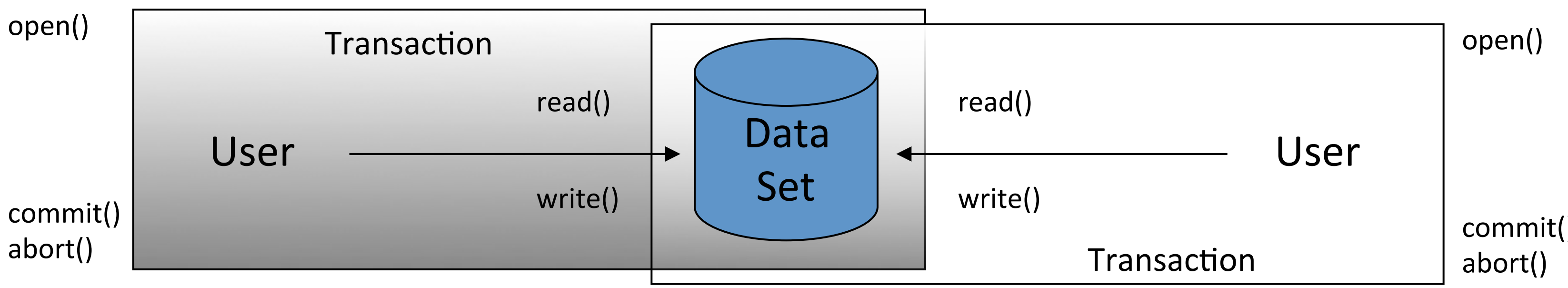
Transactions II

Controlling Concurrency

CS3524 Distributed Systems

Lecture 08

Concurrency Problems



- The interleaving of actions of concurrently executing transactions may lead to severe problems
- It depends on the sequence of actions, scheduled from different transactions, whether we create inconsistencies in our databases

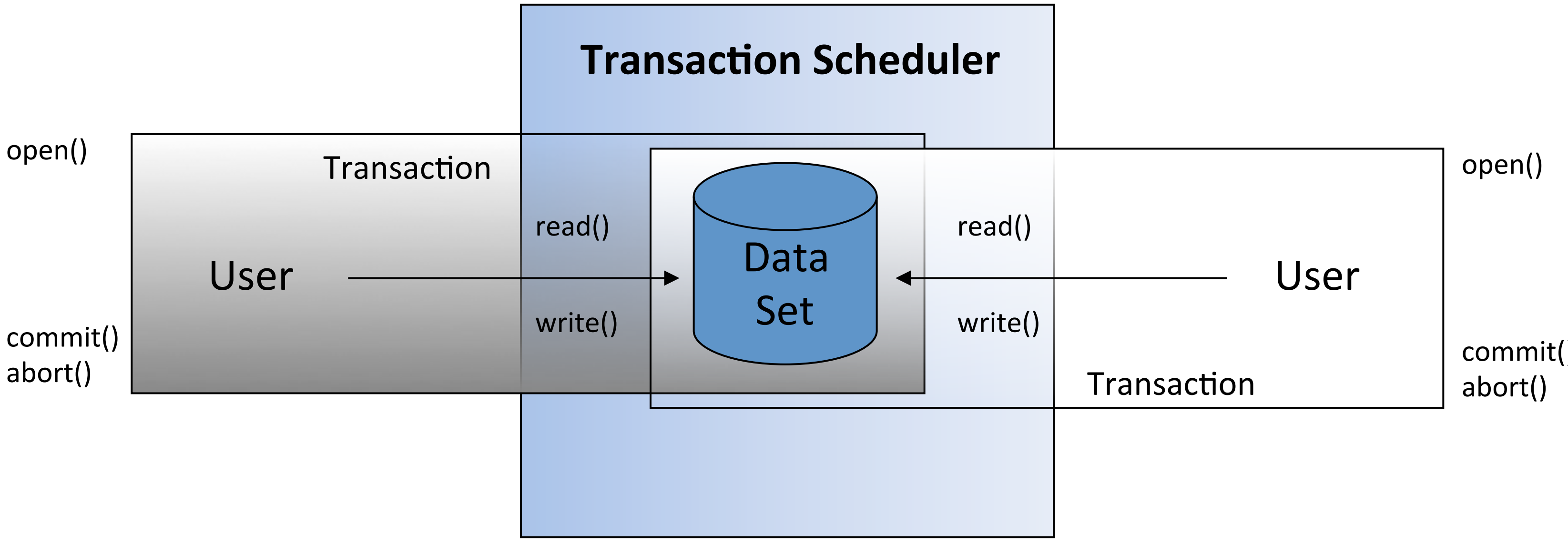
Solving Concurrency Problems

- Concurrency Problems occur due to a lack of isolation between concurrent transactions
- Problems in case of commit
  - Lost Update: Transactions may overwrite each others' updates
  - Inconsistent Retrieval: Transactions base their calculations on retrieved data that is not yet committed by other transactions ("dirty read")
- Problems in case of abort
  - Uncommitted Dependencies, Cascading Aborts
  - Premature writes

Concurrency Control

- Concurrency control enforces a particular order of operations of concurrent transactions
  - It creates a ***Transaction schedule***
- The mechanisms commonly employed for concurrency control are
  - Locking:
    - Reserve a data object for exclusive access by a single transaction
    - Most practical systems use locking
  - Optimistic concurrency control
  - Timestamp ordering
- How do we know that such a concurrency control mechanism is "correct" ?
- How do we know that the enforced order of operations represents a "correct" transaction schedule?

What is a correct Transaction Schedule?

