-

Terminology

- lock
  - logical synchronization of TXs
  - blocks access to db objects
- latch
  - physical synchronization of access
  - blocks access to shared data structures
- pessimistic concurrency control
  - assumes error will happen
  - thus locks schemes
  - lock-based database scheduler
  - full serialization of TXs
- optimistic concurrency control
  - assumes error will not happen
  - no locks but validation phase afterwards
    - check of conflicts
  - timestamp-based database schedulers
- mixed concurrency control
  - combines PCC and OCC
  - might return synchronization errors (deadlocks)

Serializability Therory

- **Operations of Transaction T_j**
  - Read and write operations of A by T_j: $r_j(A)$ $w_j(A)$
  - Abort of transaction T_j: a_j (unsuccessful termination of T_j)
  - Commit of transaction T_j: c_j (successful termination of T_j)

- **Schedule S**
  - Operations of a transaction **T_j are executed in order**
  - Multiple transactions may be executed concurrently

  ➔ **Schedule describes the total ordering of operations**

$$T_i \rightarrow S \leftarrow T_j$$

- **Equivalence of Schedules S1 and S2**
  - Read-write, write-read, and write-write dependencies on data object A executed in same order:

$$r_i(A) <_{S1} w_j(A) \Leftrightarrow r_i(A) <_{S2} w_j(A)$$
$$w_i(A) <_{S1} r_j(A) \Leftrightarrow w_i(A) <_{S2} r_j(A)$$
$$w_i(A) <_{S1} w_j(A) \Leftrightarrow w_i(A) <_{S2} w_j(A)$$

- **Example Serializable Schedules**
  - Input TXs
    - T1: BOT $r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ c_1
    - T2: BOT $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_2(A)$ c_2
  - Serial execution: $r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ c_1 $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_2(A)$ c_2
  - Equivalent schedules:
    - $r_1(A)$ $r_2(C)$ $w_1(A)$ $w_2(C)$ $r_1(B)$ $r_2(A)$ $w_1(B)$ $w_2(A)$ c_1 c_2
    - $r_1(A)$ $w_1(A)$ $r_2(C)$ $w_2(C)$ $r_1(B)$ $w_1(B)$ $r_2(A)$ $w_2(A)$ c_1 c_2
  - Wrong schedule: $r_1(A)$ $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ $w_2(A)$ c_1 c_2

- **Serializability Graph (conflict graph)**
  - Operation dependencies (read-write, write-read, write-write) aggregated
  - Nodes: transactions; edges: transaction dependencies
  - Transactions are serializable (via topological sort) **if the graph is acyclic**

- **Given two transactions T_1 and T_2, which pairs of the following three schedules are equivalent? Explain for each pair (S_1-S_2, S_1-S_3, S_2-S_3) why they are equivalent or non-equivalent.** [5 points]
  - $T_1 = \{r_1(a), r_1(c), w_1(a), w_1(c)\}$
  - $T_2 = \{r_2(b), w_2(b), r_2(c), w_2(c)\}$

- **Schedules**
  - $S_1 = \{r_1(a), r_1(c), w_1(a), w_1(c), r_2(b), w_2(b), r_2(c), w_2(c)\} = \{T_1, T_2\}$

    ➔ $S_1 \equiv S_2$ (equivalent, because $r_2(b)$, $w_2(b)$ independent of T_1)

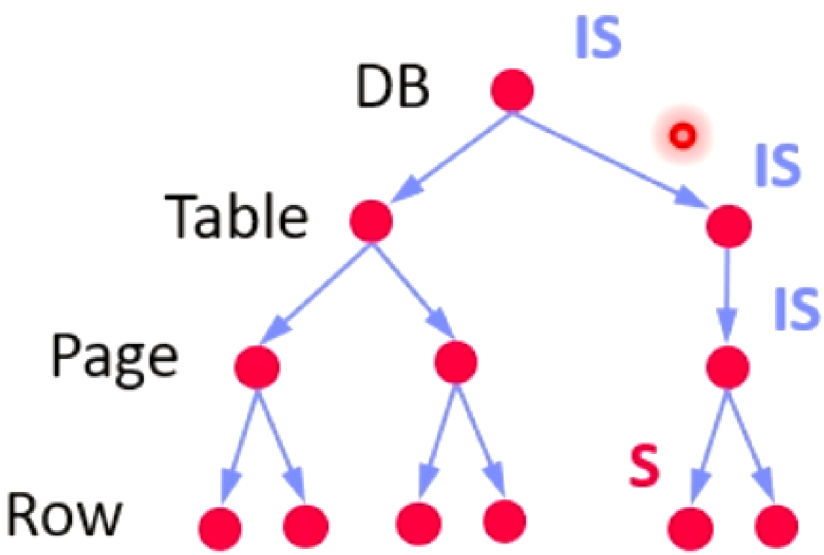
  - $S_2 = \{r_1(a), r_2(b), r_1(c), w_1(a), w_2(b), w_1(c), r_2(c), w_2(c)\}$

