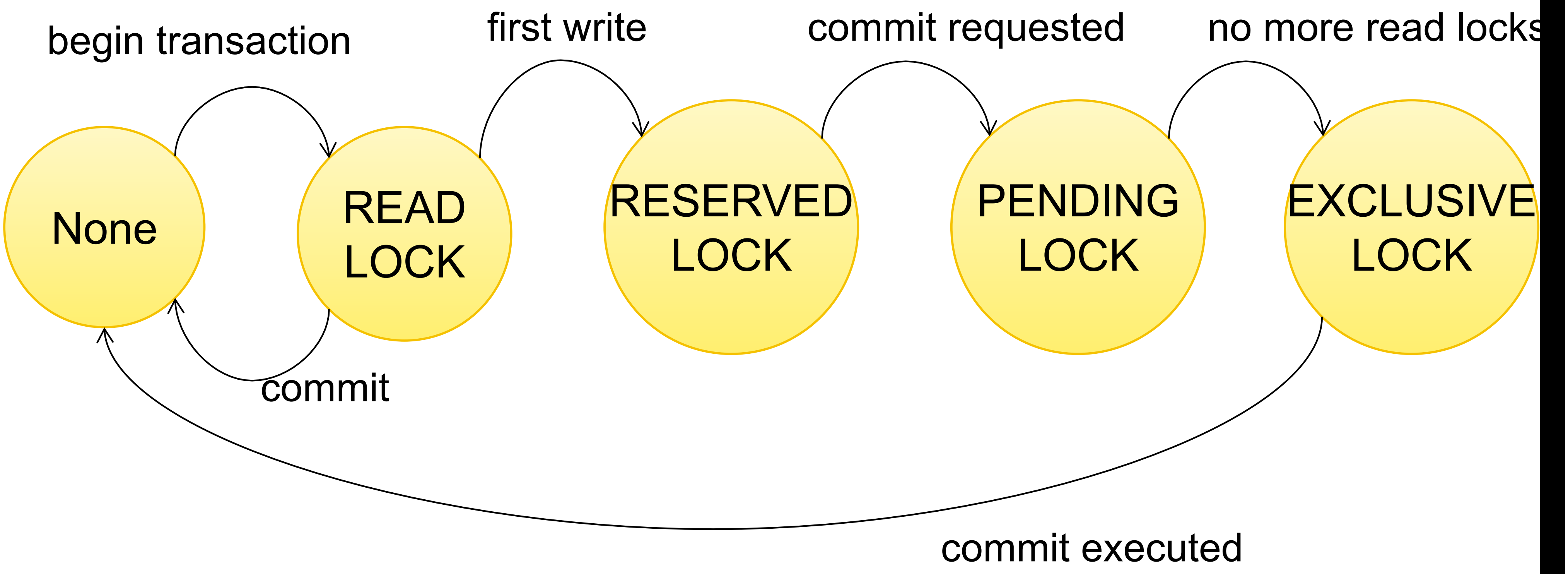
Acquire the EXCLUSIVE LOCK

Nobody can touch the database now

All updates are written permanently to the database file

Release the lock and COMMIT

SQLITE


begin transaction
first write
commit requested
no more read locks
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

SCHEDULE ANOMALIES

What could go wrong if we didn't have concurrency control:

- Dirty reads (including inconsistent reads)
- Unrepeatable reads
- Lost updates

Many other things can go wrong too

DIRTY READS

Write-Read Conflict

T_1: WRITE(A)

T_2: READ(A)

T_1: ABORT

INCONSISTENT READ

Write-Read Conflict

T_1: A := 20; B := 20;	
T_1: WRITE(A)	
	T_2: READ(A);
	T_2: READ(B);
T_1: WRITE(B)	

UNREPEATABLE READ

Read-Write Conflict

T_2: READ(A);

T_1: WRITE(A)

T_2: READ(A);

LOST UPDATE

Write-Write Conflict

T_1: READ(A)

T_1: A := A+5

T_1: WRITE(A)

T_2: READ(A);

T_2: A := A*1.3

T_2: WRITE(A);

MORE NOTATIONS

$L_i(A)$ = transaction T_i acquires lock for element A

$U_i(A)$ = transaction T_i releases lock for element A

A NON-SERIALIZABLE SCHEDULE

T1	T2
READ(A)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

EXAMPLE

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

BUT…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

Locks did not enforce conflict-serializability !!! What’s wrong ?

TWO PHASE LOCKING (2PL)

The 2PL rule:

In every transaction, all lock requests
must precede all unlock requests

EXAMPLE: 2PL TRANSACTIONS

T1	T2
L_1(A); L_1(B); READ(A)	
A := A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B := B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);

Now it is conflict-serializable

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability