Electronics and Computer Science

UNIVERSITY OF Southampton

Transactions and Concurrency

COMP3211 Advanced Databases

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk
2016-2017

Overview

- Transaction processing
- Transaction problems
- Transaction lifecycle
- ACID
- Schedules and serialisability
- Locking (including 2PL)
- Timestamps

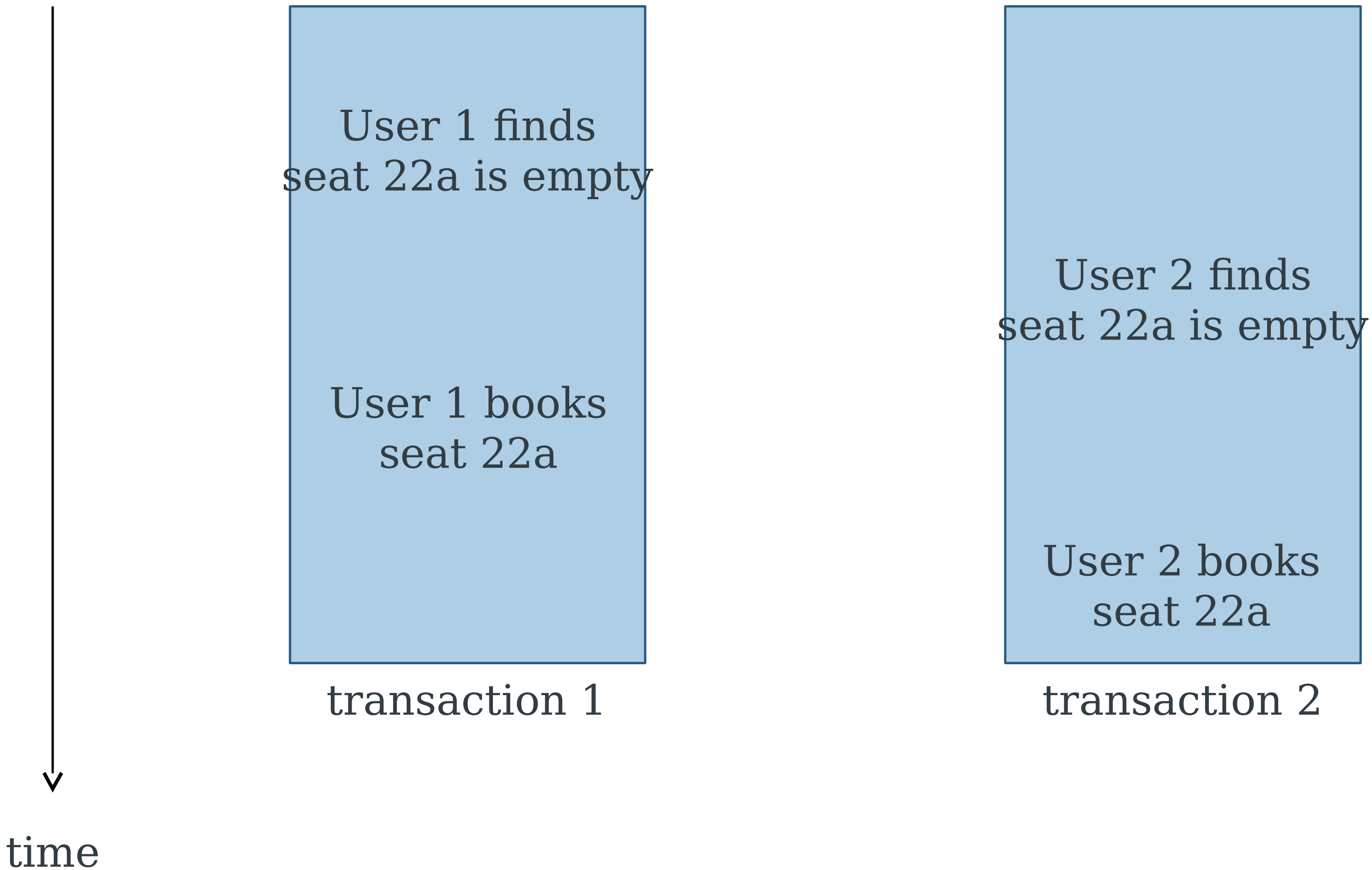
Concurrency

- In a multi-user DBMS, many users may use the system concurrently
- Stored data items may be accessed concurrently by user programs

Transaction: a logical unit of work that changes the contents of a database

- Group of database operations that are to be executed together

When updates go wrong, part one



Serial versus Serialisable

In an ideal world, we would run transactions **serially**

- Transactions runs one at a time, with no overlap

In practice, some parallelism is required

- Too many transactions for serial execution!

Transactions should be **serialisable**

- Should behave as if they were serial, but may be executed concurrently

When updates go wrong, part two

Add £100 to account 123

CRASH!

Subtract £100 from account 456

time

Atomicity

System failure partway through a transaction may leave the database in an inconsistent state

Transactions are **atomic**: operations within a transaction should either all be executed successfully or not be executed at all

Transaction Problems

Basic database access operations

read(X)

Reads a database item X_d into a program variable X_T in transaction T

write(X)

Writes the value of program variable X_T in transaction T into the database item X_d

Example Transactions

T1	**T2**
read(X)	read(X)
X := X – 10	X := X + 5
write(X)	write(X)
read(Y)	
Y := Y+10	
write(Y)	

Initial values: X=20, Y=50

Final values: X=15, Y=60

Concurrency

- Understanding transactions is important for concurrency
- Operations within a transaction may be interleaved with those from another transaction
- Depending on how operations are interleaved, database items may have incorrect values

The Lost Update Problem

Two transactions have operations interleaved so that some DB items are incorrect

The Lost Update Problem

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}		X_d	Y_d			
							20	50			
read(X)				**20**					20	50	
X := X – 10					**10**					20	50
	read(X)		10		**20**			20	50		
	x := X + 5		10		**25**			20	50		
write(X)				10		25			**10**	50	
read(Y)				10	**50**	25			10	50	
	write(X)		10	50	25			**25**	50		
Y := Y+10				10	**60**	25			25	50	
write(Y)				10	60	25			25	**60**	

The Temporary Update (Dirty Read) Problem

One transaction updates a DB item and then fails.
Item is accessed before reverting to original value.

The Temporary Update (Dirty Read) Problem

T1	**T2**	$\mathbf{X_{T1}}$	$\mathbf{Y_{T1}}$	$\mathbf{X_{T2}}$	$\mathbf{Y_{T2}}$		$\mathbf{X_d}$	$\mathbf{Y_d}$			
							20	50			
read(X)				**20**					20	50	
X := X – 10					**10**					20	50
write(X)				10					**10**	50	
	read(X)		10		**10**			10	50		
	x := X + 5		10		**15**			10	50		
	write(X)		10		15			**15**	50		
read(Y)				10	**50**	15			15	50	
CRASH!											
rollback									**20**	50	

The Incorrect Summary Problem

One transaction calculates an aggregate summary function on multiple records while other transactions update records

Aggregate function may read some values before they are updated, and some after

The Incorrect Summary Problem

T1	T2	X_{T1}	Y_{T1}	S	X_{T2}	Y_{T2}	X_d	Y_d
							20	50
S := 0				**0**			20	50
	read(X)			0	**20**		20	50
	X := X – 10			0	**10**		20	50
	write(X)			0	10		**10**	50
read(X)		**10**		0	10		10	50
S := S + X		10		**10**	10		10	50
read(Y)		10	**50**	10	10		10	50
S := S + Y		10	50	**60**	10		10	50
	read(Y)	10	50	60	10	**50**	10	50
	Y := Y + 10	10	50	60	10	**60**	10	50
	write(Y)	10	50	60	10	60	10	**60**

The Unrepeatable Read Problem

One transaction reads an item twice, while another changes the item between the two reads

T1:	**T2:**
read(X)	
	read(X)
	X := X – 10
	write(X)
read(X)	

Transaction Processing

When a transaction is submitted for execution, the system must ensure that:

- All operations in the transaction are completed successfully, with effect recorded permanently in the database, or
- There is no effect on the database or other transactions

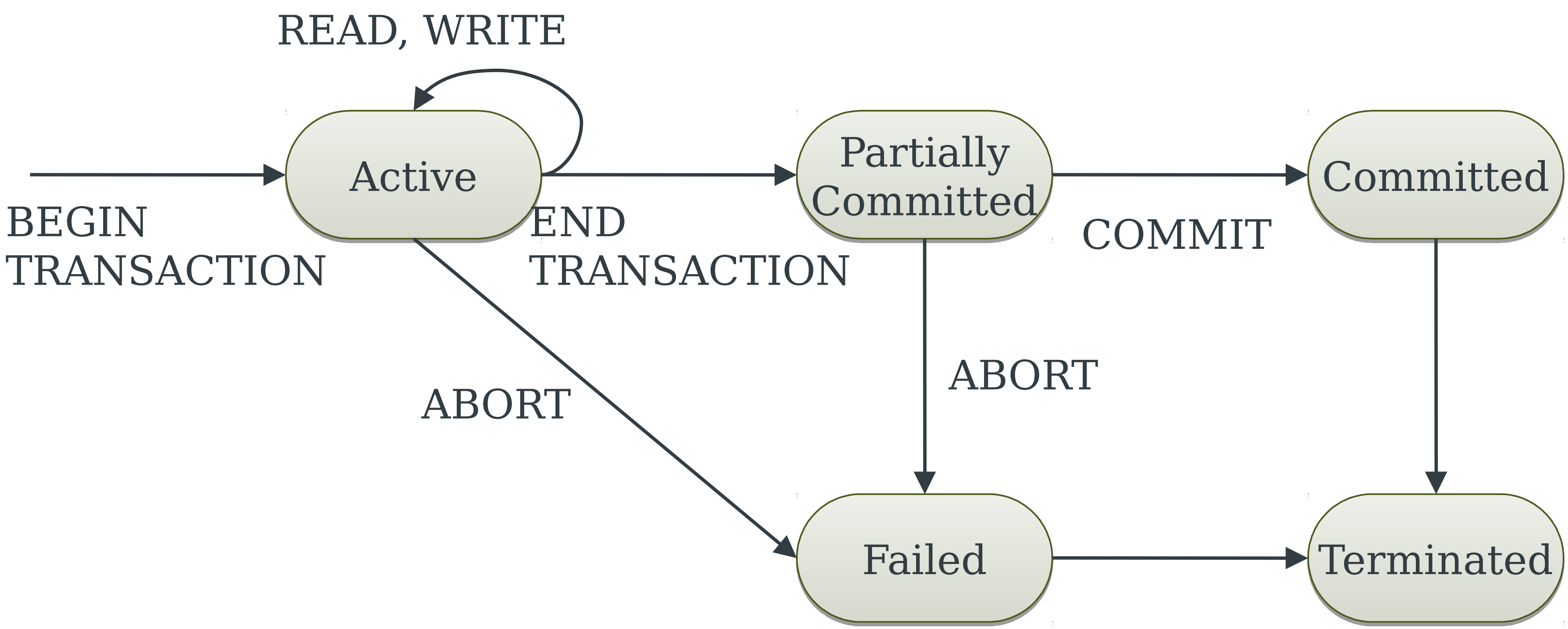
Transactions may be **read-only** or **update**

Transaction Life Cycle

Need to track start and end of transactions, and commit and abort of transactions

- BEGIN_TRANSACTION
- READ, WRITE
- END_TRANSACTION
- COMMIT_TRANSACTION
- ROLLBACK (or ABORT)

Transaction Life Cycle



ACID

ACID Properties

Atomicity

- A transaction is atomic and is either performed completely or not at all

Consistency preservation

- Correct transaction execution must take the database from one consistent state to another

Isolation

- A transaction should not make updates externally visible (to other transactions) until committed

Durability (permanence)

- Once database is changed and committed, changes should not be lost because of failure

Schedules

A **schedule** S of n transactions is an ordering of the operations of the transactions, subject to the constraint that for each transaction T that participates in S, the operations in T must appear in the same order in S that they do in T

Two operations in a schedule are **conflicting** if:

- They belong to different transactions and
- They access the same data item and
- At least one of the operations is a write()

Serial and Serialisable

A schedule is **serial** if, for each transaction T in the schedule, all operations in T are executed consecutively (no interleaving), otherwise it is **non-serial**

A schedule S of n transactions is **serialisable** if it is equivalent to some serial schedule of the same n transactions

Schedule Equivalence

Two schedules are **result equivalent** if they produce the same final state on the database

Two schedules are **conflict equivalent** if the order of any two conflicting operations is the same in both schedules

Serial Schedule T1;T2

T1	**T2**	$\mathbf{X_{T1}}$	$\mathbf{Y_{T1}}$	$\mathbf{X_{T2}}$	$\mathbf{Y_{T2}}$		$\mathbf{X_d}$	$\mathbf{Y_d}$			
							20	50			
read(X)				**20**					20	50	
X := X – 10					**10**					20	50
write(X)				10					**10**	50	
read(Y)				10	**50**				10	50	
Y := Y + 10					10	**60**				10	50
write(Y)				10	60				10	**60**	
	read(X)		10	60	**10**			10	60		
	X := X + 5		10	60	**15**			10	60		
	write(X)		10	60	15			**15**	60		

Serial Schedule T2;T1

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
	read(X)			**20**		20	50
	X := X + 5			**25**		20	50
	write(X)			25		**25**	50
read(X)		**25**		25		25	50
X := X – 10		**15**		25		25	50
write(X)		15		25		**15**	50
read(Y)		15	**50**	25		15	50
Y := Y + 10		15	**60**	25		15	50
write(Y)		15	60	25		15	**60**

Non-Serial and Non-Serialisable Schedule

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
read(X)		**20**				20	50
X := X – 10		**10**				20	50
	read(X)	10		**20**		20	50
	x := X + 5	10		**25**		20	50
write(X)		10		25		**10**	50
read(Y)		10	**50**	25		10	50
	write(X)	10	50	25		**25**	50
Y := Y+10		10	**60**	25		25	50
write(Y)		10	60	25		25	**60**

Non-Serial but Serialisable Schedule

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}		X_d	Y_d			
							20	50			
read(X)				**20**					20	50	
X := X – 10					**10**					20	50
write(X)				10					**10**	50	
	read(X)		10		**10**			10	50		
	X := X + 5		10		**15**			10	50		
	write(X)		10		15			**15**	50		
read(Y)				10	**50**	15			15	50	
Y := Y + 10					10	**60**	15			15	50
write(Y)				10	60	15			15	**60**	

Locking

Locking

Locks are used to synchronise access by concurrent transactions to a database

Typically, two lock modes: **shared** and **exclusive**

- Shared: for reading
- Exclusive: for writing

Binary locks (equivalent to exclusive mode only) are also possible, but generally too restrictive

Lock Operations

lock-shared(X)

Attempt to acquire a shared lock on X

lock-exclusive(X)

Attempt to acquire an exclusive lock on X

unlock(X)

Relinquish all locks on X

Lock Outcome

The result of an attempt to obtain a lock is either:

- Grant lock (able to access the item)
- Wait for lock to be granted (not yet able to access the item)
- (Abort)

		Lock Requested	
		Shared	Exclusive
Lock held in mode	Shared	Grant	Wait
	Exclusive	Wait	Wait

Locking Rules

1. Must issue lock-shared(X) or lock-exclusive(X) before a read(X) operation
2. Must issue lock-exclusive(X) before a write(X) operation
3. Must issue unlock(X) after all read(X) and write(X) operations are completed
4. Cannot issue lock-shared(X) if already holding a lock on X
5. Cannot issue lock-exclusive(X) if already holding a lock on X
6. Cannot issue unlock(X) unless holding a lock on X

Lock Conversion

Rules 4 and 5 may be relaxed in order to allow lock conversion

- A lock-shared(X) may be *upgraded* to a lock-exclusive(X)
- A lock-exclusive(X) may be *downgraded* to a lock-shared(X)

Locking Example

T1:	**T2:**
lock-shared(Y)	lock-shared(X)
read(Y)	read(X)
unlock(Y)	unlock(X)
lock-exclusive(X)	lock-exclusive(Y)
read(X)	read(Y)
X := X + Y	Y := Y + X
write(X)	write(Y)
unlock(X)	unlock(Y)

Locking Example

Two possible serial schedules:

- T1;T2
- T2;T1

Take X=20 and Y=50 as initial values

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
lock-shared(Y)						20	50
read(Y)			**50**			20	50
unlock(Y)			50			20	50
lock-exclusive(X)			50			20	50
read(X)		**20**	50			20	50
X := X + Y		**70**	50			20	50
write(X)		70	50			**70**	50
unlock(X)		70	50			70	50
	lock-shared(X)	70	50			70	50
	read(X)	70	50	**70**		70	50
	unlock(X)	70	50	70		70	50
	lock-exclusive(Y)	70	50	70		70	50
	read(Y)	70	50	70	**50**	70	50
	Y := Y + X	70	50	70	**120**	70	50
	write(Y)	70	50	70	120	70	**120**
	unlock(Y)	70	50	20	120	70	120