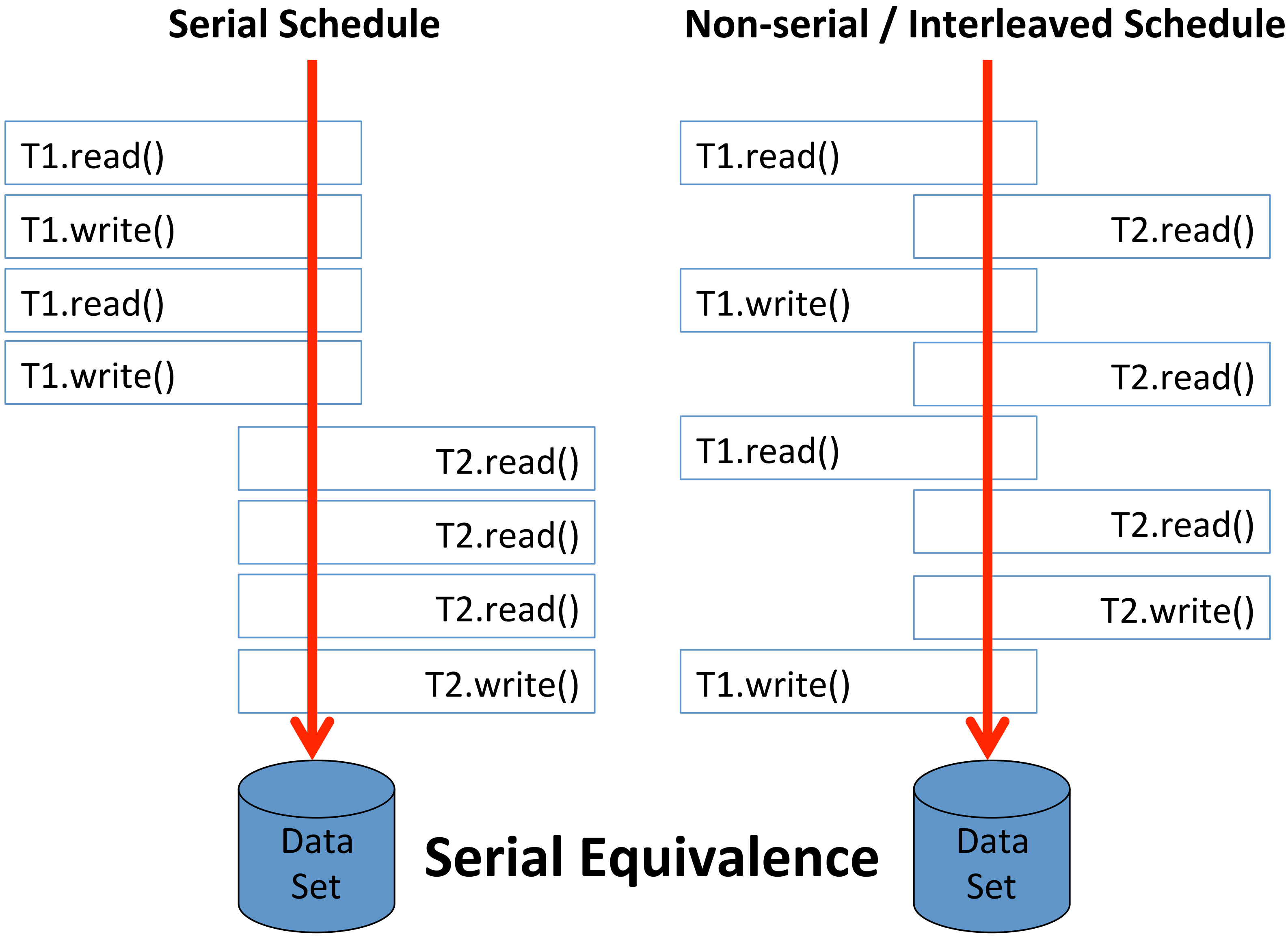


- A "schedule" is a sequence of operations from different transactions – transactions operate in an interleaved fashion
  - Such a schedule may compromise the integrity / consistency of a database

Serialised Transaction Schedules

- Radical solution
  - Completely serialised execution of transactions
  - In order to avoid the concurrency problems described, one obvious solution would be to schedule only one transaction at a time for execution
- Such a completely "serialised" schedule can be regarded as "correct":
  - Enforces Isolation: transactions are completely isolated and cannot interfere with each other
  - Enforces consistency: Serial execution of transactions never leaves a database in an inconsistent state
- However: we want a concurrent execution of transactions, that preserves the ACID principles

Transaction Schedules



Serial Equivalence, Serialisability

- We try to find a non-serial schedule that is ***equivalent*** to a corresponding serial schedule:
  - A non-serial schedule over a set of transactions is correct if it produces the same results as a completely serial execution of the same set of transactions
- If we can show or guarantee serialisability of a transaction schedule, then we can guarantee consistency of data manipulation

Serial Equivalence

- Definition: Serial Equivalence

Two or more transactions are serial equivalent, if they produce the same result operating in an interleaved fashion, as if they would operate in a completely serialized fashion.

- Serial Equivalence is used as a design criterion for concurrency control protocols!

Serialisability

- What makes a non-serial schedule equivalent to a serial schedule?
- Observation:
  - It is the sequence and order of **read / write** operations (of different transactions) in a schedule that determines whether the schedule is serialisable
- A concurrency control protocol must enforce such a correct sequencing!

