Concurrency I

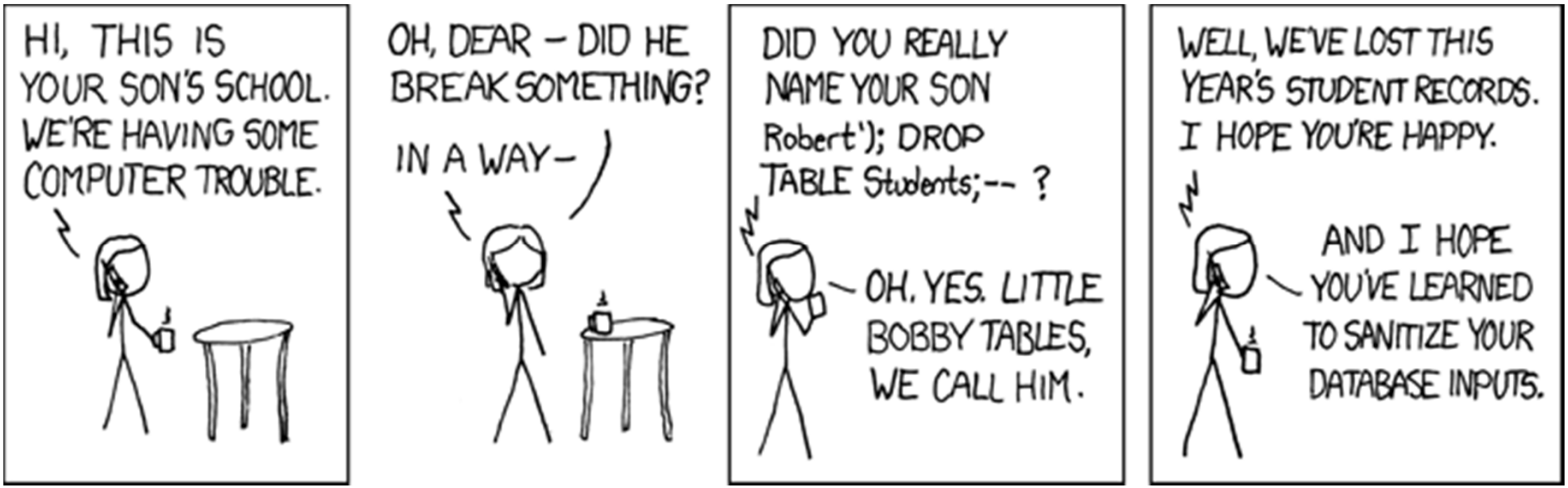
G51DBS Database Systems

Dr Jason Atkin

jaa@cs.nott.ac.uk

Last lecture: Security, including…

SQL injection attacks…



Source: http://xkcd.com/327/

I meant to show you this last week, but forgot, despite a reminder email during the lecture…
… shame I don't read emails during the lecture.

This Lecture

- Concurrency control
- Serialisability
  - Schedules of transactions
  - Serial and serialisable schedules
- Locks
- 2 Phase Locking Protocol
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22

Transactions so Far

- Transactions are the ‘logical unit of work’ in a database
  - ACID properties
  - Also the unit of recovery
- Transactions will involve some read and/or writes on a database
- COMMIT
  - Signals the successful end of a transaction
  - Changes are made permanent and visible to other transactions
- ROLLBACK
  - Signals the unsuccessful end of a transaction
  - Changes are undone

Transactions so Far

- Atomicity
  - Transactions conceptually have no component parts
  - They run completely, or not at all
- Consistency
  - Transactions take the database from one consistent state to another
- Isolation
  - Incomplete transactions are invisible to others until they have committed
- Durability
  - Committed transactions must be made permanent

Transaction Log

- Recover database after a system failure
- Check points: flush memory to disk. Record what is still running
- Transaction log:
  - Stores all operations
  - Including start and commit
- MUST write to log BEFORE committing the transaction

- Build undo and redo list
- Undo list has all transactions running at last check point
- Work forwards:
  - Add to undo list when transaction starts
  - Move to redo list when transaction commits
- Work backwards – undo all ‘undo’ operations
- Work forwards – redo all ‘redo’ operations

Concurrency

- Large databases are used by many people
  - Many transactions are to be run on the database
  - It is helpful to run these simultaneously
  - Still need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Easy to preserve atomicity and isolation
  - Long transactions (e.g. backups) will delay others

Concurrency Problems

- In order to run two or more concurrent transactions, their operations must be interleaved
- Each transaction gets a share of the computing time
- This can lead to several problems, e.g.:
  - Lost updates
  - Uncommitted updates
  - Inconsistent Analysis
- All arise when isolation is broken

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

- T1 and T2 both read X, both modify it, then both write it out
  - The net effect of both transactions should be no change to X
  - Only T2's change is seen however

Uncommitted Update

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
ROLLBACK	
	X = X + 5
	Write(X)
	COMMIT

- T2 sees the change to X made by T1, but T1 is then rolled back
  - The change made by T1 is rolled back
  - It should be as if that change never happened

Inconsistent Analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X + Y
Read(Y)	
Y = Y + 5	
Write(Y)	

- T1 doesn't change the sum of X and Y, but T2 records a change
  - T1 consists of two parts - take 5 from X then add 5 to Y
  - T2 sees the effect of the first change, but not the second

Concurrency Control

- Concurrency control is the process of managing simultaneous operations on the database without having them interfere with each other
  - Possibly reading and writing the same data
  - Long transactions must not hold up others
  - ACID properties must be maintained

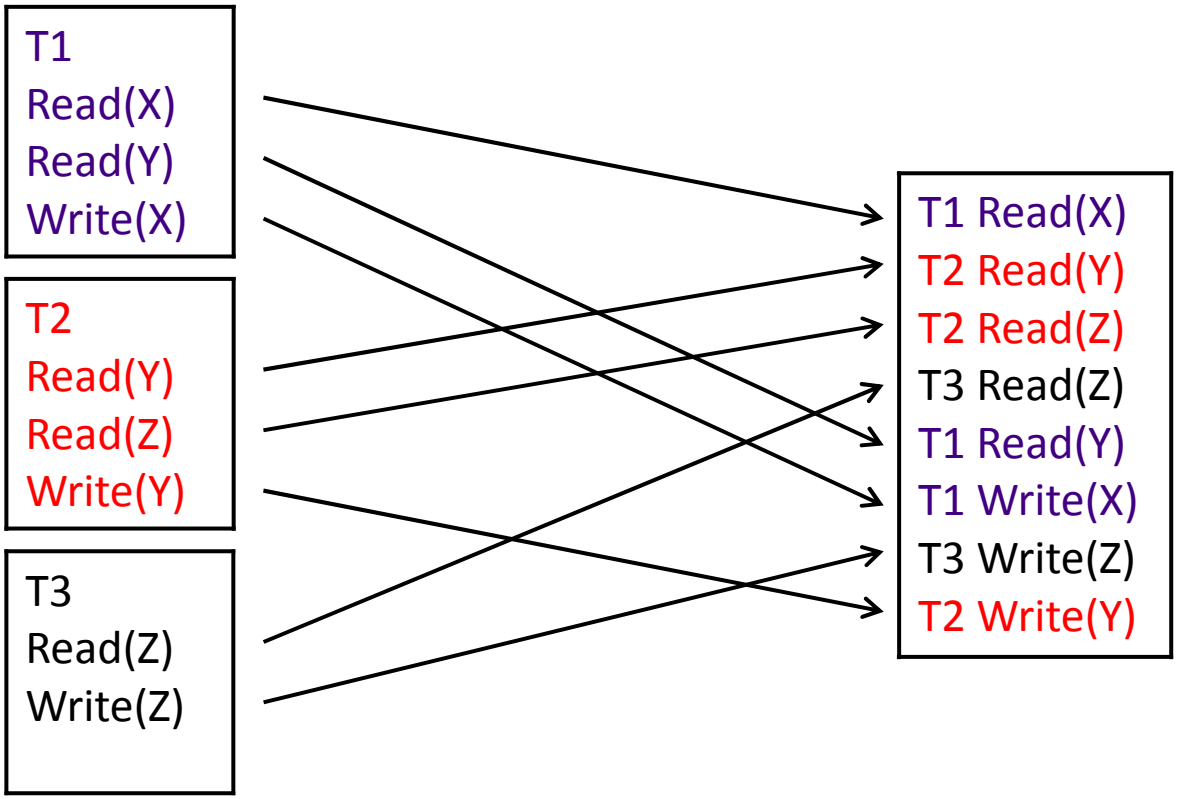
Schedules

- A *schedule* is a sequence of the operations in a set of concurrent transactions **that preserves the order of operations in each of the individual transactions**

- A ***serial*** schedule is a schedule where the operations of each transaction are executed consecutively **without any interleaved operations from other transactions** (each must commit before the next can begin)

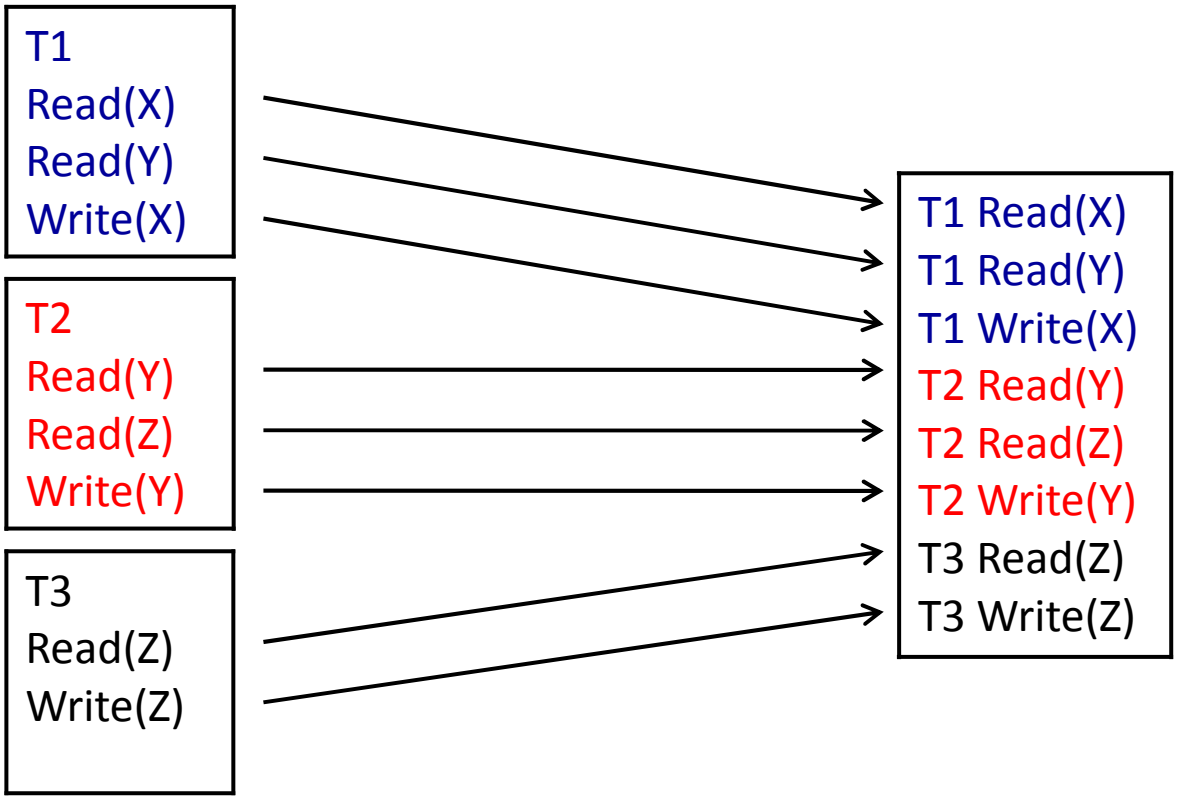
Example Schedule

- Three transactions:
- Example schedule



Example Schedule

- Three transactions:



- Example serial schedule

T1 Read(X)
T1 Read(Y)
T1 Write(X)
T2 Read(Y)
T2 Read(Z)
T2 Write(Y)
T3 Read(Z)
T3 Write(Z)

Serial Schedules

- A serial schedule is guaranteed to avoid interference between transactions, and preserve database consistency

- However, this approach does not allow for concurrent access, i.e. Interleaving operations from multiple transactions

Serialisability

- Two schedules are equivalent if they always have the same effect
- A schedule is *serialisable* if it is equivalent to some serial schedule
- For example:
  - If two transactions only read from some data items, the order in which they do this is not important
  - If T1 reads and then updates X, and T2 reads then updates Y, then again this can occur in any order

Serial and Serialisable

- Interleaved (nonserial) Schedule

T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)

- Serial Schedule

T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)

This schedule is serialisable – has the same effect as a serial schedule

Conflict Serialisability

- Two transactions have a confict:
  - NO If they refer to different resources
  - NO If they only read
  - YES **If they use the *same* resource and at least one of them *writes* to it**
- A schedule is conflict serialisable if the transactions in the schedule have a conflict, but the schedule is still serialisable

Conflict Serialisable Schedule

- Interleaved Schedule

T1 Read(X)
T1 Write(X)
T2 Read(X)
T2 Write(X)
T1 Read(Y)
T1 Write(Y)
T2 Read(Y)
T2 Write(Y)

- Serial Schedule

T1 Read(X)
T1 Write(X)
T1 Read(Y)
T1 Write(Y)

T2 Read(X)
T2 Write(X)
T2 Read(Y)
T2 Write(Y)

This schedule is serialisable, even though T1 and T2 read and write the same resources X and Y: They have a conflict

Conflict Serialisability

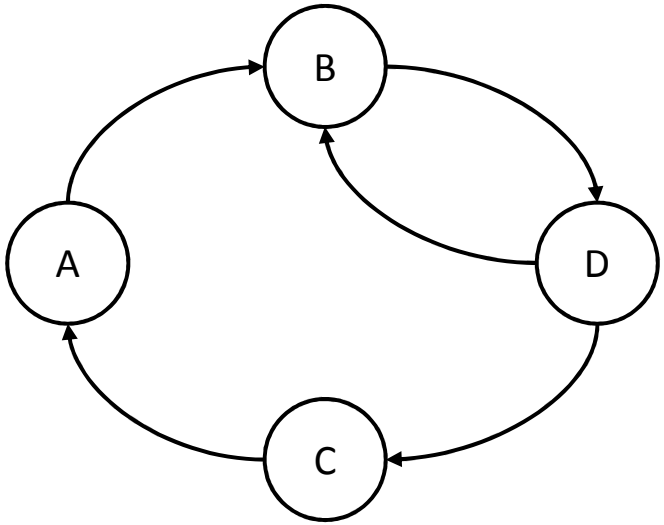
- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are **guaranteed** to behave as a serial schedule

Important questions:

- How do we determine whether or not a schedule is conflict serialisable?
- How do we construct conflict serialisable schedules

Graphs

- In mathematics, a graph is a structure (V,E) of Vertices and Edges. In the case of a directed graph, these edges include directions
- For example:



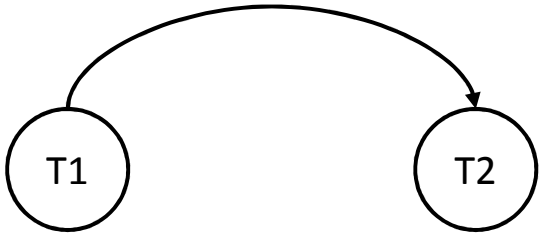
Precedence Graphs

- To determine if a schedule is conflict serialisable we use a precedence graph
  - Transactions are vertices of the graph
  - There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serialisable schedule
- Look at each resource in turn and identify whether anything writes to it
- Add an edge T1 → T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R)
    - T1 has to read old value
  - T1 Write(R) followed by T2 Read(R)
    - T2 has to read new value
  - T1 Write(R) followed by T2 Write(R)
  - All of these mean T1 has to come before T2 in an equivalent serial schedule
- **The schedule is serialisable if there are no cycles**

Precedence Graph Example

- No cycles: conflict serialisable schedule

- T1 reads X before T2 writes X
- T1 writes X before T2 reads X
- T1 writes X before T2 writes X

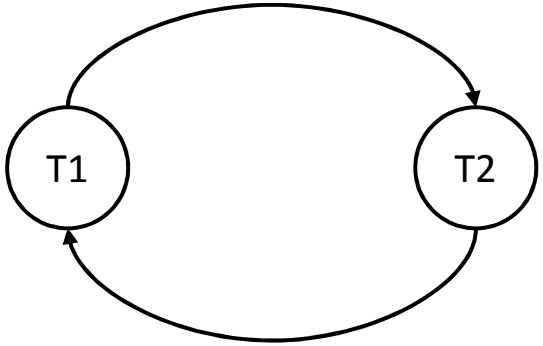


T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

Precedence Graph Example

- The lost update problem has this precedence graph:

- **T1 reads X before T2 writes X**
- **T1 writes X before T2 writes X**



- **T2 reads X before T1 writes X**

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Ensuring serialisability : Overview

- We want to be able to ensure that schedules are serialisable
- We will see two ways to do this:

1. Based on locking: if a transaction uses a resource before another transaction, then ensure that the second one cannot use it at all until the first finishes with it
2. Timestamping: if we spot a problem with serialisability then restart one transaction

Locking Overview

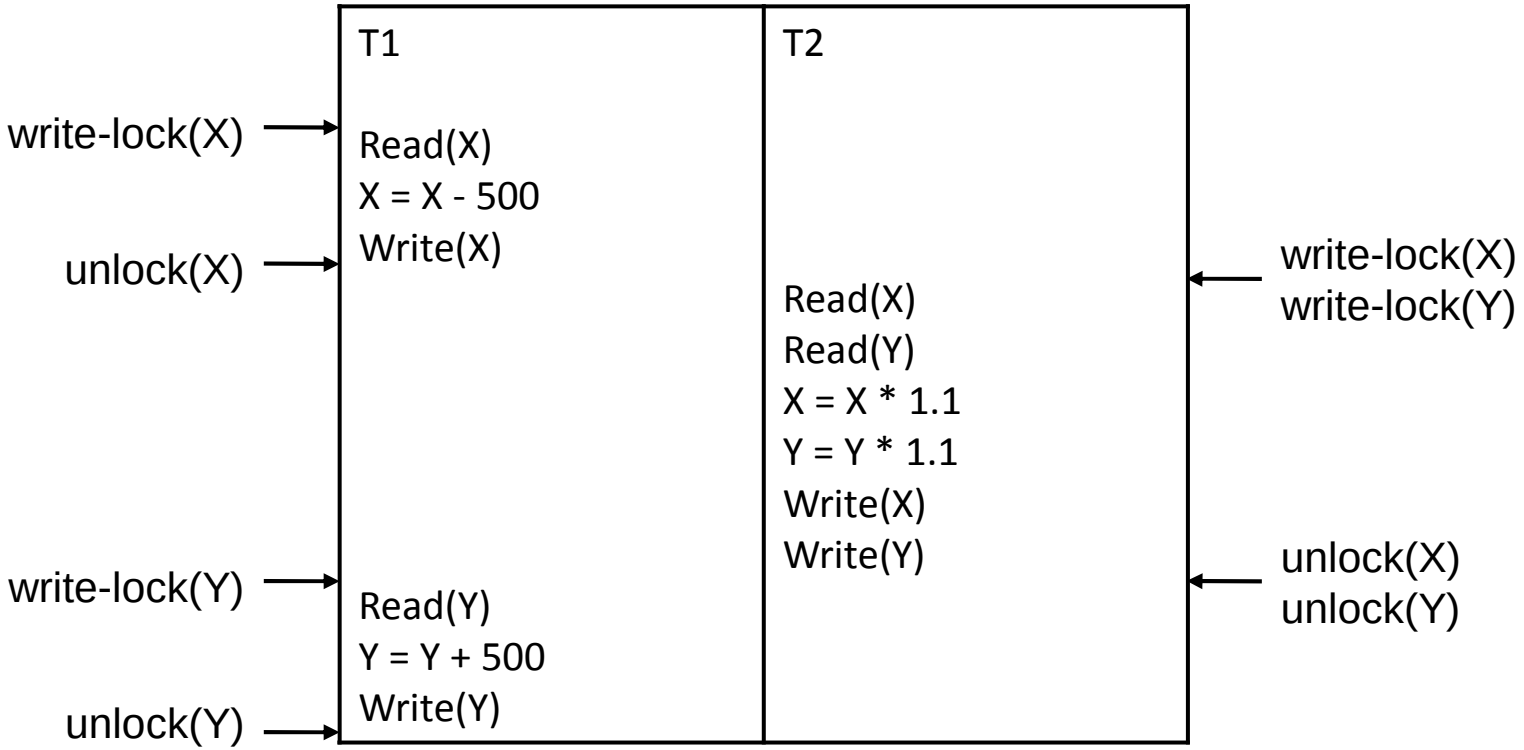
- Locking is a procedure used to control concurrent access to data (resources)
  - To ensure serialisability of concurrent transactions
- A resource could be a single item of data, some or all of table, or even a whole database
- In order to use a 'resource' a transaction must first acquire a lock on that resource
- This may deny access to other transactions to prevent incorrect results
- **There are two types of lock**
  - Shared lock (often called a read lock)
  - Exclusive lock (often called a write lock)
- Read locks allow several transactions to read data simultaneously, but none can write to that data
- Write locks allow a single transaction exclusive access to read and write a resource

Locking Process

- The DBMS is responsible for issuing locks (not the user)
- Before reading from a resource a transaction must acquire a read-lock
- Before writing to a resource a transaction must acquire a write-lock
- A lock might be released during execution when no longer needed, or upon COMMIT or ROLLBACK
- A transaction may not acquire a **lock** (read or write) on any resource that is currently **write-locked** by another transaction
- A transaction may not acquire a **write lock** on any resource that is currently **locked** (read or write) by another transaction
- **If the requested lock is not available, the transaction waits**

Locking Example

- Locking on its own **won't** successfully allow us to serialise all schedules. For example:



Two-Phase Locking

- A transaction follows the two-phase locking protocol (2PL) if **all** locking operations precede **all** unlocking operations
- Other operations can happen at any time throughout the transaction

- Two phases:
  - **Growing** phase where locks are acquired
  - **Shrinking** phase where locks are released

Two-Phase Locking Example

- T1 follows 2PL protocol
  - All locks in T1 are acquired before any are released
  - This happens even if the resource is no longer used
- T2 does not
  - Releases a lock on X, which is no longer needed, before acquiring on Y

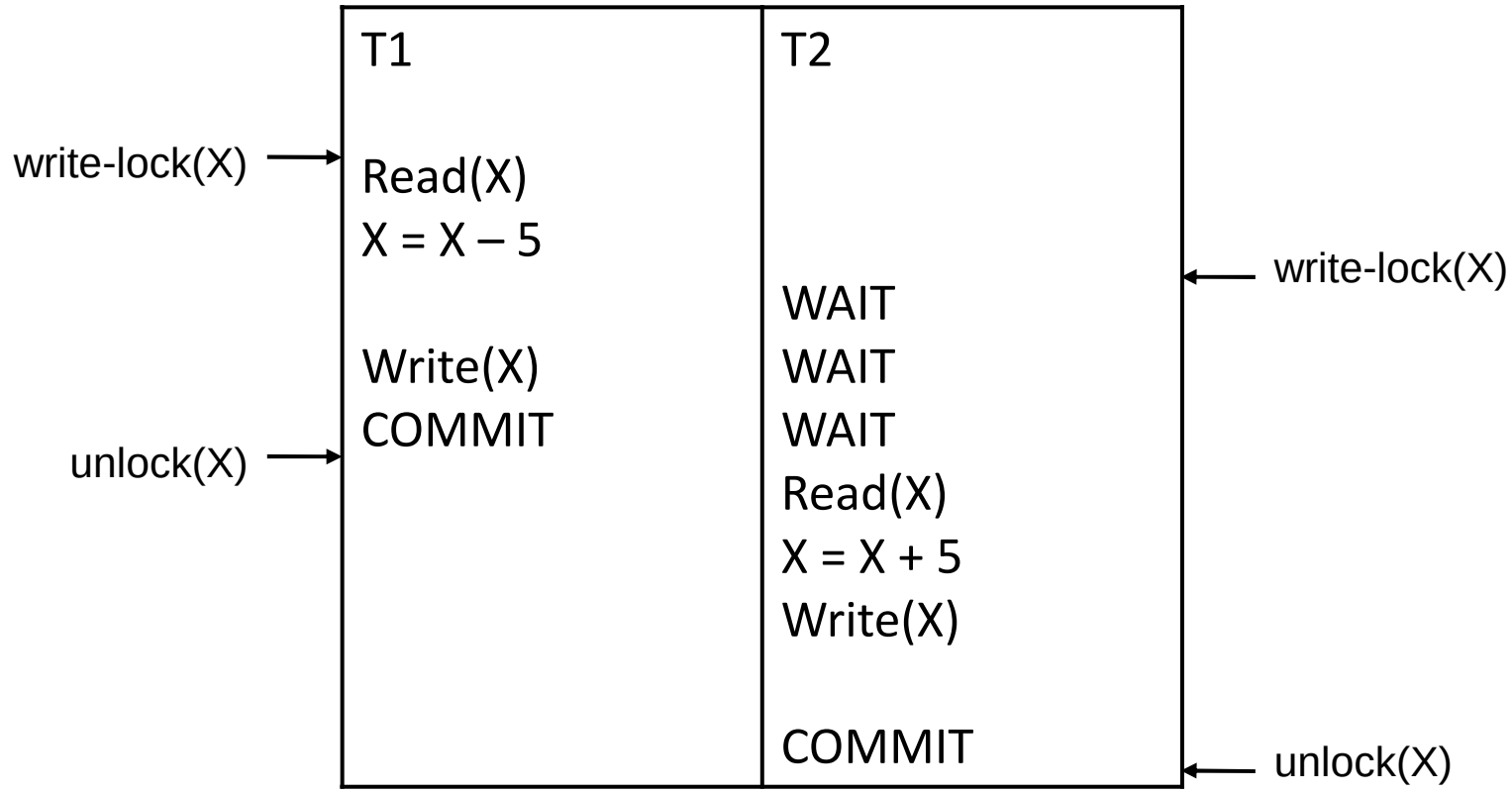
T1
write-lock(X)
Read(X)
X = X + 100
Write(X)
write-lock(Y)
unlock(X)
Read(Y)
Y = Y – 100
Write(Y)
unlock(Y)
COMMIT

T2
write-lock(X)
Read(X)
X = X + 100
Write(X)
unlock(X)
write-lock(Y)
Read(Y)
Y = Y – 100
Write(Y)
unlock(Y)
COMMIT

