Detecting deadlocks

Overview of this video

We are finally starting to deal with deadlocks. We will in this video see some techniques for finding them (which will also let you deal with them)

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:

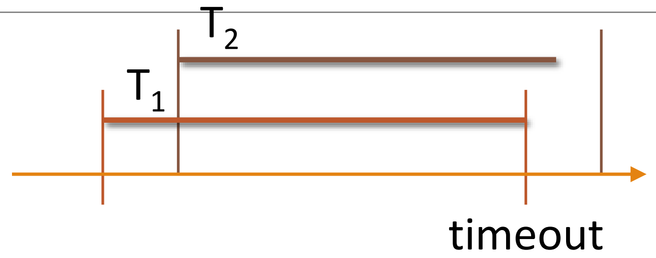
- **Detect deadlocks & fix them**
- **Enforce deadlock-free schedules**

Not based on (strict) 2PL

Deadlock Detection: Approaches

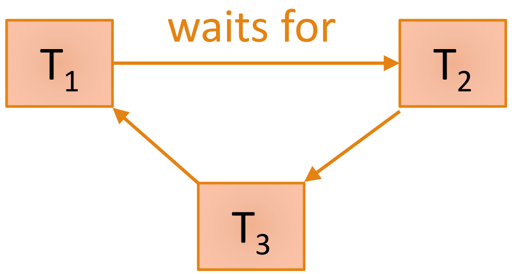
Timeouts

- 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

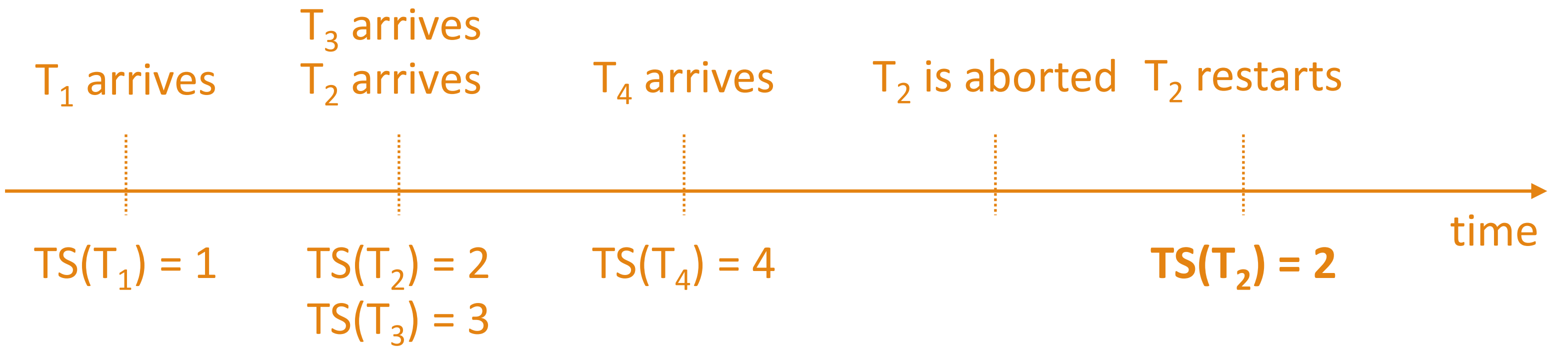
Timestamps for Deadlock Detection

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"



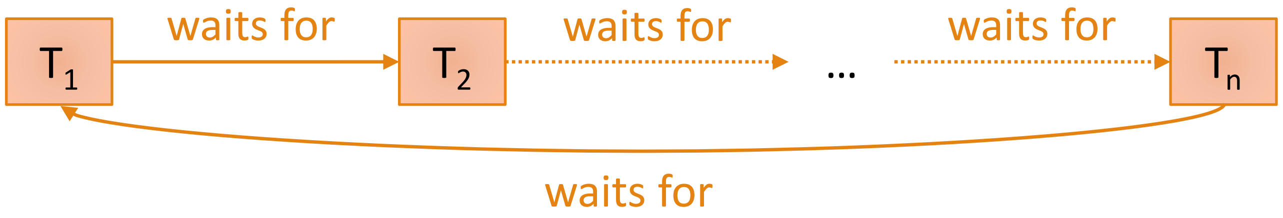
Timestamps do not change even after a restart!

Caution: We will see a different timestamp-based method in the next video, where timestamps *do* change after restart

How Are Timestamps Used?

Want to prevent cyclic dependencies such as

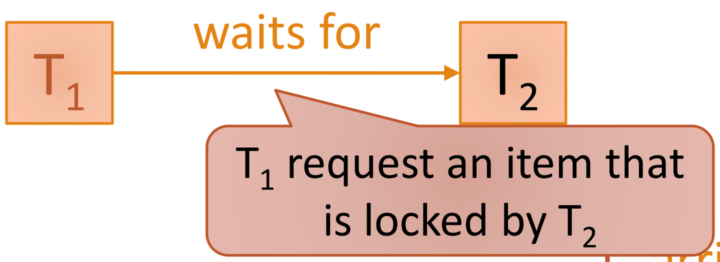
- $\mathbf{T_1}$ holds a lock on X_1 and **waits for a lock on $\mathbf{X_2}$**
- $\mathbf{T_2}$ holds a lock on X_2 and **waits for a lock on $\mathbf{X_3}$**
- …
- $\mathbf{T_n}$ holds a lock on X_n and **waits for a lock on $\mathbf{X_1}$**



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

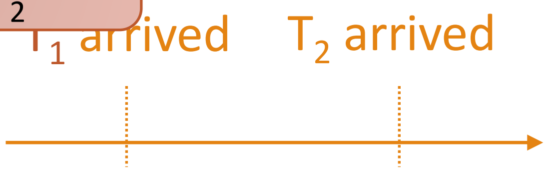
Wait-Die Scheme

("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

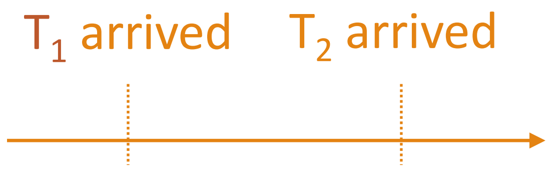
Wound-Wait Scheme

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

$$T_1 \xrightarrow{\text{waits for}} T_2$$

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

Why Wound-Wait Works

Eventually, any finite number of transactions finishes under Wound-Wait

At all times, the oldest transaction can move

Hence, eventually it finishes and there is one less transaction left and we are still doing Wound-Wait!

Wait-Die is similar, but we look at the oldest transaction or the transaction it (recursively) waits for

Summary

There are some typical approaches to finding deadlocks:

- Timeout
- Wait-for graphs
- Timestamp based approaches (Die young!):
  - Wait-Die: If T_1 is younger, abort it…
  - Wound-Wait: If T_2 is younger, abort it…