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d						
						20	50						
	lock-shared(X)											20	50
	read(X)				**20**		20	50					
	unlock(X)				20		20	50					
	lock-exclusive(Y)							20			20	50	
	read(Y)				20	**50**	20	50					
	Y := Y + X				20	**70**	20	50					
	write(Y)				20	70	20	**70**					
	unlock(Y)				20	70	20	70					
lock-shared(Y)							20	70	20	70			
read(Y)					**70**	20	70	20	70				
unlock(Y)					70	20	70	20	70				
lock-exclusive(X)						70	20	70	20	70			
read(X)				**20**	70	20	70	20	70				
X := X + Y				**90**	70	20	70	20	70				
write(X)				90	70	20	70	**90**	70				
unlock(X)				90	70	20	70	90	70				

Serial Schedules

After T1;T2, we have: X=70, Y=120

After T2;T1, we have: X=90, Y=70

What about a non-serial schedule?

T1	T2	X_{T1}	Y_{T1}	X_{T2}	Y_{T2}	X_d	Y_d
						20	50
lock-shared(Y)						20	50
read(Y)			**50**			20	50
unlock(Y)			50			20	50
	lock-shared(X)		50			20	50
	read(X)		50	**20**		20	50
	unlock(X)		50	20		20	50
	lock-exclusive(Y)		50	20		20	50
	read(Y)		50	20	**50**	20	50
	Y := Y + X		50	20	**70**	20	50
	write(Y)		50	20	70	20	**70**
	unlock(Y)		50	20	70	20	70
lock-exclusive(X)			50	20	70	20	70
read(X)		**20**	50	20	70	20	70
X := X + Y		**70**	50	20	70	20	70
write(X)		70	50	20	70	**70**	70
unlock(X)		70	50	20	70	70	70

Locking Example

After schedule, we have: X=70, Y=70

- The schedule is not serialisable
(not result equivalent to either of the serial schedules)

- Locking, by itself, isn’t enough

Two-Phase Locking (2PL)

Locking and Serialisability

Using locks doesn't guarantee serialisability by itself

Extra rules for handling locks:

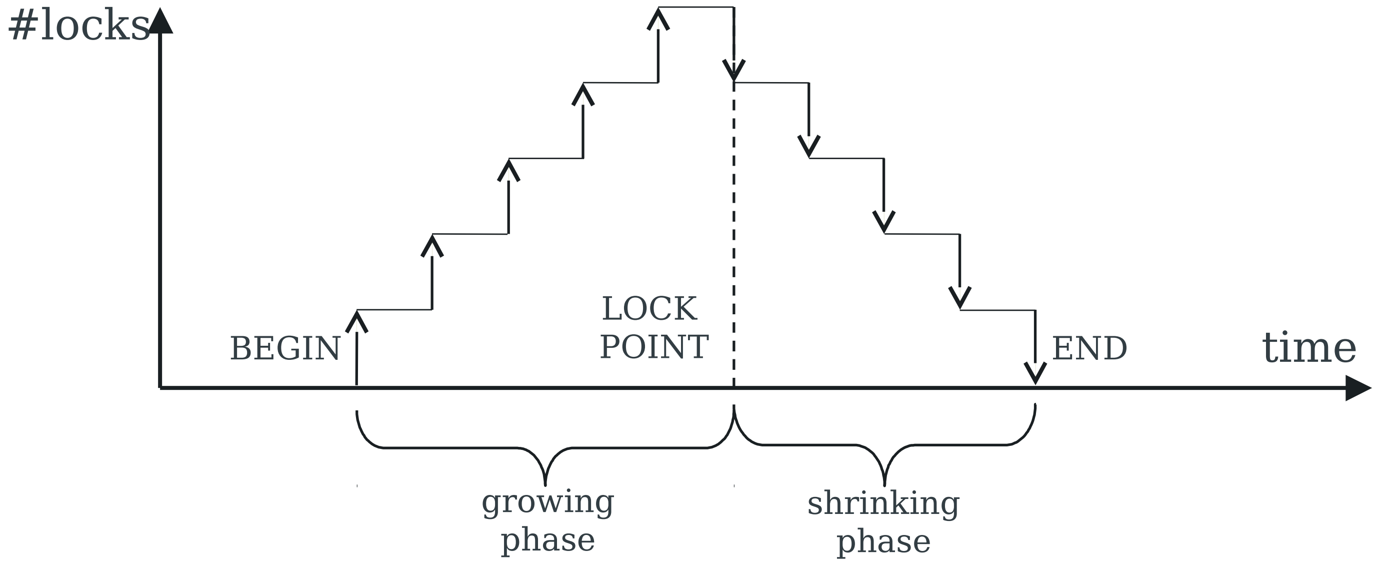
- All locking operations precede the first unlock operation in a transaction
- Locks are only released after a transaction commits or aborts