Consistency: Schedule T1 before T2

- We examine the following example schedule:
  - T1: transfer of money between accounts
  - T2: reads balance of both accounts and prints out sum

Schedule

In this example, the sum of both accounts remains the same

T1

```
acc1 = read( account_a )
write( account_a, acc1 - 100 )
acc2 = read( account_b )
write( account_b, acc2 + 100 )
```

T2

```
balance1 = read( account_a )
balance2 = read( account_b )
print( balance1 + balance2 )
```

Schedule OK, print consistent state of DB

Consistency: Schedule T2 before T1

- The order in which serialised transactions are done is unimportant, consistency is maintained

Schedule

T2

```
balance1 = read( account_a )
balance2 = read( account_b )
print( balance1 + balance2 )
```

T1

```
acc1 = read( account_a )
write( account_a, acc1 - 100 )
acc2 = read( account_b )
write( account_b, acc2 + 100 )
```

The sum of both accounts remains the same

Schedule OK , print consistent state of DB

Schedule T1 and T2 interleaved

- This schedule with interleaved transaction is not serial equivalent!
- The print result of transaction T2 shows a result that is different to that in the serialised schedule !

Schedule

```
T1: acc1 = read( account_a )
T1: write( account_a, acc1 - 100 )
                    T2: balance1 = read( account_a )
                    T2: balance2 = read( account_b )
                    T2: print( balance1 + balance2 )
T1: acc2 = read( account_b )
T1: write( account_b, acc2 + 100 )
```

T2 prints before T1 finishes – differs from serial schedule

Error !!

Schedule T1 and T2 interleaved

- This schedule with interleaved transaction is not serial equivalent!
- The print result of transaction T2 shows a result that is different to that in the serialised schedule !

Schedule

```
T1: balance1 = read( account_a )
T1: write( account_a, balance1 - 100 )
                        T2: balance1 = read( account_a )
                        T2: balance2 = read( account_b )
T1: balance2 = read( account_b )
T1: write( account_b, balance2 + 100 )
```

T2 prints after T1 finishes , differs from serial schedule: balance2 has old value

Error !!

```
                        T2: print( balance1 + balance2 )
```

Serialisability Condition Conflict between Operations

- **Serialisability can be defined in terms of operation conflicts**:

For two transactions to be serialisable, it is **necessary and sufficient** that ***all pairs of conflicting operations*** of two transactions are executed in the **same order** on all data objects accessed by these operations

- A pair of operations conflict, if their combined effect depends on the order in which they are executed

Serialisability
Conflict between Operations

- Serialisability of a schedule – Read operations:
  - If there are two subsequent read operations in a schedule, e.g.:

    S = { ... , T1.read(x), T2.read(x), ... }

    then the order is not important / does not influence the overall outcome
- Read operations of different transactions are **not** in conflict

Serialisability
Conflict between Operations

- Serialisability of a schedule – Read and write operations:
  - If there two subsequent operations from different transactions in a schedule, a read and a write operation, e.g.:

    S = { ..., T1.read(x), T2.write(x), ... }

    or two subsequent write operations, e.g.:

    S = { ..., T1.write(x), T2.write(x), ... }

    then the order **is** important and influences the overall outcome
- **Read** and **write** operations of different transactions **are in conflict**
- **Write** operations of different transactions are in conflict

Serialisability Condition
Pairs of conflicting Operations

Operations of different Transactions		Conflict	Reason
read	read	No	Because the effect of a pair of read operations does not depend on the order in which they occur
read	write	Yes	Because the effect of a read and a write operation depends on the order of their execution
write	write	Yes	Because the effect of a pair of write operations depends on the order of their execution

T1 and T2 interleaved, observing the Serialisability Condition

- We can interleave both transactions as long as we comply with the serialisability condition
  - In each transaction, **read** and **write** operations on "account_a" and "account_b" are in the same order

Schedule

```
T1: balance1 = read( account_a )
T1: write( account_a, balance1 - 100 )
                    T2: balance1 = read( account_a )
T1: balance2 = read( account_b )
T1: write( account_b, balance2 + 100 )
                    T2: balance2 = read( account_b )
                    T2: print( balance1 + balance2 )
```

T1 and T2 interleaved, observing the Serialisability Condition

- We can interleave both transactions as long as we comply with the serialisability condition
  - In each transaction, **read** and **write** operations on “account_a” and “account_b” are in the same order

Schedule

S = { T1.read(balance1, account_a),
T1.write(account_a, balance1 - 100),
T2.read(b1, account_a),
T1.read(balance2, account_b),
T1.write(account_b, balance2 + 100),
T2.read(b2, account_b),
T2.print (b1 + b2) ← **correct**
}

Conflicting pair of operations

Conflicting pair of operations

T1 and T2 interleaved, observing the Serialisability Condition

- Alternative schedule, same result
  - Again, in each transaction, **read** and **write** operations on "account_a" and "account_b" are in the same order

Schedule

S = { T1.read (balance1, account_a),
T2.read (b1, account_a),
T1.write (account_a, balance1 - 100),
T1.read (balance2, account_b),
T2.read (b2, account_b),
T1.write (account_b, balance2 + 100),
T2.print (b1 + b2)
}