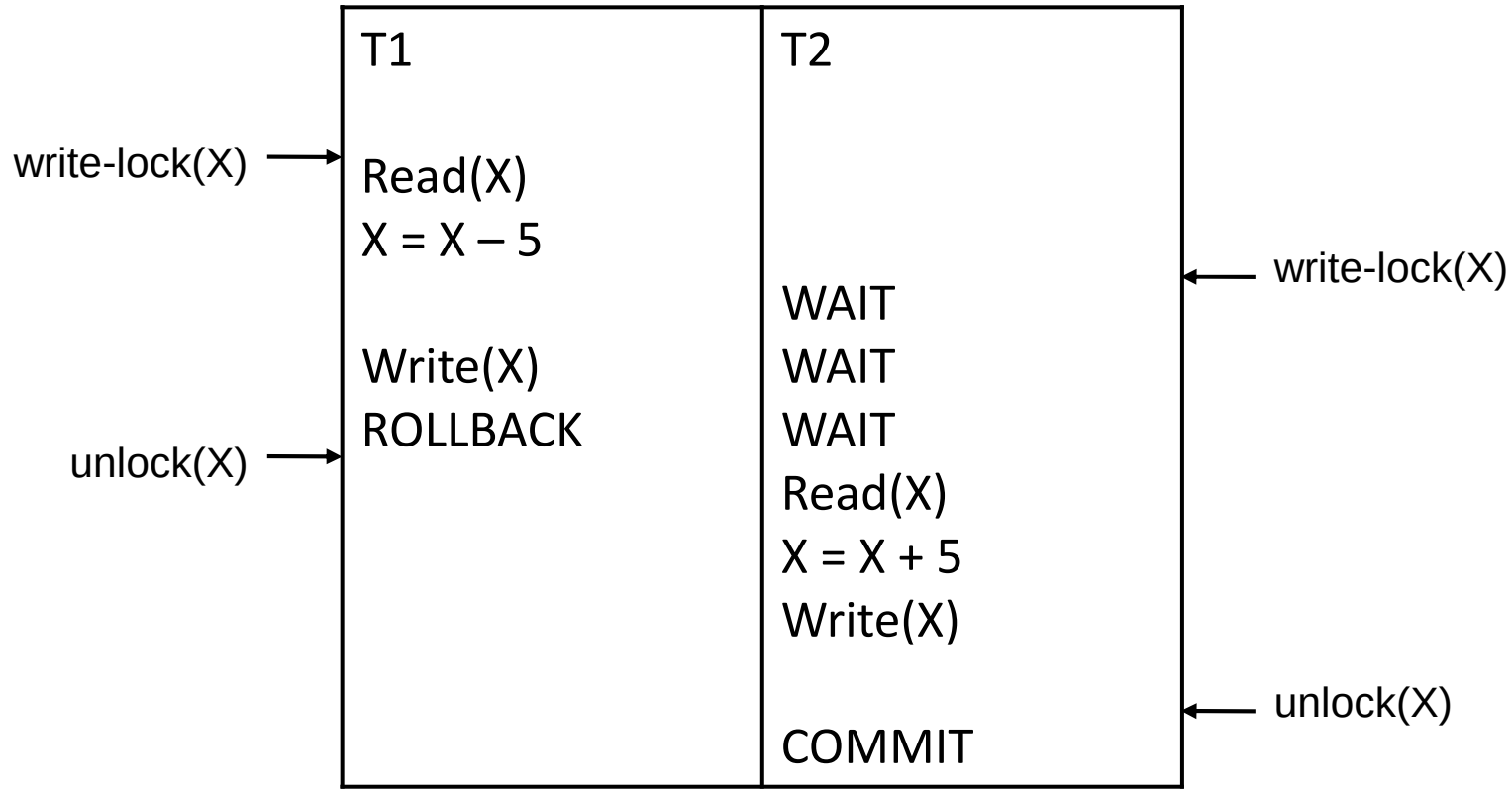
Serialisability Theorem

Any schedule of two-phase locking transactions is conflict serialisable

2PL Prevents Lost Update

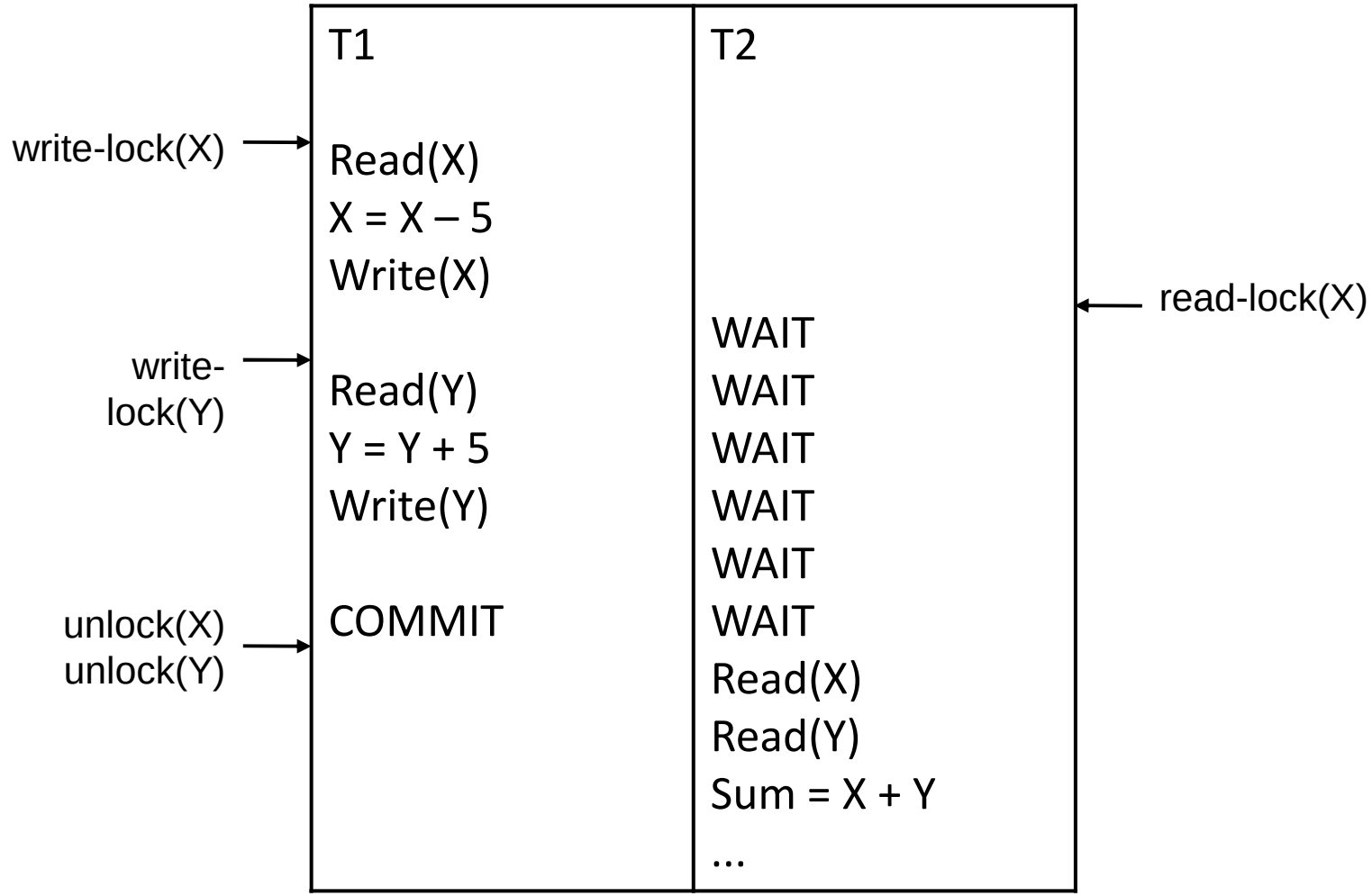


2PL Prevents Uncommitted Update



The value of X is restored during rollback, before the write-lock is released

2PL Prevents Inconsistent Analysis



Concurrency in SQL

- Concurrency in MySQL (and most other DBMSs) is handled automatically
- **UPDATE, INSERT, DELETE etc will obtain write locks**
- **SELECT will obtain a read lock – or may read an old cached value**

- In MySQL, the locking protocol depends on the engine
  - MyISAM: Table Level Locking
  - Memory: Table Level Locking
  - InnoDB: Row Level Locking

Concurrency in MySQL

- Sometimes you might want to lock a resource specifically for updating:

```
SELECT ID FROM Artist WHERE Name = 'Muse';

... Some processing

INSERT INTO CD VALUES (NULL, 2, 'The
  Resistance', 9.99, 'Rock');
```

- In the short time between these queries, the ID for muse may have been written to
- The SELECT needs to lock the resource

Locking in a SELECT

- For times when a Subquery isn't appropriate:

```
SELECT *
 FROM table
 WHERE ...
 FOR UPDATE;
```

- **FOR UPDATE** write-locks all rows that we read **until the end of the transaction.**
- It has the added benefit of reading the very latest values of these rows (not using cached values)
- You can use **LOCK IN SHARE MODE** to obtain a read lock instead

This Lecture in Exams

Define the term *Schedule*, *Serial Schedule* and *Serialisable* in the context of database concurrency

Explain the Lost Update problem, and provide an example schedule demonstrating this

Explain how the two-phase locking protocol can avoid the lost update problem

Next Lecture

- Deadlocks
  - Deadlock detection
  - Deadlock prevention
- Timestamping
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22

Concurrency II

G51DBS Database Systems

Dr Jason Atkin

jaa@cs.nott.ac.uk

Last Lecture

- Serialisability
  - Schedules of transactions
  - Serial and serialisable schedules
  - Conflict serialisable schedules
- Locks
  - Shared (Read)
  - Exclusive (Write)
- Two-phase locking
  - Growing Phase
    - Transactions obtain locks
  - Shrinking Phase
    - Transactions release locks
- Two-phase locking guarantees conflict serialisability
  - Allows Concurrency
  - Avoids loss of Isolation

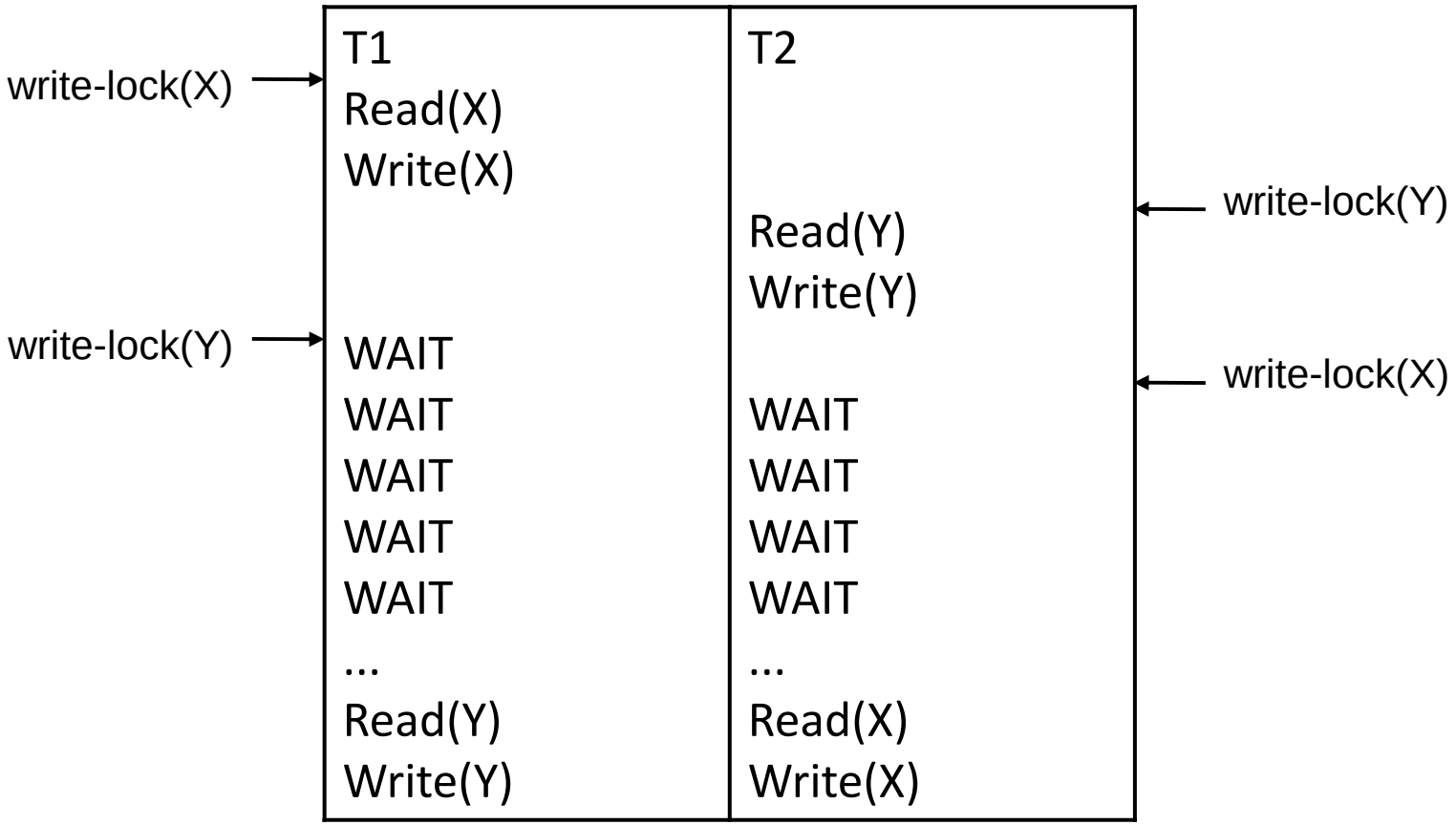
This Lecture

- Deadlocks
  - Deadlock detection
  - Deadlock recovery
  - Deadlock prevention
- Timestamping
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapter 22

Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example:
    T1 has a lock on X and is waiting for a lock on Y
    T2 has a lock on Y and is waiting for a lock on X
    Both wait for each other
- We can detect deadlocks that will happen in a schedule using a *wait-for graph* (WFG)
- We are tracking what is 'waiting for' something else
  - If there is any **cycle** of things waiting for other things, then they will **deadlock**

2PL Deadlock Example



Precedence vs Wait-For Graphs

Precedence graph

- Aim to find out what order transactions would have to take place in equivalent serial schedule
- Each transaction is a vertex
- Looks at read-writes
- Edge from T1 to T2 if
  - T1 **reads** X then T2 **writes** X
  - T1 **writes** X then T2 **reads** X
  - T1 **writes** X then T2 **writes** X

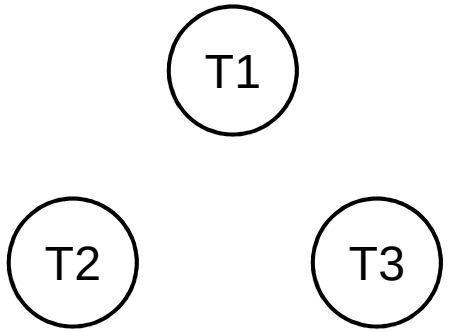
Wait-for Graph

- Aim to find out if we (will) have a deadlock
- Each transaction is a vertex
- Looks at locks
- Edge from T2 to T1 if
  - T1 **read-locks** X then T2 tries to **write-lock** it
  - T1 **write-locks** X then T2 tries to **read-lock** it
  - T1 **write-locks** X then T2 tries to **write-lock** it

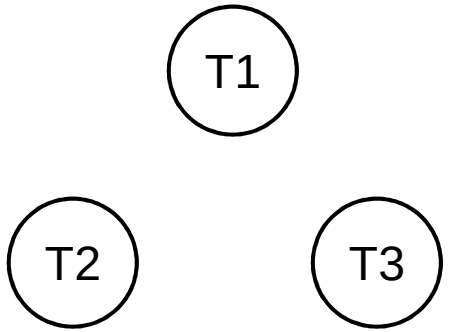
Example

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



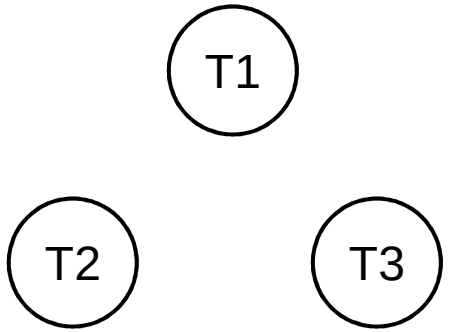
Wait for graph



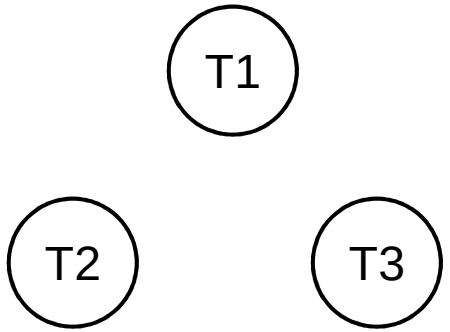
Precedence graph

Example

Schedule	Locks:
T1 Read(X)	T1 RL(X)
T2 Read(Y)	T2 RL(Y)
T1 Write(X)	T1 WL(X)
T2 Read(X)	T2 RL(X)
T3 Read(Z)	T3 RL(Z)
T3 Write(Z)	T3 WL(Z)
T1 Read(Y)	T1 RL(Y)
T3 Read(X)	T3 RL(X)
T1 Write(Y)	T1 WL(Y)



Wait for graph

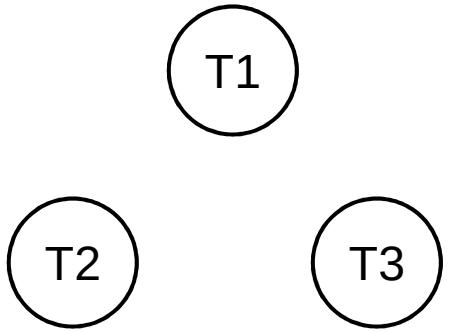


Precedence graph

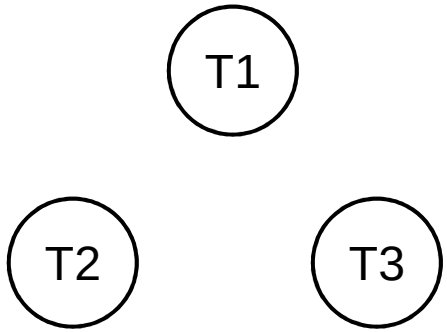
We could assume that a transaction asks for a read or write lock when needed then keeps them all until commit – i.e. definitely two-phase locking

The writes/ write locks matter:

Schedule	Locks:
T1 Read(X)	T1 RL(X)
T2 Read(Y)	T2 RL(Y)
T1 Write(X)	**T1 WL(X)**
T2 Read(X)	T2 RL(X)
T3 Read(Z)	T3 RL(Z)
T3 Write(Z)	**T3 WL(Z)**
T1 Read(Y)	T1 RL(Y)
T3 Read(X)	T3 RL(X)
T1 Write(Y)	**T1 WL(Y)**



Wait for graph

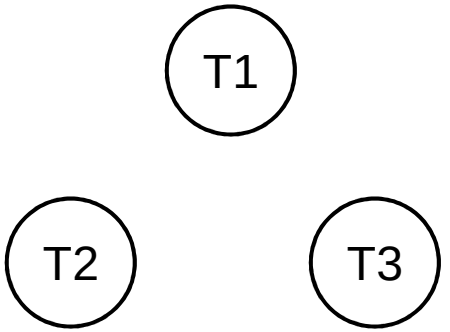


Precedence graph

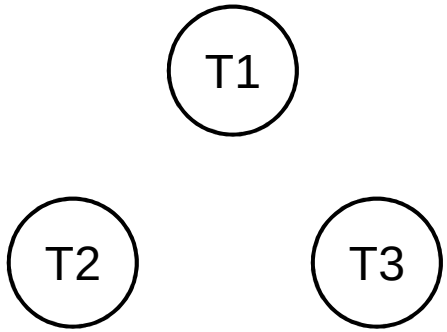
Example : Precedence graph first

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph



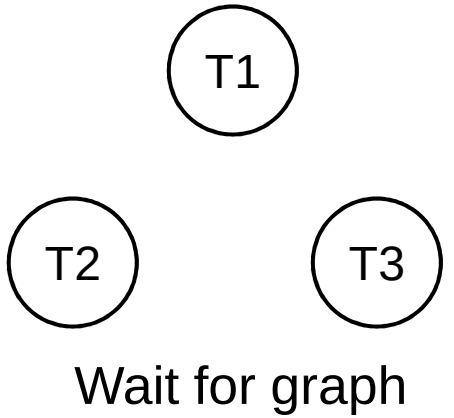
Precedence graph

There are three WRITEs that we need to consider, and any related READs

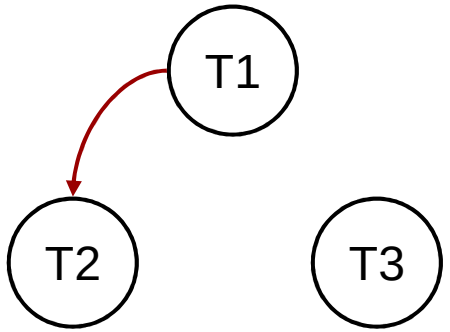
Example

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph

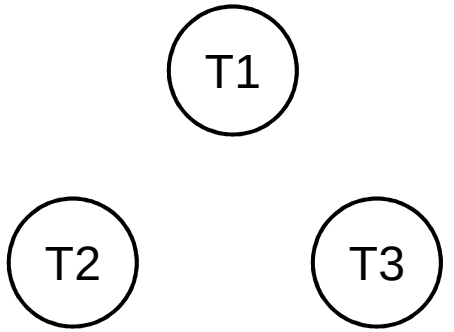


Precedence graph

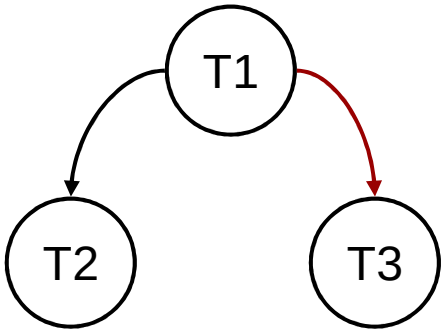
Example

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph

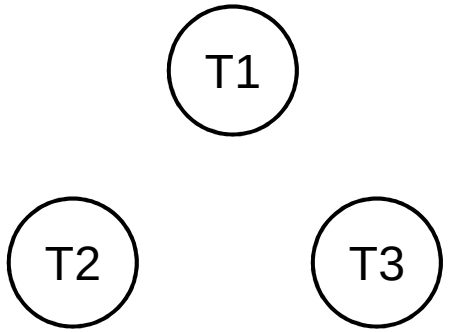


Precedence graph

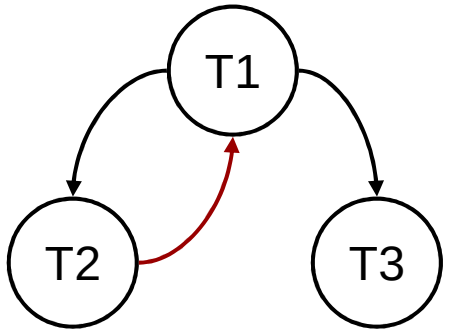
Example

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph

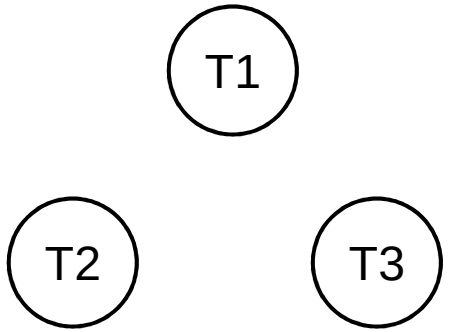


Precedence graph

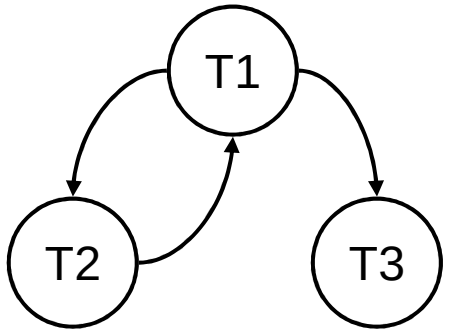
Finished precedence graph

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph



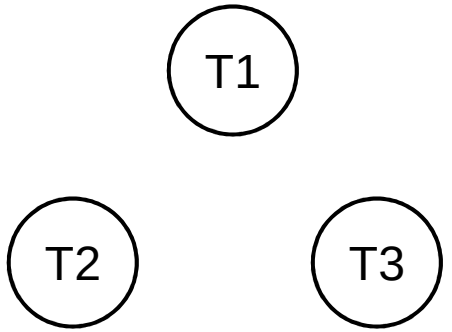
Precedence graph

Only T3 uses Z so there are no problems with writing Z

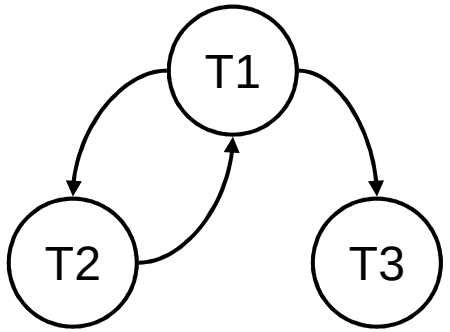
Finished precedence graph

Schedule

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph

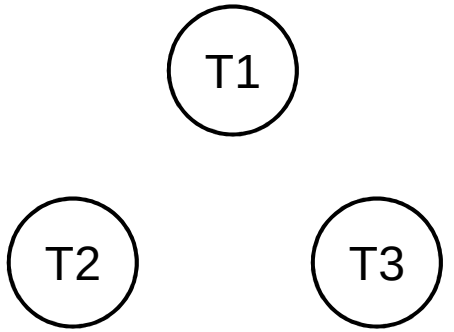


Precedence graph

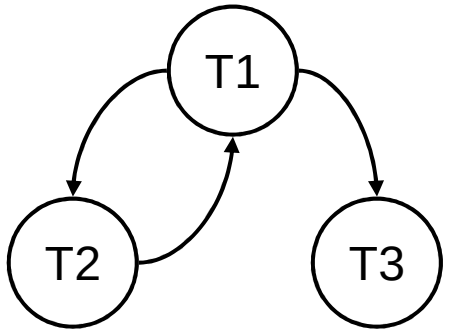
Cycle means that the schedule is not serialisable
(there is no equivalent serial schedule)

Example : The Wait-For Graph

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	write-lock(X)
T2 Read(X)	
T3 Read(Z)	
T3 Write(Z)	
T1 Read(Y)	
T3 Read(X)	
T1 Write(Y)	



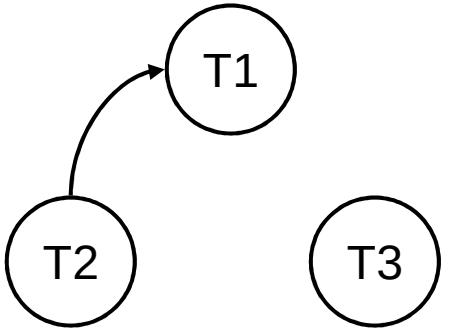
Wait for graph



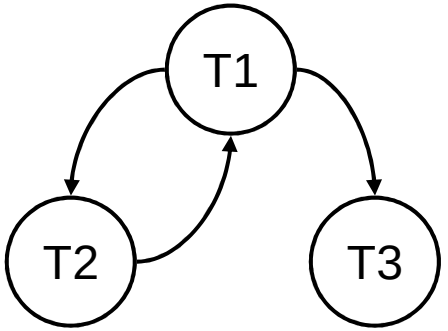
Precedence graph

Example

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	**write-lock(X)**
T2 Read(X)	**tries read-lock(X)**
T3 Read(Z)	
T3 Write(Z)	
T1 Read(Y)	
T3 Read(X)	
T1 Write(Y)	



Wait for graph

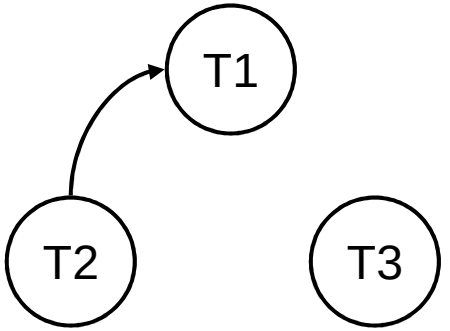


Precedence graph

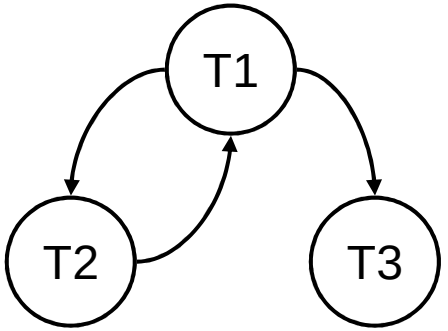
T2 has to wait for T1 to finish

Example

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	write-lock(X)
T2 Read(X)	tries read-lock(X)
T3 Read(Z)	read-lock(Z)
T3 Write(Z)	write-lock(Z)
T1 Read(Y)	read-lock(Y)
T3 Read(X)	
T1 Write(Y)	



Wait for graph

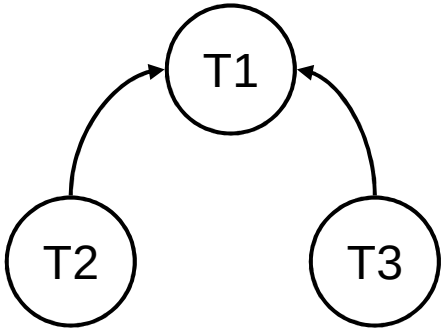


Precedence graph

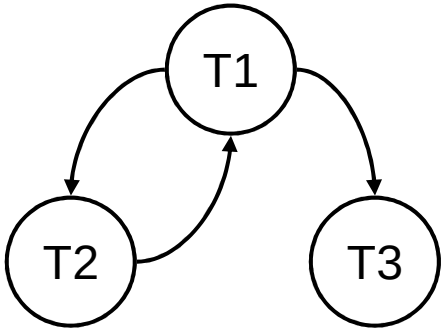
No problem with locking different data
Or with multiple read locks

Example

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	**write-lock(X)**
T2 Read(X)	tries read-lock(X)
T3 Read(Z)	read-lock(Z)
T3 Write(Z)	write-lock(Z)
T1 Read(Y)	read-lock(Y)
T3 Read(X)	**tries read-lock(X)**
T1 Write(Y)	



Wait for graph

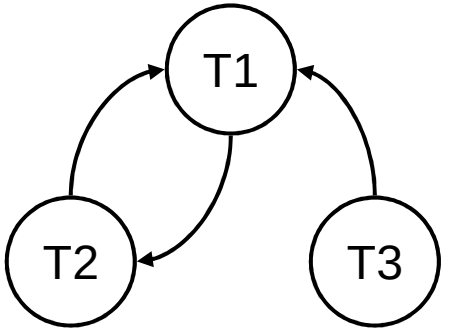


Precedence graph

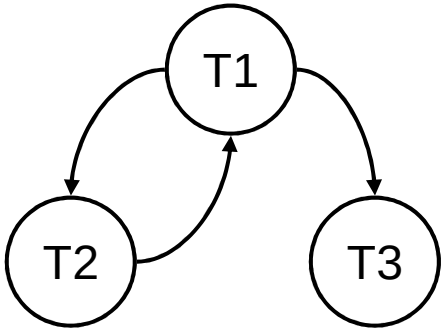
T3 needs to wait for T1 to finish

Example

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	**read-lock(Y)**
T1 Write(X)	write-lock(X)
T2 Read(X)	tries read-lock(X)
T3 Read(Z)	read-lock(Z)
T3 Write(Z)	write-lock(Z)
T1 Read(Y)	read-lock(Y)
T3 Read(X)	tries read-lock(X)
T1 Write(Y)	**tries write-lock(Y)**



Wait for graph

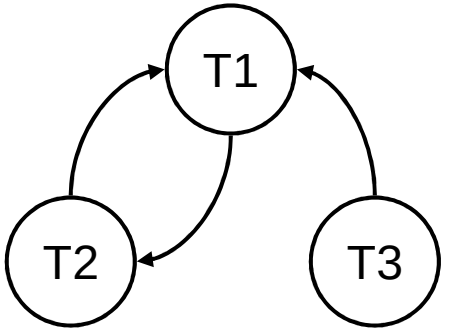


Precedence graph

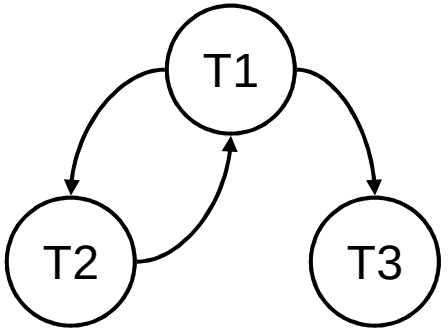
T1 needs to wait for T2 to finish

Example

Schedule	Locks
T1 Read(X)	read-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	write-lock(X)
T2 Read(X)	tries read-lock(X)
T3 Read(Z)	read-lock(Z)
T3 Write(Z)	write-lock(Z)
T1 Read(Y)	read-lock(Y)
T3 Read(X)	tries read-lock(X)
T1 Write(Y)	tries write-lock(Y)