Two-Phase Locking

Two phases:

- Growing phase: obtain locks, access data items
- Shrinking phase: release locks

Guarantees serialisable transactions



Two-Phase Locking Example

T1:	**T2:**
lock-shared(Y)	lock-shared(X)
read(Y)	read(X)
lock-exclusive(X)	lock-exclusive(Y)
unlock(Y)	unlock(X)
read(X)	read(Y)
X := X + Y	Y := X + Y
write(X)	write(Y)
unlock(X)	unlock(Y)

Deadlock

When 2PL goes wrong

Consider the following schedule of T1 and T2

T1:	**T2:**
lock-shared(Y)	
read(Y)	
	lock-shared(X)
	read(X)
lock-exclusive(X)	
unlock(Y)	
	sive(Y)
…	

T1 can't get an exclusive lock on X; T2 already has a shared lock on X

T2 can't get an exclusive lock on Y; T1 already has a shared lock on Y

Deadlock

Deadlock exists when two or more transactions are waiting for each other to release a lock on an item

Several conditions must be satisfied for deadlock to occur

- Concurrency: two processes claim exclusive control of one resource
- Hold: one process continues to hold exclusively controlled resources until its need is satisfied
- Wait: processes wait in queues for additional resources while holding resource already allocated
- Mutual dependency

Deadlock

- Final condition for deadlock is that some mutual dependency must exist
- Breaking deadlock requires that one transaction is aborted

Processes	Resource List	Wait List
A	1, 10	8
B	3, 4, 15	10
C	2, 0	
D	6, 8	15

Dealing with Deadlock

Deadlock prevention

- Every transaction locks all items it needs in advance; if an item cannot be obtained, no items are locked
- Transactions updating the same resources are not allowed to execute concurrently

Deadlock detection - detect and reverse one transaction

- Wait-for graph
- Timeouts

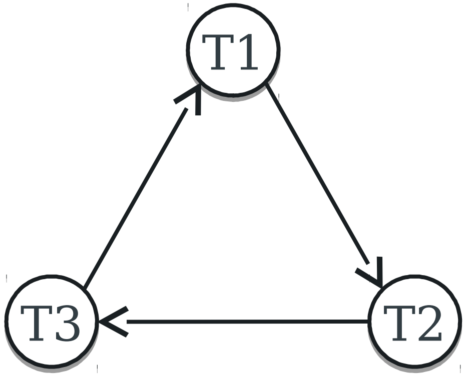
Wait-For Graph

Representation of interactions between transactions

Directed graph containing:

- A vertex for each transaction that is currently executing
- An edge from T1 to T2 if T1 is waiting to lock an item that is currently locked by T2

Deadlock exists iff the WFG contains a cycle



Timeouts

If a transaction waits for a resource for longer than a given period (the timeout), the system assumes that the transaction is deadlocked and aborts it

Timestamps

Timestamps

- An alternative to locks – deadlock cannot occur
- Timestamps are unique identifiers for transactions – the transaction start time: TS(T)
- For each resource X, there is:
  - A read timestamp, read-TS(X)
  - A write timestamp, write-TS(X)
- read-TS(X) and write-TS(X) are set to the timestamp of the most recent corresponding transaction that accessed resource X

Timestamp Ordering

Transactions are ordered based on their timestamps

- Schedule is serialisable
- Equivalent serial schedule has the transactions in order of their timestamps

For each resource accessing by conflicting operations, the order in which the resource is accessed must not violate the serialisability order

Basic Timestamp Ordering

TS(T) is compared with read-TS(X) and write-TS(X)

- Has this item been read or written before transaction T has had an opportunity to read/write?
- Ensure that timestamp ordering is not violated

If timestamp ordering is violated, transaction is aborted and resubmitted with a new timestamp