Conflicting pair of operations

Conflicting pair of operations

Correct, based on previous state of database

The Serialisability Condition

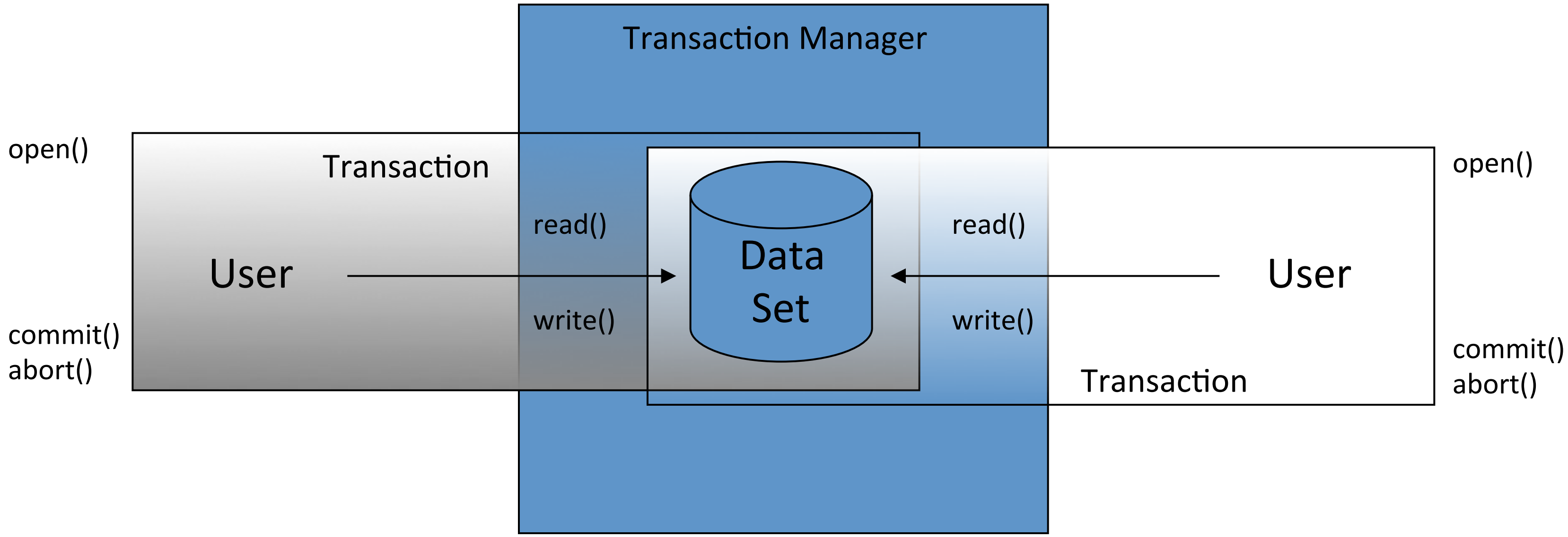
Definition based on conflicts between operations

*The **necessary** and **sufficient** condition for two transactions to be serialised is that the order of invocation of all conflicting pairs of operations are the same for all the objects which are invoked by both transactions.*

[Weihl, 1989]

Concurrency Control Techniques

Concurrent Transactions



- Guarantee ***Isolation***:
  - avoid interference between transactions that concurrently access / manipulate data sets

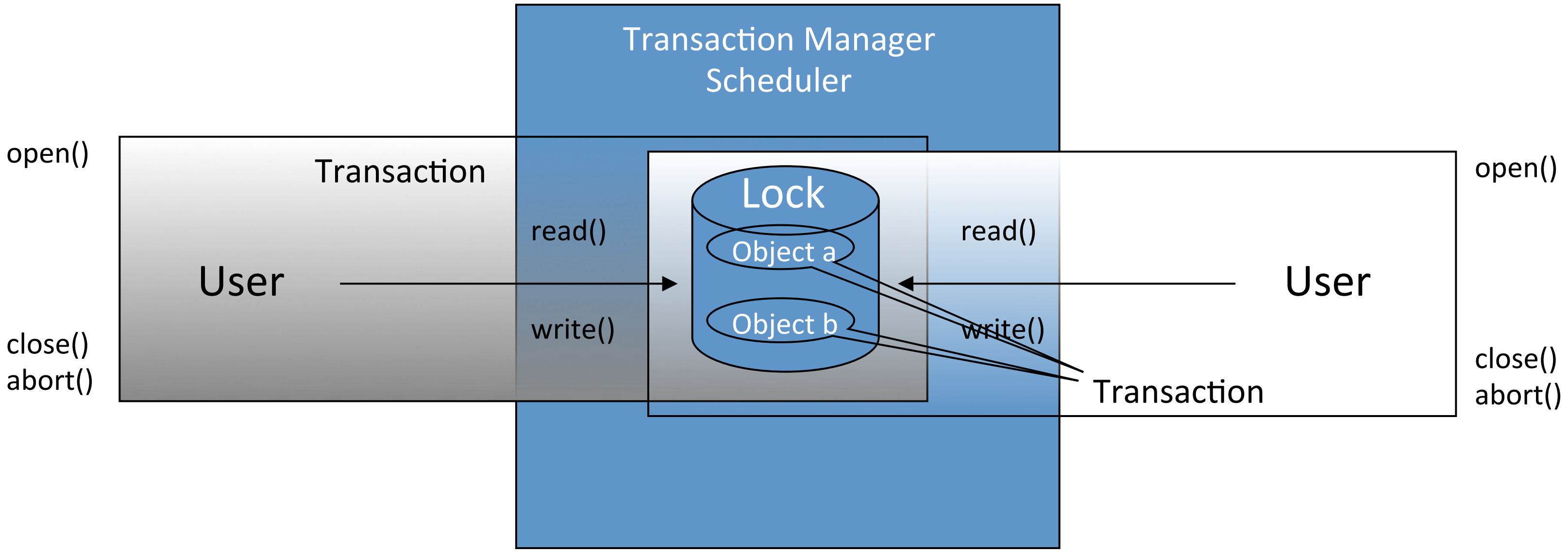
Concurrency Control Protocols

- We enforce isolation with concurrency control protocols
- Two kinds of concurrency control behaviour
  - Transactions wait to avoid conflicts (pessimistic concurrency control)
  - Transactions are restarted after conflicts have been detected (optimistic concurrency control)
- Methods
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering

Concurrency Control

- The mechanisms commonly employed for concurrency control are
  - Locking:
    - Reserve a data object for exclusive access by a single transaction
    - Most practical systems use locking
  - Optimistic concurrency control
  - Timestamp ordering

Simple Locking



- A transaction management system typically manages objects with locks
- The transaction scheduler will perform lock() and unlock() operations on objects under its authority, on behalf of transactions

Simple Exclusive Locking

- If a transaction wants to read / write objects, it establishes locks for these objects
  - The server sets a lock on each object for a particular transaction before it is accessed (read or write operations)
- While an object is locked
  - only the transaction holding the lock can read and/or manipulate this object
  - Other transaction have to wait for the lock to be released or may be able to share a lock (this depends on the locking concepts used)

Serialisability of Schedules with Locks

- The introduction of locks reserves data objects for exclusive read / write operations of one transaction
- Does this guarantee **serialisability**?

Serialisability with Locks

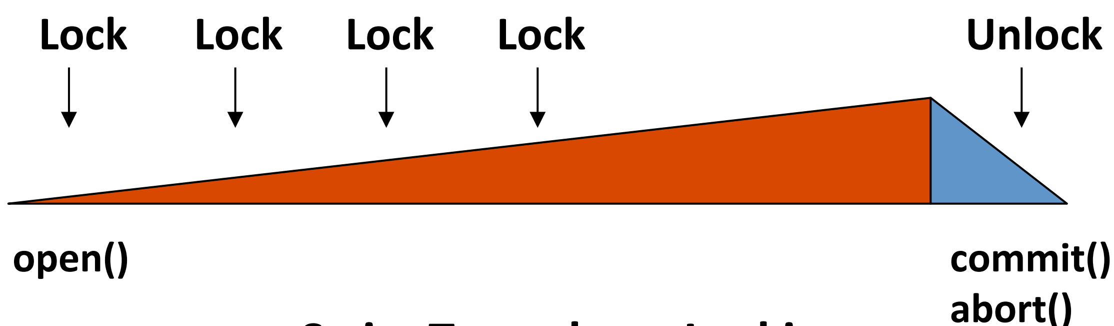
- This is not a serialisable schedule – unlock of object A too early !!

Time	Transaction T1	Transaction T2
t1	`open_transaction()`	
t2	**`lock(A)`**	`open_transaction()`
T3	**`read( A, balance_a )`**	**`lock(A)`** WAIT
T4	**`write( A, balance_A - 100 )`**	
t5	**`unlock(A)`**	**`read( A, balance_A )`**
t6	`lock(B)`	**`write( A, balance_A - 10 )`**
t7	**`read( B, balance_B )`**	**`unlock(A)`**
t8	**`write( B, balance_B + 100 )`**	`commit()`
t9	**`unlock(B)`**	
t10	`commit()`	

Error !!

Two-phase Locking (2PL)

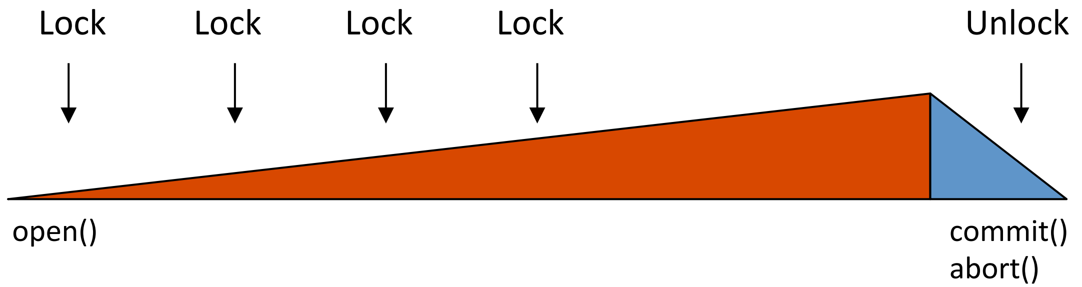
- A transaction follows the two-phase **locking protocol**, if all locking operations precede the first unlock operation in the transaction
  - All locking operation occur first during the course of the transaction
  - No alternating lock / unlock operations!
- Strict two-phase locking
  - All unlock operations occur during the final commit / abort



Strict Two-phase Locking

Strict Two-Phase Locking

Shared Read Locks



- When an object is accessed within a transaction:
  a) If the object is not already locked, it is locked and the transaction proceeds
  b) If the object has a conflicting lock by some other transaction, the transaction waits until the object is unlocked
  c) If the object has a non-conflicting lock set by another transaction, the transaction shares the lock and proceeds
  d) If the object has already been locked for read in the transaction, and no other transaction shares this read lock, the lock can be promoted if necessary and the transaction proceeds (if promotion is prevented by conflict, rule b) is used)
- The server unlocks all objects it locked for the transaction at commit / abort

2-Phase Locking

- Serialisability of this schedule is guaranteed because we observe the so-called 2-phase locking protocol
  - Unlock only happens at the end of the transaction !

Time	Transaction T1	Transaction T2
t1	open_transaction()	
t2	**lock(A)**	open_transaction()
t3	**read(A, balance_a)**	lock(A)
	write(A, balance_A – 100)	**WAIT**
t4	**lock(B)**	
	read(B, balance_B)	
t5	**write(B, balance_B + 100)**	
t6	commit(**unlock(A,B)**)	**read(A, balance_A)**
		write(A, balance_A - 10)
		commit(**unlock(A)**)

Simple Exclusive Lock

- This simple strategy guarantees serializability
  - The order of all conflicting pairs of operations is the same, because a conflicting transaction cannot proceed while it does not hold locks on objects it needs
- Important!
  - **Strict two-phase locking**
- Problem
  - This is not the most efficient strategy; it unnecessarily restricts access to shared resources
  - For example, two transactions that simply wish to perform a read() operation on an object would not interfere – allow shared read !