Wait for graph

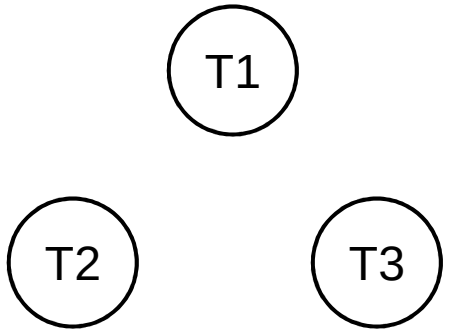


Precedence graph

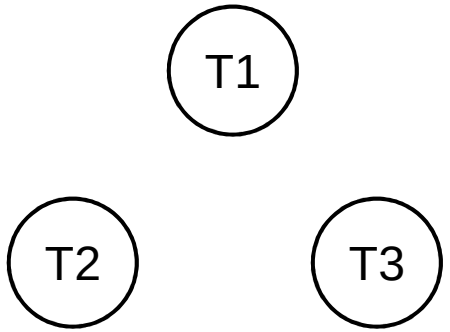
A cycle in the wait-for graph means we will encounter deadlock

Alternative locking approach…

Schedule	Locks:
T1 Read(X)	**T1 WL(X)**
T2 Read(Y)	T2 RL(Y)
T1 Write(X)	
T2 Read(X)	T2 RL(X)
T3 Read(Z)	**T3 WL(Z)**
T3 Write(Z)	
T1 Read(Y)	**T1 WL(Y)**
T3 Read(X)	T3 RL(X)
T1 Write(Y)	



Wait for graph

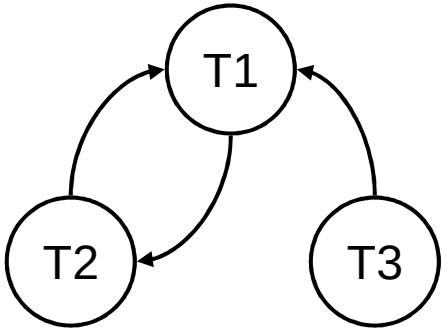


Precedence graph

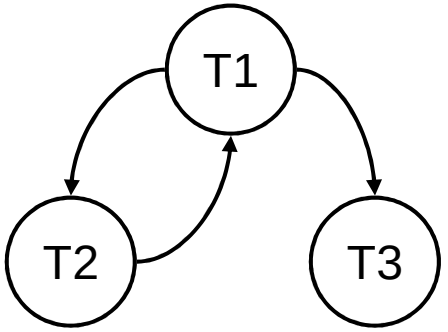
We could assume that any which will need a write lock ask for it at the read, rather than getting a read lock first (i.e. T1 write-locks Y when reading it)

Example

Schedule	Locks
T1 Read(X)	write-lock(X)
T2 Read(Y)	read-lock(Y)
T1 Write(X)	
T2 Read(X)	tries read-lock(X)
T3 Read(Z)	write-lock(Z)
T3 Write(Z)	
T1 Read(Y)	tries write-lock(Y)
T3 Read(X)	tries read-lock(X)
T1 Write(Y)	



Wait for graph



Precedence graph

Note: Cycle even if T1 got the write lock as soon as it read Y
Even though we implemented 2-phase locking

Deadlock Recovery

- Deadlocks **can** arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
- Most DBMSs will detect deadlocks with a wait-for graph
  - Chose a single transaction as a 'victim' to rollback
    - Which transaction has been running the longest?
    - Which transactions have made the most updates?
    - Which transactions have the most updates still to make?
  - Rollback may restart or may be able to undo part of the transaction to remove the deadlock

Extension of “Serialisability Theorem”

Any schedule of two-phase locking transactions is conflict serialisable

But it may still suffer from the problem of deadlock

Deadlock Prevention

- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - For high lock contention this works well
    - Transactions are never blocked once they start
  - For low lock contention this is not as effective
    - Locks are held longer than necessary
    - Transactions might hold on to locks for a long time, but not use them much
- A general resolution is to detect that deadlock will occur and roll back some transactions

Timestamping

- Transactions can be run concurrently using a variety of techniques
- We looked at using locks to preserve isolation

- An alternative is timestamping
  - Requires less overhead in terms of tracking locks or detecting deadlock
  - Determines the order of transactions before they are executed
  - Most useful for a small number of transactions

Timestamping

- Each **transaction** has a timestamp, TS, and if T1 starts before T2 then TS(T1) < TS(T2)
- Can use the system clock or an incrementing counter to generate timestamps
- Newer transactions get higher timestamps
- Each **resource** has two timestamps
- R(X), the largest timestamp of any transaction that has read X
  - i.e. newest transaction to read it
- W(X), the largest timestamp of any transaction that has written X
  - i.e. newest transaction to write it

Timestamp Protocol

- If T tries to read X
  - **If TS(T) < W(X)**
    T is rolled back and restarted with a later timestamp
    (a transaction with a larger timestamp has altered X)
  - If TS(T) ≥ W(X) then the read succeeds and we set **R(X) to be max(R(X), TS(T))**
    (record that we read it)
- T tries to write X
  - **If TS(T) < W(X)**
    or TS(T) < R(X)
    then T is rolled back and restarted with a later timestamp
    (a transaction with a larger timestamp has read or written this)
  - Otherwise the write succeeds and we set **W(X) to TS(T)**
    (record that we read it)

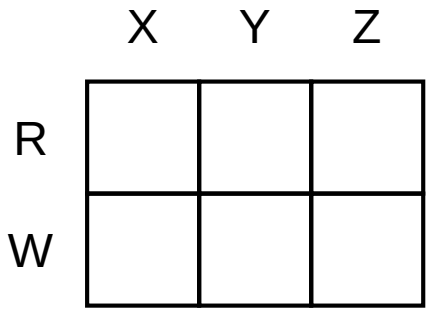
Timestamping Example

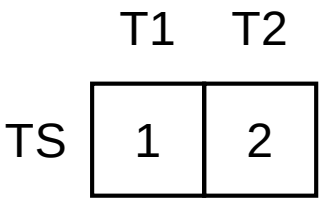
- Given T1 and T2 we will assume
  - The transactions make alternate operations
  - Timestamps are allocated from a counter starting at 1
  - T1 goes first

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

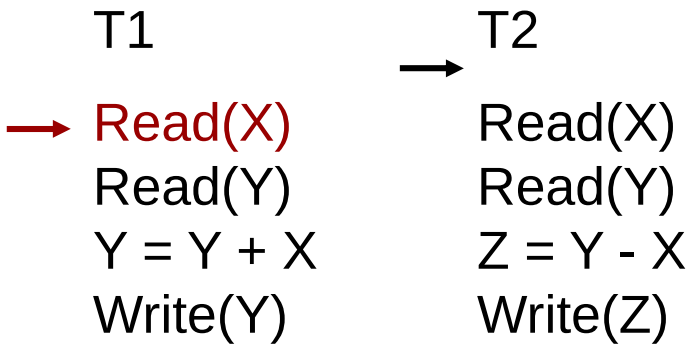
Timestamp Example

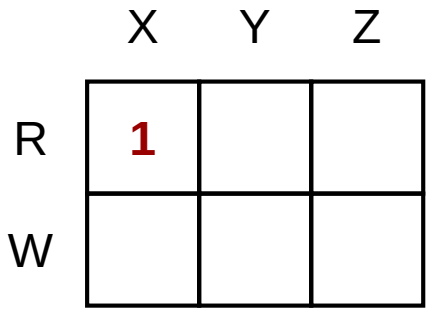
T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

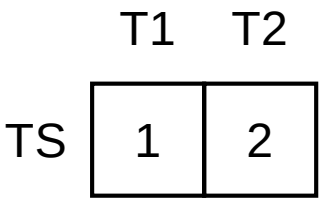




Timestamp Example

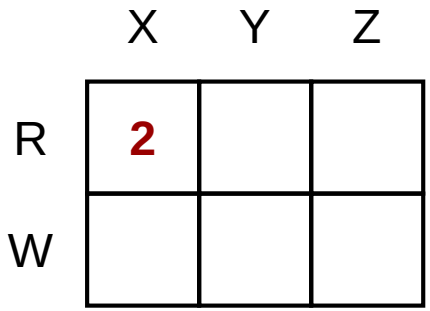


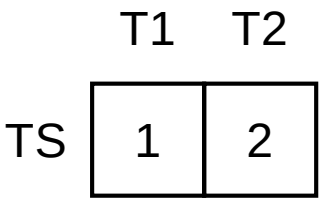




Timestamp Example

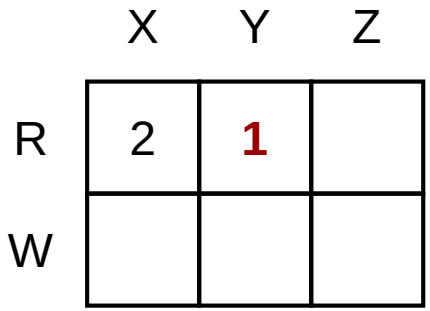
T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

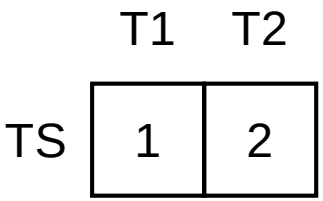




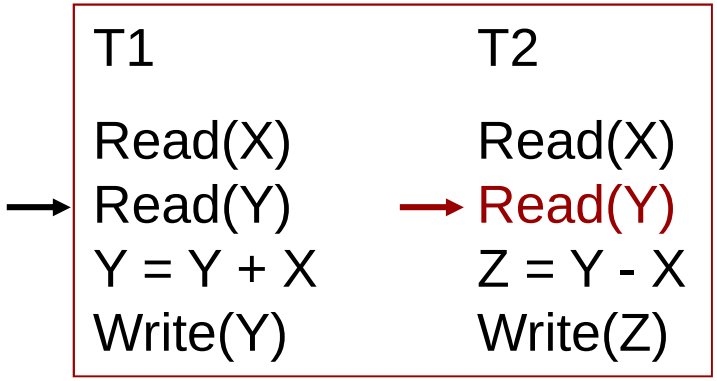
Timestamp Example

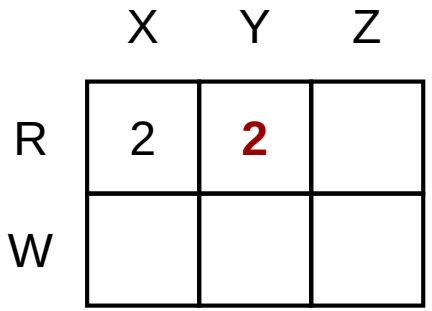
T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