Basic Timestamp Ordering: write(X)

if read-TS(X) > TS(T) or write-TS(X) > TS(T)

then

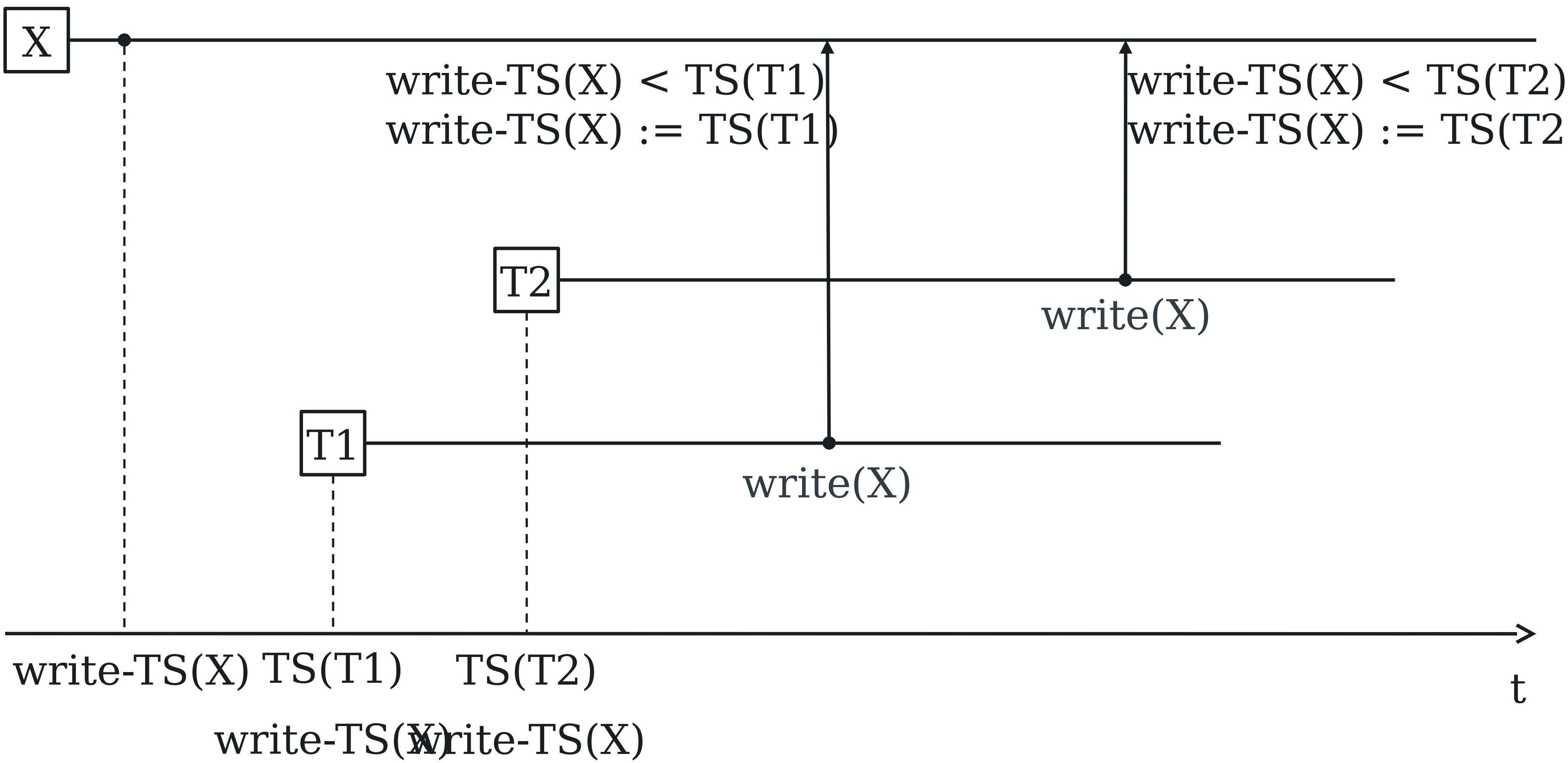
abort and rollback T and reject operation