Shared *read* Locks

- Concurrency of transactions can be improved by distinguishing between read and write locks
  - Before a read() is performed, a ***read*** lock is set on an object
  - Before a write() is performed, a ***write*** lock is set
- ***read*** locks are also called *shared* locks
  - Each transaction can set a read lock on an object that is either unlocked or already has a read lock from another transaction – transactions "share" read locks
- ***read*** locks saveguard against ***write*** locks
  - guarantee that the object remains readable but no other transaction can set a ***write*** lock and make inconsistent updates

Shared *read* Locks

- Conflict rules
  - If transaction T has set a ***read*** lock then transaction U can set a ***read*** lock as well
  - If transaction T has already set a ***read*** lock then transaction U is **not** allowed to set a ***write*** lock until T commits / aborts
  - If transaction T has already set a ***write*** lock then transaction U is **not** allowed to set either ***read*** or ***write*** locks (***exclusive lock***)

Lock Compatibility

		Transaction T2 requests Lock	
		read	write
Transaction T1 Lock already set on object	none	OK	OK
	read	OK	Wait
	write	Wait	Wait

- The fact that a lock has been set on an object by transaction T1 does not necessarily mean that transaction T2 cannot also obtain a lock
- The lock may be *shared* if they are both *read* locks
- Lock Promotion – make a lock more exclusive
  - A read lock can be "promoted" to a write lock if it is **not** shared by other transactions

Read / Write Locks
Lock Promotion

- Transaction T

openTransaction:
Read-Lock b:
bal = b.getBalance()
Write-Lock b:
*b.setBalance(bal * 1.1)*
Write-Lock a:
a.withdraw(bal / 10)
Unlock a,b
closeTransaction

- Transaction U

openTransaction:
Read-Lock b: ***wait for T to unlock b***

wait

Read-Lock b:
bal = b.getBalance()
Write-Lock b:
*b.setBalance(bal * 1.1)*
Write-Lock a:
a.withdraw(bal / 10)
Unlock a,b
closeTransaction

Lock Promotion !

Read lock becomes more exclusive Write lock

Avoid Problems with Locks Inconsistent Retrieval

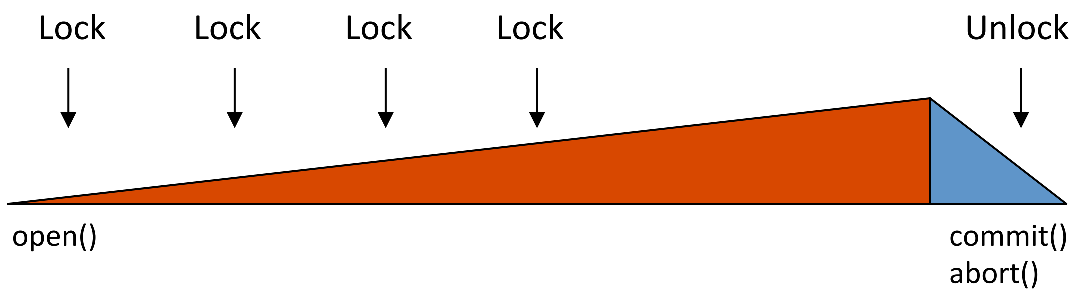
- Cause:
  - conflict between write() and read() operations of two transactions
- Is prevented by serialising with a lock
  - the read() lock of transaction X occurs before or after the write() lock of transaction Y
  - Locks are released at transaction commit
- Lock conditions:
  - If the read lock is set first in transaction X, the write lock cannot be set in transaction Y
    - Y can set a read lock, Y cannot set a write lock
    - no lock promotion possible, as there is a read lock from X
  - If the write lock is set first in transaction X, transaction Y cannot even set a read lock (Y is delayed, until X commits)
- Important !!
  - Locks must not be released (or demoted) during the transaction, only at commit/abort
  - “Strict two-phase Locking”

Avoid Problems with Locks
Lost Update

- Cause:
  - concurrent write() of transaction X overwrites write() action of transaction Y
- Is prevented by making later transactions delay their read(), until earlier transactions are completed
  - Transaction X sets first a read lock to read an object and promotes this lock to a write lock to write the same object
  - Transaction Y is delayed with its request for a read lock until transaction X is finished
- Important !!
  - Locks must not be released (or demoted) during the transaction, only at commit/abort
  - "Strict two-phase Locking"

Strict Two-Phase Locking

Shared Read Locks



- When an object is accessed within a transaction:
  a) If the object is not already locked, it is locked and the transaction proceeds
  b) If the object has a conflicting lock by some other transaction, the transaction waits until the object is unlocked
  c) If the object has a non-conflicting lock set by another transaction, the transaction shares the lock and proceeds
  d) If the object has already been locked for read in the transaction, and no other transaction shares this read lock, the lock can be promoted if necessary and the transaction proceeds (if promotion is prevented by conflict, rule b) is used)
- The server unlocks all objects it locked for the transaction at commit / abort

Simple Exclusive Locking

- If a transaction wants to read / write objects, it establishes locks for these objects
  - The server sets a lock on each object for a particular transaction before it is accessed (read or write operations)
  - The server removes these locks when the transaction is finished and not immediately after the operation – exclusive locking
- While an object is locked
  - only the transaction holding the lock can access/manipulate this object
  - Other transaction have to wait for the lock to be released or may be able to share a lock (this depends on the locking concepts used)
- The use of locks can lead to deadlocks !