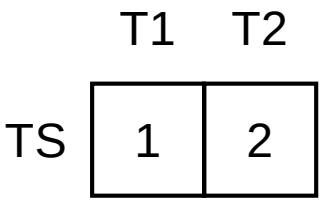




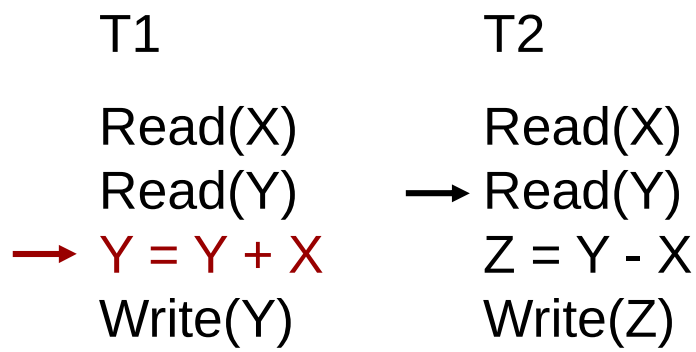
Timestamp Example

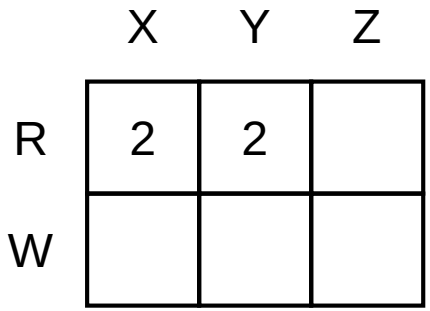


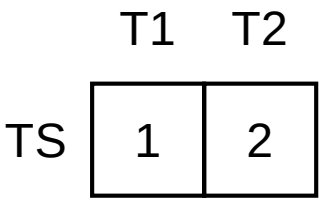




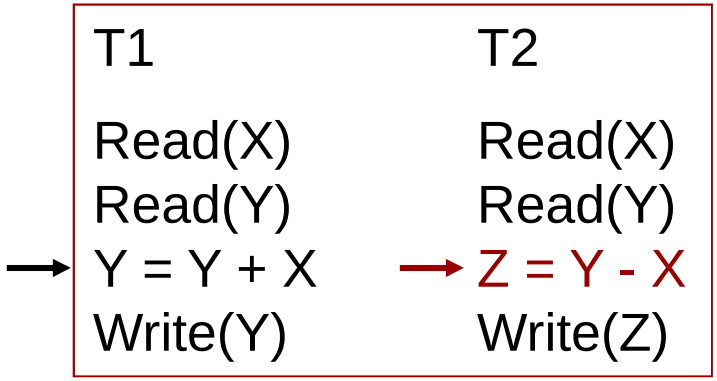
Timestamp Example

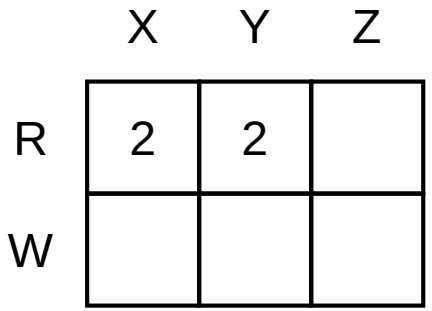


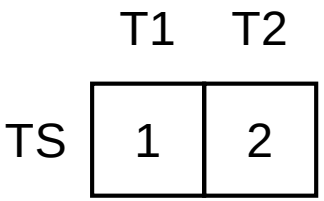




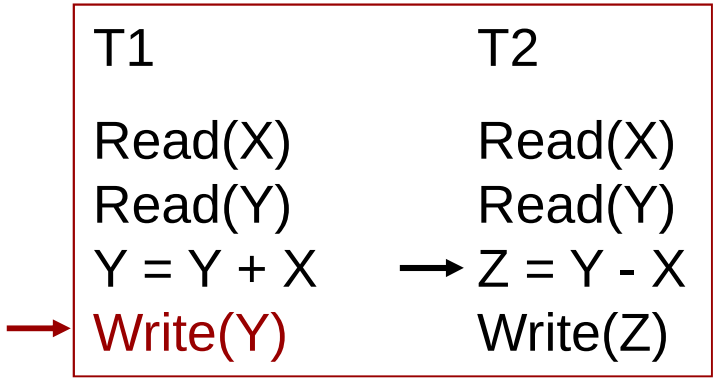
Timestamp Example

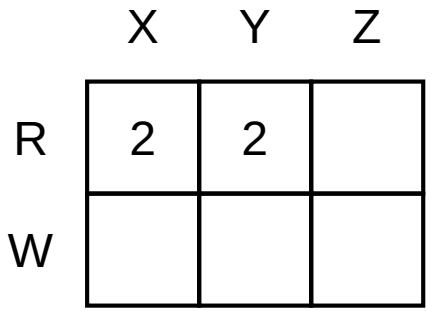


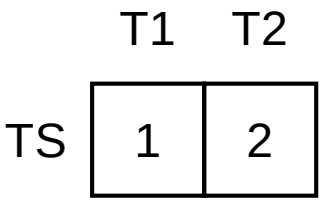




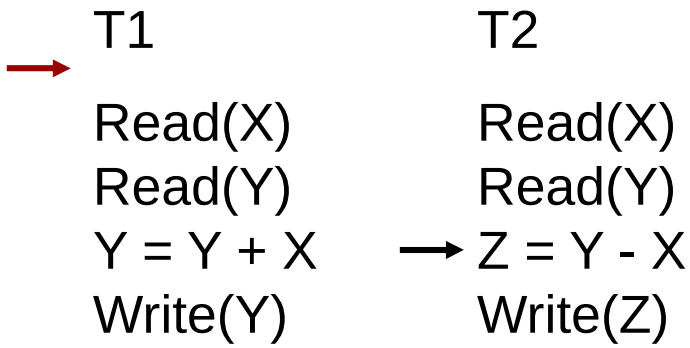
Timestamp Example

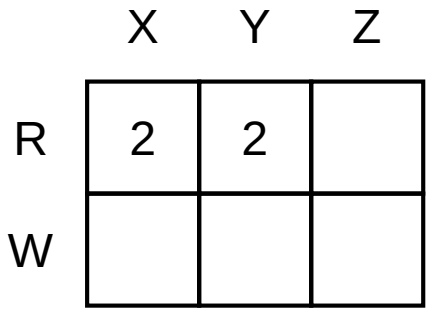


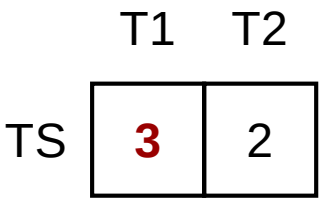




Timestamp Example

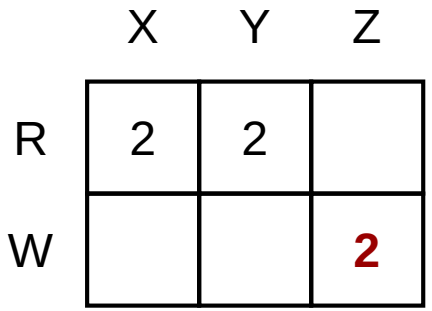


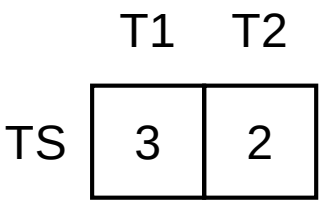




Timestamp Example

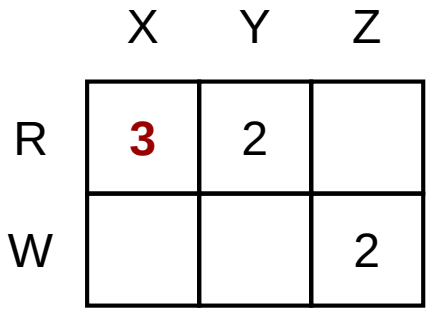
T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

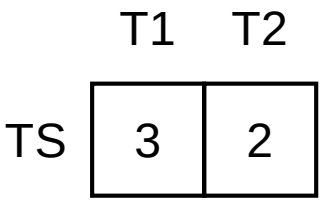




Timestamp Example

T1	T2
→ Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)





Timestamp Example

T1	T2
Read(X)	Read(X)
→ Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	**3**	
W			2

	T1	T2
TS	3	2

Timestamp Example

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
→ Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W			2

	T1	T2
TS	3	2

Timestamp Example

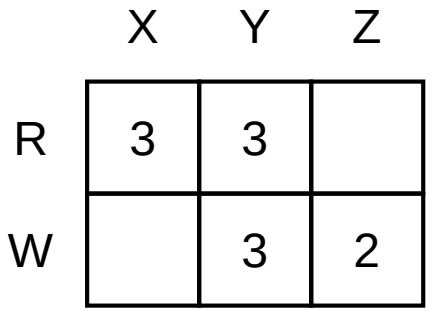
T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
→ Write(Y)	Write(Z)

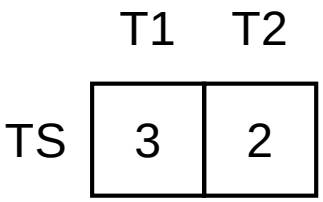
	X	Y	Z
R	3	3	
W		**3**	2

	T1	T2
TS	3	2

Timestamp Example

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)





Timestamping

- The protocol means that transactions with higher ids (start times) take precedence when conflict arises
  - No deadlock
  - When no conflict arises lower timestamps proceed first
- If a transaction tries to use a resource which has been read or written by a later transaction it gets rolled back and restarted
- Timestamping guarantees a schedule is conflict serialisable

- Problems
  - Long transactions might keep getting restarted by new transactions - starvation
  - Rolls back old transactions, which may have done a lot of work
  - Theoretically at least, two transactions could keep rolling each other back

This Lecture in Exams

Explain, using an example, how deadlock may occur between two transactions utilising two-phase locking protocol

Describe how a DBMS might attempt to prevent deadlock from occurring, and how a DMBS might recover from deadlock that has already occured

Describe how timestamping can be used as an alternative to two-phase locking, to provide concurrent access to database resources

Next Lecture

- Physical Database Design
  - RAID Arrays
  - Parity
- Database File Structures
- Indexes