COMP207
Database Development

Lecture 10

Transaction Management:
More on Combining Concurrecy and Recovery
and how to Prevent Deadlocks

Conflict-Serialisability vs Recovery

Conflict-Serialisability

- Many nice properties:
  - Equivalent to serial schedules
  - Ensure consistency / correctness
- Can be enforced by two-phase locking (2PL)

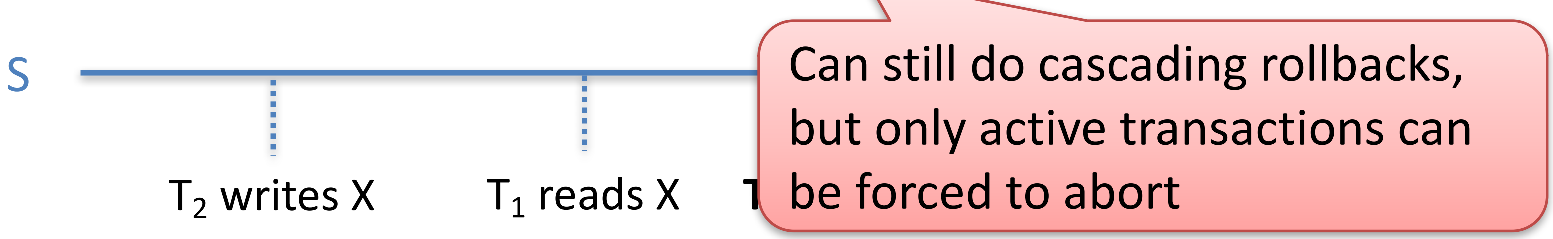
Logging and Recovery

- Suitable logging techniques ensure that we can restore desired database states
  - Undo incomplete transactions
  - Redo committed transactions
  - Undo a single or a selected number of transactions
- Robust: works even after system failures

Recoverable Schedules

- The problem for Durability in regards to cascading rollbacks occur because a transaction T_1 reads data from some transaction T_2, then T_1 commits and afterwards T_2 aborts.

- A schedule S is **recoverable** if the following is true:
  - if a transaction T_1 commits and has read an item X that was written before by a different transaction T_2, …
  - then **T_2 must commit before T_1 commits**.



Recoverable Schedules

- Additional implicit requirement:
  All log records have to reach disk in the order in which they are written.
- Compare:
  - Recoverable: S_1: $w_2(X)$; $w_1(Y)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; c_1; c_2
  - Not recoverable: S_3: $w_2(X)$; $w_1(Y)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; c_2; c_1

If in S_1 the commit record for T_2 would reach disk earlier than the commit record for T_1, then T_1 could in principle abort → cascading rollback

Cascading Rollbacks Again

- A recoverable schedule:

S_4: $w_1(X)$; $w_1(Y)$; $w_2(X)$; $r_2(Y)$; $w_2(Y)$; c_1; c_2

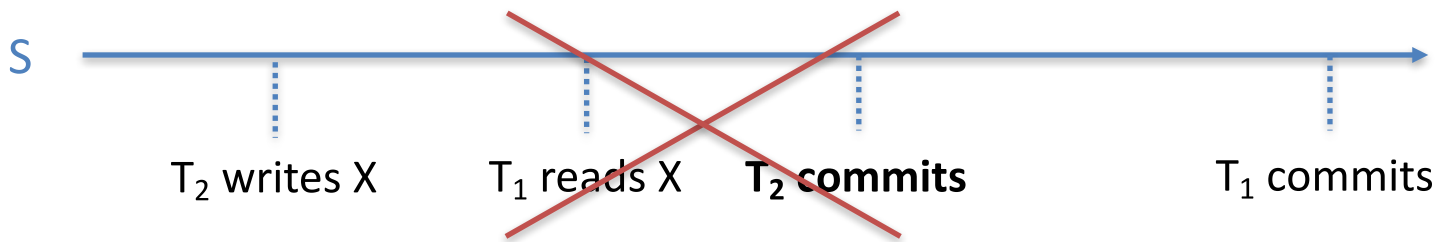
Suppose T_1 needs to be rolled back here

- T_1 rolls back → T_2 has to be rolled back

Cascadeless Schedules

- A schedule is **cascadeless** if each transaction in it reads only values that were written by transactions that have already committed.

No reading of “dirty data”.
No cascading rollbacks.



Cascadeless Schedules

- A schedule is **cascadeless** if each transaction in it reads only values that were written by transactions that have already committed.

No reading of “dirty data”.
No cascading rollbacks.

- As for recoverable schedules:
  Log records have to reach disk in the right order.

Example

- The schedules S_1-S_4 on the previous slides are **not cascadeless**:

reads uncommitted data from T_1

S_1: $w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_1; c_2$

S_2: $w_1(X); w_1(Y); w_2(X); r_2(Y); w_2(Y); c_2; c_1$

S_3: $w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_2; c_1$

S_4: $w_1(X); w_1(Y); w_2(X); r_2(Y); w_2(Y); c_1; c_2$

- This variant of S_1 is **cascadeless**:

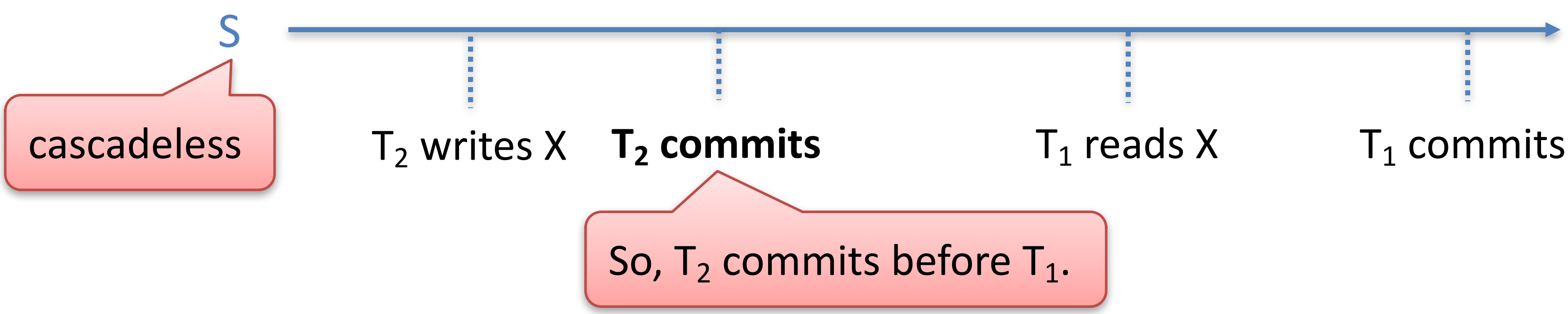
S_5: $w_2(X); w_1(Y); w_1(X); c_1; r_2(Y); w_2(Y); c_2$

reads committed data from T_1

- Note: S_5 is **not serialisable**.

Cascadeless Schedules: Properties

- Cascadeless schedules are **recoverable**:



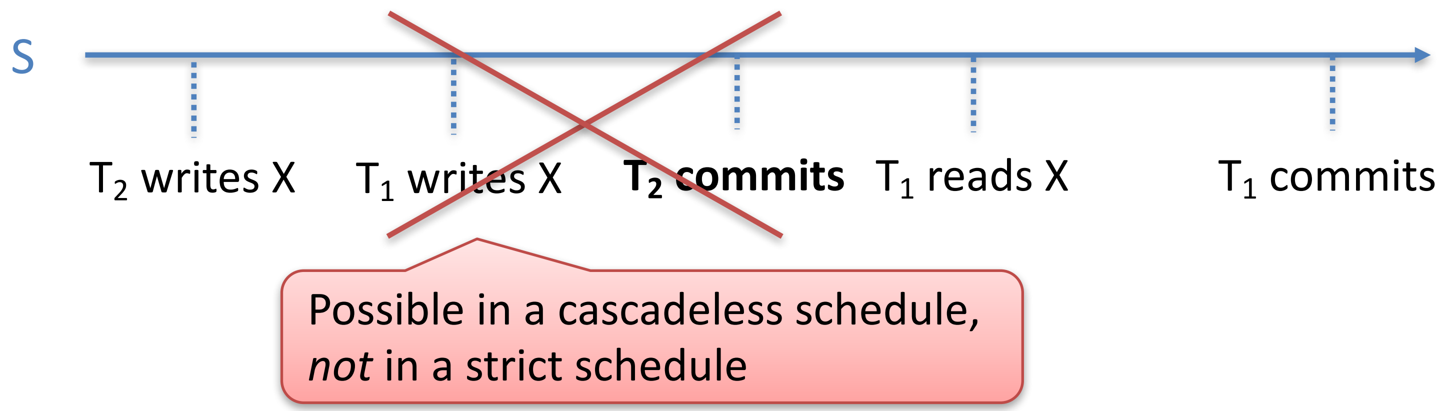
- Cascadeless schedules are in general **not serialisable**.
  (recall example on previous slide)

Can We Have Both?

No Cascading Aborts & Serialisability?

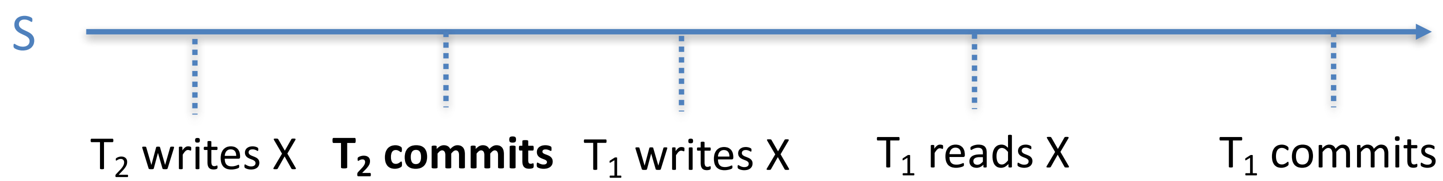
Strict Schedules

- A schedule is **strict** if each transaction in it reads and writes only values that were written by transactions that have already committed.



Strict Schedules

- A schedule is **strict** if each transaction in it reads and writes only values that were written by transactions that have already committed.



- Of course, log records have to reach disk in order.

Strict Two-Phase Locking (Strict 2PL)

- Most popular variant of two-phase locking (2PL)
- Enforces both:
  - Conflict-serialisability
  - Strict schedules
- **Strict locking** condition (in addition to 2PL condition):

> - with simple locking: **any lock**
> - with shared/exclusive locks: **just exclusive locks**

A transaction T **must not release any lock (that allows T to write data)** until:

- T has committed or aborted, and
- the commit/abort log record has been written to disk.

Example 1

2PL transaction

Not strict 2PL

Transaction T
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)
commit

For undo logging, we assume that **commit**…

1. Writes all log records to disk
2. Writes all modified database items to disk
3. Writes the commit record to disk

Example 2

2PL transaction

Transaction T
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)
commit

Strict 2PL transaction

New transaction T’
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
commit
unlock(X)
unlock(Y)

Locks released only after fully committed, and all log records written to disk

Example With Shared/Exclusive Locks

Transaction T
s-lock(X)
read_item(X)
x-lock(Y)
unlock(X)
read_item(Y)
Y := X + Y
write_item(Y)
commit
unlock(Y)

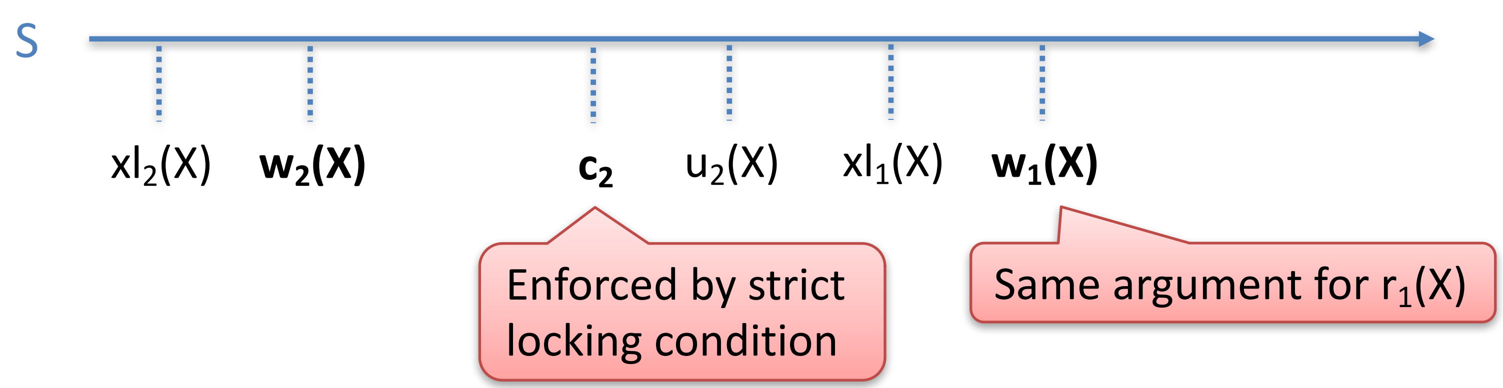
Strict 2PL

T can release the shared lock on X here (shared locks do not allow T to write data)

T is allowed to release the exclusive lock on Y only here.

Strict Two-Phase Locking *Enforces* Conflict-Serialisable & Strict Schedules

- If S is a schedule consisting of strict 2PL transactions:
  - S is conflict-serialisable.
  - S is strict.
- Strictness:



Still… Risk of Deadlocks

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
commit
unlock(X)
unlock(Y)

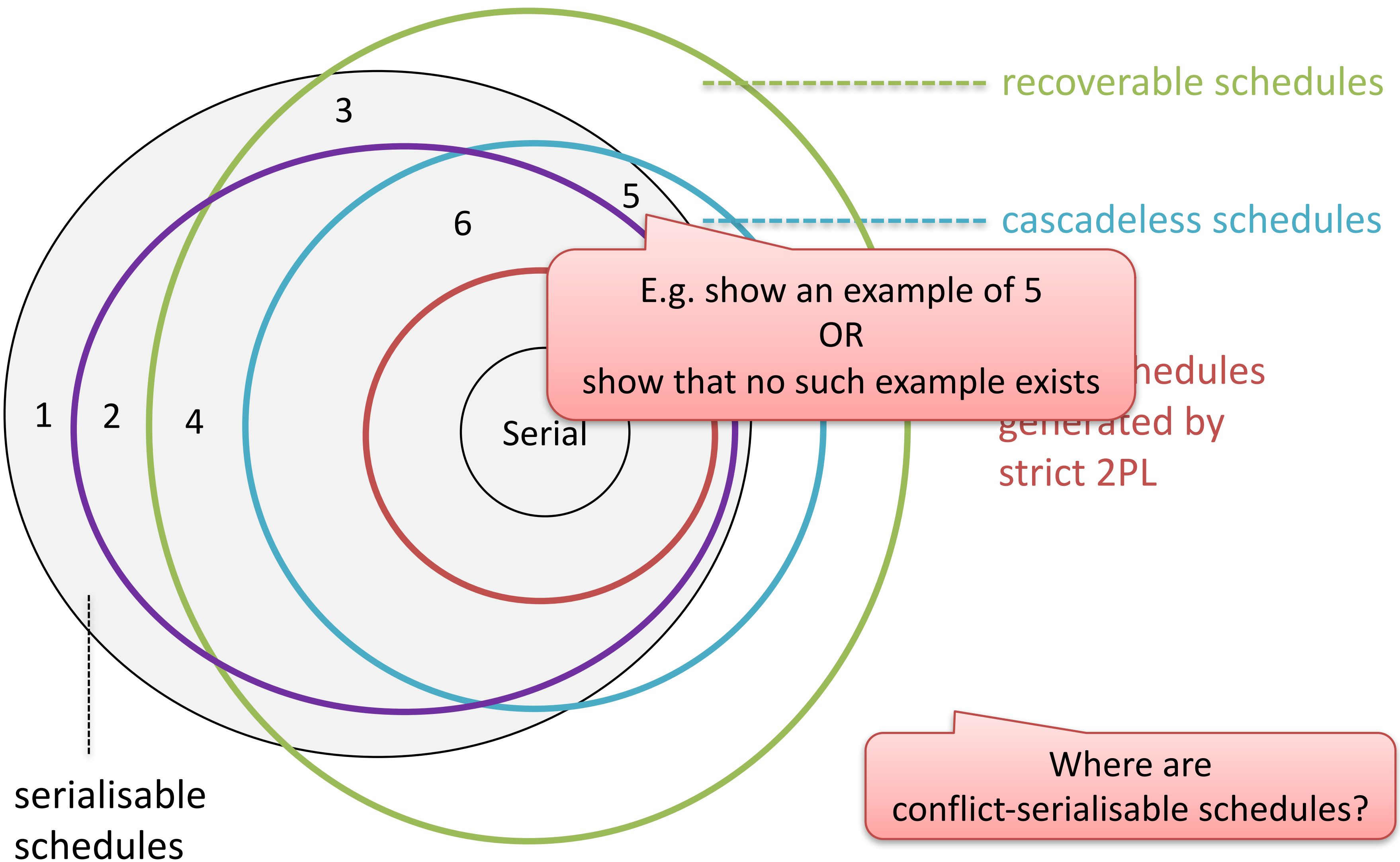
T_2
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
lock(X)
read_item(X)
X := X + 100
write_item(X)
commit
unlock(X)
unlock(Y)

T_2's request for lock on X denied

$l_1(X)$; $r_1(X)$; $w_1(X)$; $l_2(Y)$; $r_2(Y)$; $w_2(Y)$; _?_

T_1's request for lock on Y denied

How the Types of Schedules are Related



Example or proof

1. Example: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; $r_3(Y)$; $w_3(X)$; c_3; c_2; c_1
2. Example: $w_1(X)$; $w_1(Y)$; $w_2(X)$; $w_2(Y)$; $r_3(Y)$; $w_3(X)$; c_3; c_2; c_1
3. Example: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; $r_3(Y)$; $w_3(X)$; c_2; c_3; c_1
4. Example: $w_1(X)$; $w_1(Y)$; $w_2(X)$; $w_2(Y)$; $r_3(Y)$; $w_3(X)$; c_2; c_3; c_1
5. Example: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; c_2; $r_3(Y)$; $w_3(X)$; c_3; c_1
6. Example: $w_1(X)$; $w_1(Y)$; $w_2(X)$; $w_2(Y)$; c_2; $r_3(Y)$; $w_3(X)$; c_3; c_1

Strict 2PL and Deadlocks

- Strict 2PL yields **conflict-serialisable, strict schedules**
- Problem: **deadlocks**

T_1	T_2
lock(X)	lock(Y)
read_item(X)	read_item(Y)
X := X + 100	Y := Y + 100
write_item(X)	write_item(Y)
lock(Y)	**lock(X)**
…	…