Deadlocks

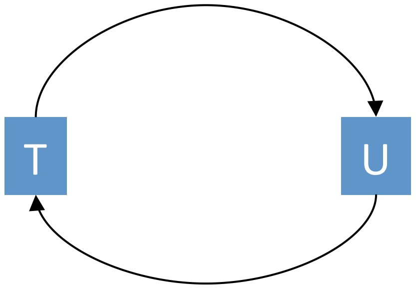
- The use of locks may lead to deadlocks
- Definiton
  - A deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- The scheduler is responsible for detecting and breaking the deadlock
  - Deadlocks may be broken by simply aborting one of the transactions involved
- How do you choose which transaction to abort?
  - Abort the oldest
  - Abort depending on the complexity of the transactions
- Rather than detecting deadlock (an overhead), you could just use timeouts, but how long should the timeout be?

Example: Deadlock with Write Locks

Transaction T		**Transaction U**	
a.deposit (100)	Acquire write lock on a		
		b.deposit (200)	Acquire write lock on b
b.withdraw (100)	Wait for lock on b to be released by U		
		a.withdraw (200)	Wait for lock on a to be released by T
Wait ….. Wait …..			
		Wait ….. Wait …..	

Wait-for Graph

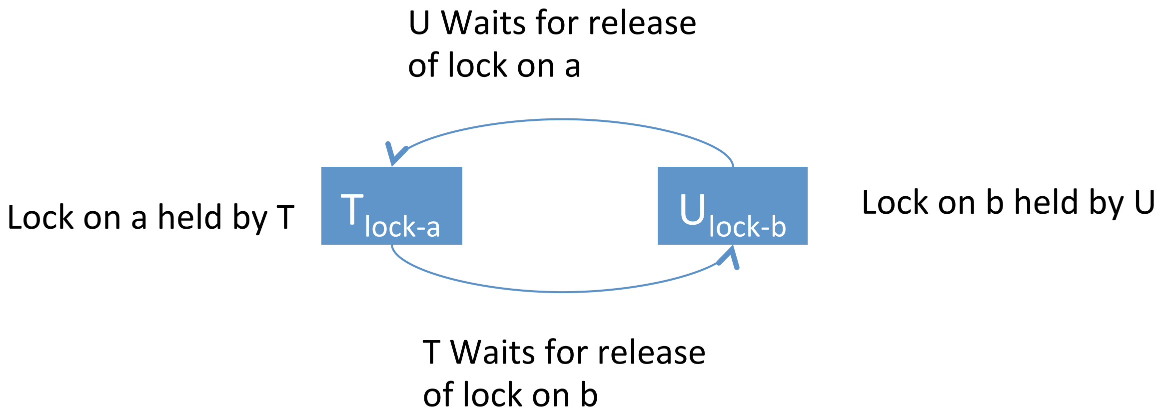
- Representation of a deadlock situation:
  - Nodes in this graph represent transactions
  - Edges between nodes represent wait-for relationships between current transactions
  - The dependency between transactions is indirect – via a dependency on objects
  - A cycle in the graph indicates a deadlock:



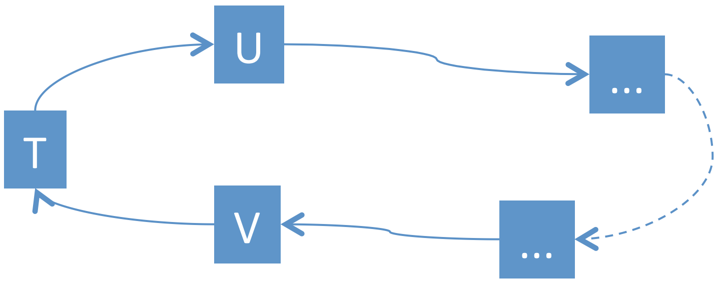
  - Transaction T and U wait for each other because of a lock on one object
  - None of these locks can ever be released
  - One of the transactions would have to be aborted to break this deadlock

Wait-for Graph

a)

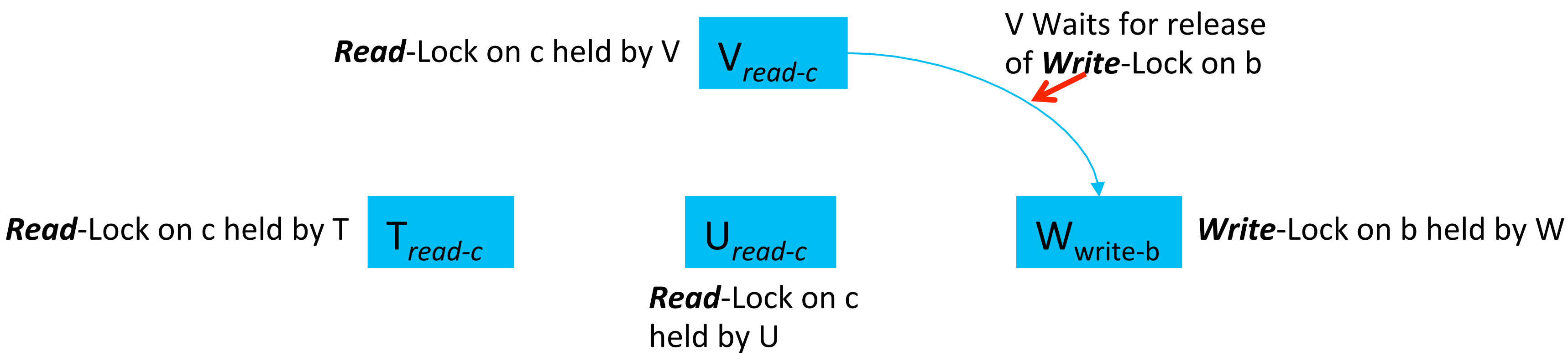


b)