else

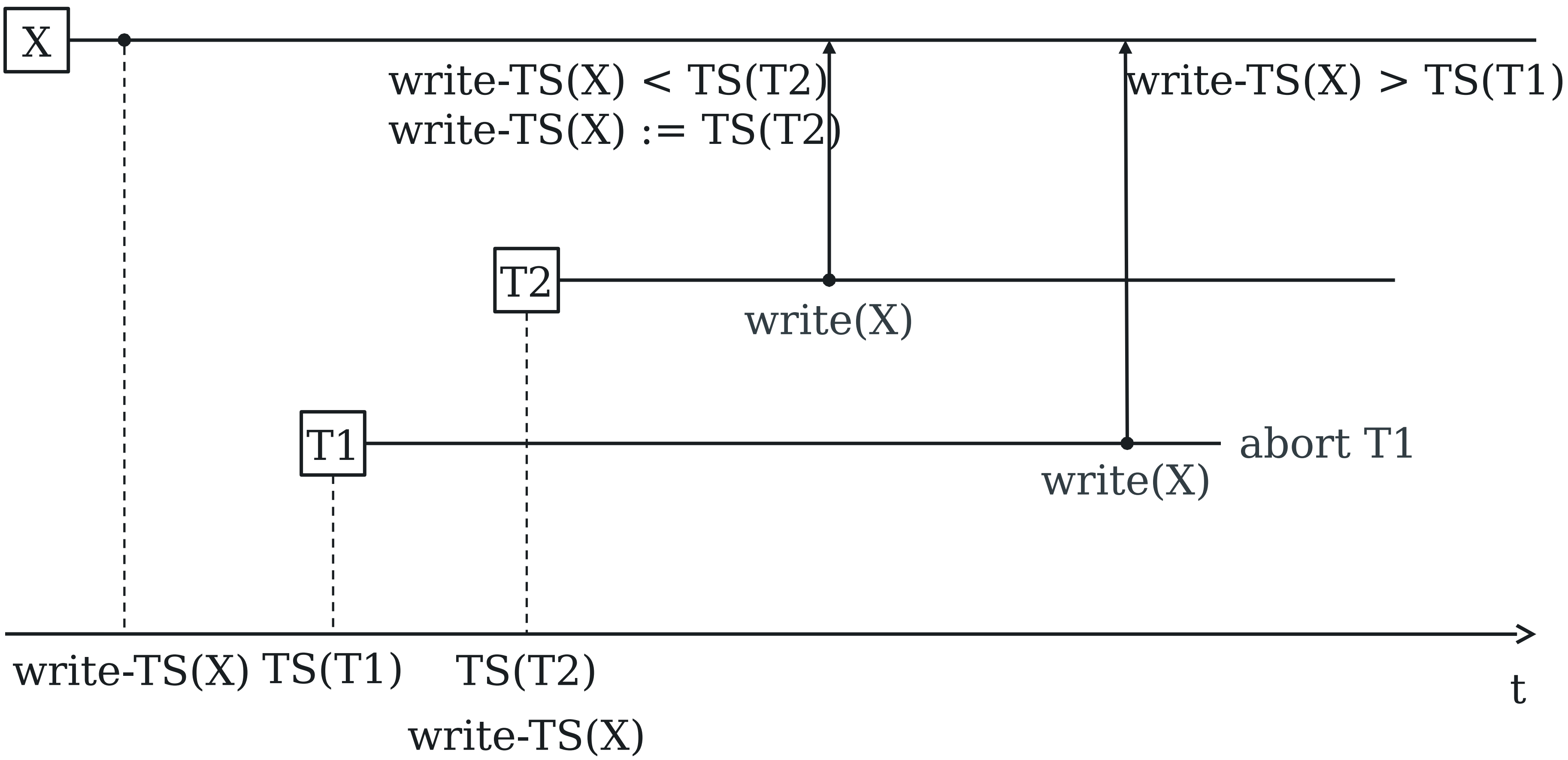
execute write(X)

set write-TS(X) to TS(T)

Basic Timestamp Ordering



Basic Timestamp Ordering



Basic Timestamp Ordering: read(X)

if write-TS(X) > TS(T)

then

abort and rollback T and reject operation

else

execute read(X)

set read-TS(X) to max(TS(T), read-TS(X))

Thomas's Write Rule

- Modification of Basic TO that rejects fewer write operations
- Weakens the checks for write (X) so that obsolete write operations are ignored
- Does not enforce serialisability

Thomas’s Write Rule

if read-TS(X) > TS(T)

then
roll back T and reject operation

if write-TS(X) > TS(T)

then
do not execute write (X)
continue processing

else
execute write(X)
set write-TS(X) to TS(T)

Granularity and Concurrency

Granularity of Data Items

What should be locked?

- Record
- Field value of record
- Disc block
- File
- Database

Coarser granularity gives lower degree of concurrency

Finer granularity gives higher overhead