$l_1(X); r_1(X); w_1(X);$

$l_2(Y); r_2(Y); w_2(Y);$ _?_

Roll back (and restart) one of the transactions

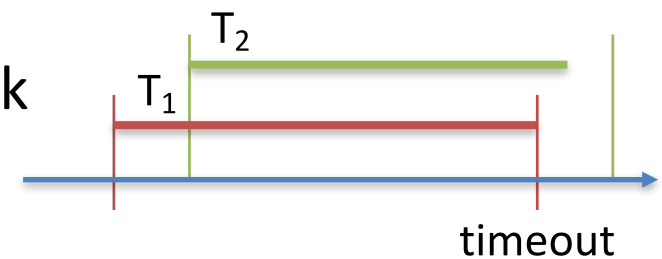
- Two approaches for deadlock prevention:
  - **Detect deadlocks & fix them**
  - **Enforce deadlock-free schedules**

Not based on (strict) 2PL

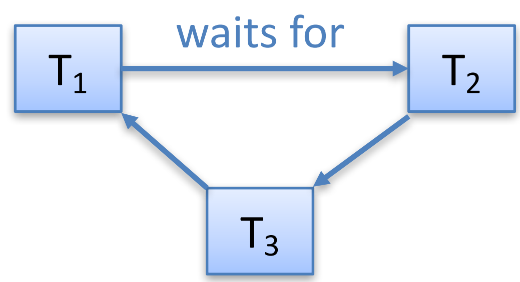
Deadlock Detection

Deadlock Detection: Approaches

- **Timeouts**
  - Assume a transaction is in a deadlock if it exceeds a given time limit



- **Wait-for graphs**
  - Nodes: transactions
  - Edge from T_1 to T_2 if T_1 waits for T_2 to release a lock
  - Deadlocks correspond to cycles



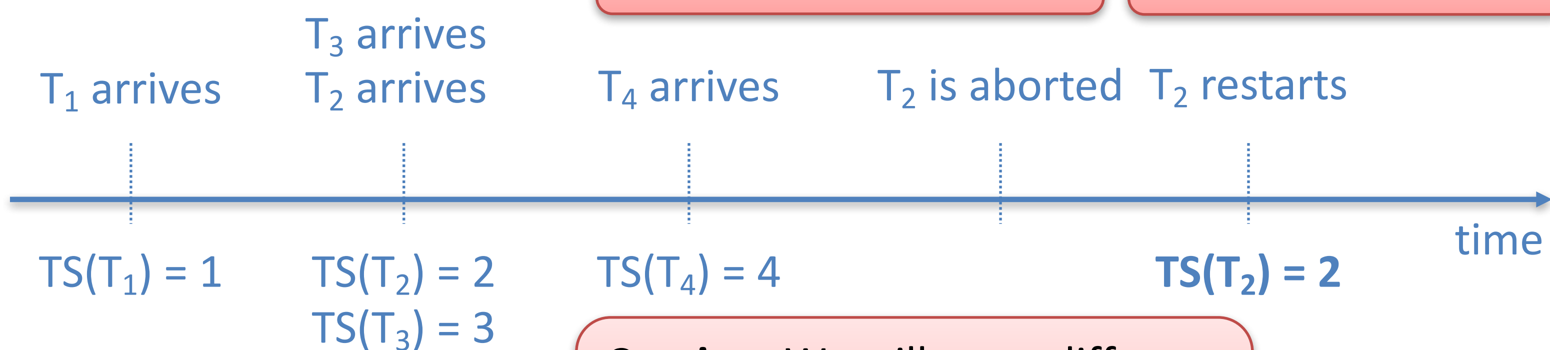
- **Timestamp-based**

Timestamps for Deadlock Detection

- Each transaction T is assigned a unique integer **TS(T)** upon arrival (the **timestamp of T**).
- If T_1 arrived earlier than T_2, we require **TS(T_1) < TS(T_2)**

"T_2 is younger than T_1"

"T_1 is older than T_2"