- Note: in case b), a cycle T --> U --> … --> V --> T exists – all these transactions are blocked waiting for locks

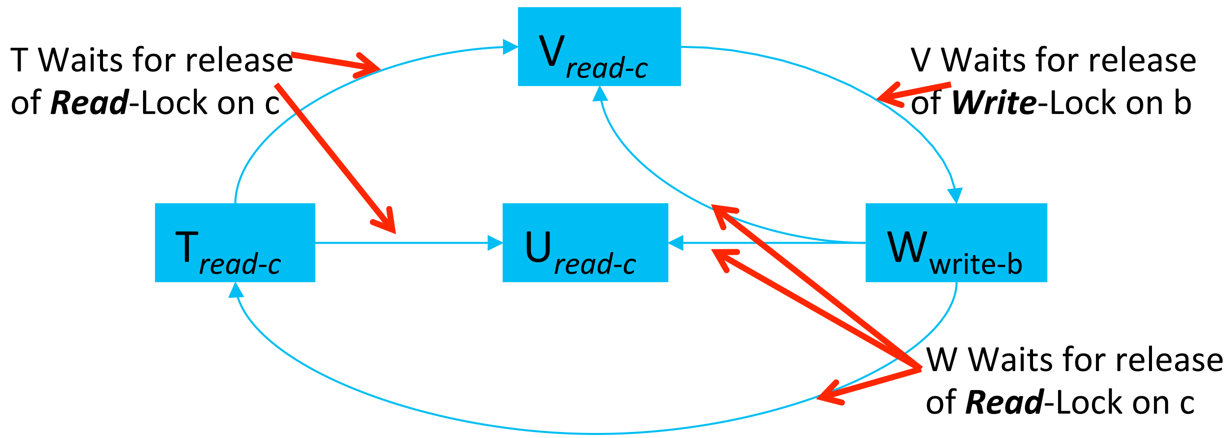
Deadlock with Read and Write Locks



- Transactions T, U, V share a Read-Lock on object c
- Transaction W holds a Write-Lock on b
- Transaction V tries to obtain a Read-Lock on b, waits for the Write-Lock to be released by W (see wait-for graph a))
- Scenario:
  - T wants a Write-Lock on c
  - W wants a Write-Lock on c

Deadlock with Read and Write Locks

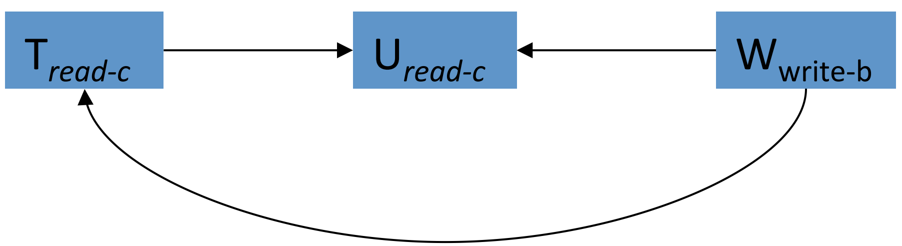
- T, U, V share a read lock on object c
- W holds a write lock on object b
- Scenario: Transactions T and W request a write lock on object c
  - A dead lock arises: T cannot promote its lock on c to a Write-Lock, because U and V still hold a Read-Lock: T waits-for U,V
  - V is waiting to obtain a read lock on object b, waits for W to release write lock – DEADLOCK – W is waiting for V to release its Read-Lock
  - W waits to obtain write lock on object c, waits for T, U, V to release read lock, W cannot set a Write-Lock on object c, because T,U,V hold read locks
  - T waits to obtain write lock on object c, waits for U, V to release read lock



  - Wait-for graph shows the waiting cycles between transactions – in the example, there are two of them: <V --> W -->T-->V> and <V-->W-->V>

Break Deadlock

- Wait-for Graph shows how to break this deadlock
  - V is in both waiting cycles
  - If V is aborted, then its read lock on object c is released
  - U is not waiting for anything, therefore we assume that it ends naturally and releases the read lock on object c
  - T can now promote its read lock on c to a write lock, it can now perform its write(), end naturally and release write lock on c
  - W can now obtain the write lock on object c



  - What else can we do to resolve / avoid deadlocks?

Handling Deadlocks

- Deadlock Detection
  - Use the Wait-for graph to identify cycles and select transactions to be aborted
    - Select the oldest transactions
    - Select the transaction involved in most of the cycles
    - Select according to their complexity
- Deadlock Prevention
  - Lock all objects at the very beginning of a transaction in one atomic action
  - Problem
    - Reduced concurrency: unnecessary access restriction to shared resources
    - It must be know in advance which objects are manipulated --> this is impossible in interactive applications

Handling Deadlocks

- Timeouts
  - Each lock is given a period of time where it is invulnerable
  - After this timeout, it becomes vulnerable
  - If transaction X holds a lock that becomes vulnerable and transaction Y is waiting for X, then X is aborted
  - Problem
    - Hard to decide on an appropriate length of timeout
    - Transactions may be aborted, when the lock becomes vulnerable and another transaction waits, but there is no deadlock

Improving Concurrency

- Locks serialise access to date objects and guarantee consistency, but transactions have to wait
- This counteracts the goal of fast, concurrent access to date by multiple users
- Alternative
  – Use an optimistic concurrency scheme --> "act first, look later" – e.g.: Two-version locking