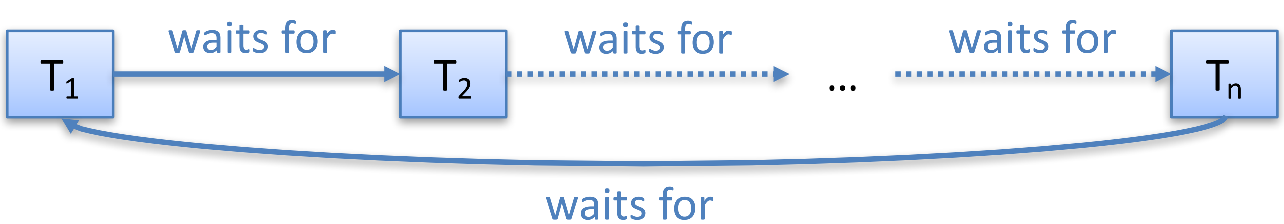


Caution: We will see a different timestamp-based method later, where timestamps *do* change.

- **Timestamps do not change** even after a restart!

How Are Timestamps Used?

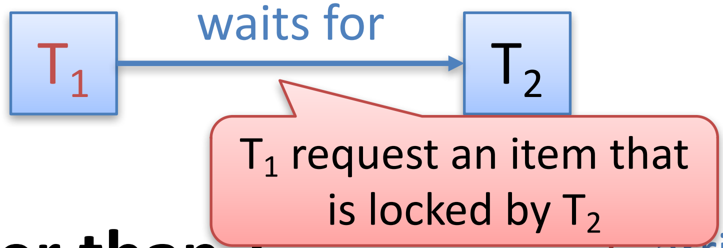
- Want to prevent cyclic dependencies such as
  - **T_1** holds a lock on X_1 and **waits for a lock on X_2**
  - **T_2** holds a lock on X_2 and **waits for a lock on X_3**
  - …
  - **T_n** holds a lock on X_n and **waits for a lock on X_1**



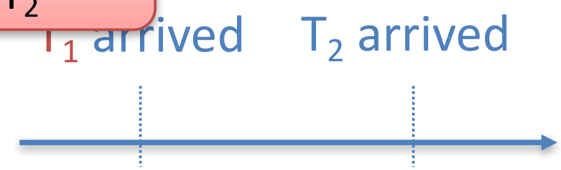
- Use timestamps to decide which transaction can wait further and which must abort to prevent deadlock

Wait-Die Scheme

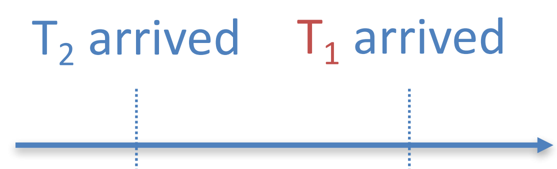
("older transactions *always* wait for unlocks")



- **Case 1: T_1 is older than T_2**
  T_1 is allowed to wait further for **T_2** to unlock



- **Case 2: T_1 is younger than T_2**
  T_1 is rolled back ("dies")

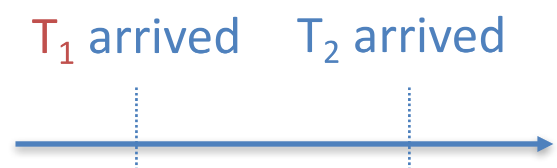


- Note: only older transactions are allowed to wait, so no cyclic dependencies created

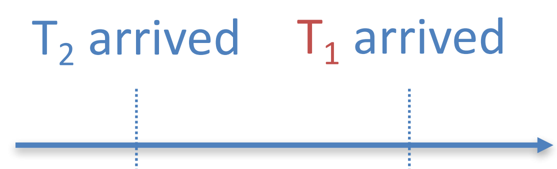
Wound-Wait Scheme

("older transactions *never* wait for unlocks")

- **Case 1: T_1 is older than T_2**
  T_2 is rolled back unless it has finished (it is "wounded")



- **Case 2: T_1 is younger than T_2**
  T_1 is allowed to wait further for **T_2** to unlock



- Note: only younger transactions are allowed to wait, so no cyclic dependencies created

To be continued…