    ➔ $S_1 \not\equiv S_3$ (transitive)

    ➔ $S_2 \not\equiv S_3$ (non-equivalent, because $w_1(c)$, $r_2(c)$ of c in different order)

  - $S_3 = \{r_1(a), r_2(b), r_1(c), w_1(a), w_2(b), r_2(c), w_1(c), w_2(c)\}$

Locking Schemes

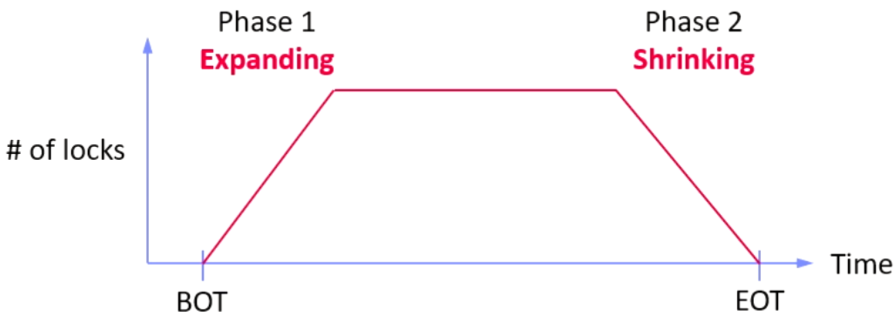
- exclusive/write x-lock
  - only current lock may write
- shared/read s-lock
  - current lock and other locks may read
- multi-granularity-locking
  - abuses hierarchy of db ojects
  - intentional x/s-lock
  - 

- lock compatibility
	None	S	X	IS	IX
S	Yes	Yes	No	Yes	No
X	Yes	No	No	No	No
IS	Yes	Yes	No	Yes	Yes
IX	Yes	No	No	Yes	Yes

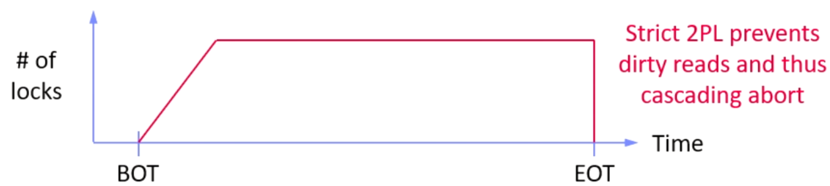
Two-Phase Locking

- concurrency protocol that guarantees serializable
- pessimistic concurrency control
- expanding phase
  - aquires locks needed by TX

- shrinking phase
  - release locks aquired by TX



-
- potential problems fixed
  - **Strict 2PL (S2PL) and Strong Strict 2PL (SS2PL)**
    - **Problem:** Transaction rollback can cause (**Dirty Read**)
    - Release all X-locks (S2PL) or X/S-locks (SSPL) **at end of transaction (EOT)**

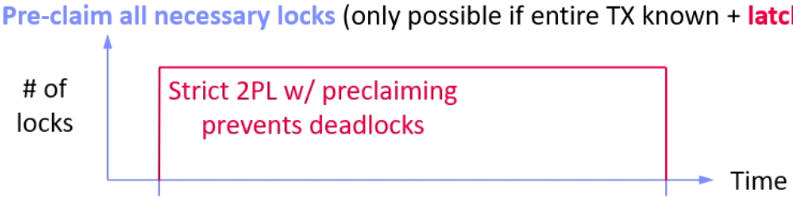


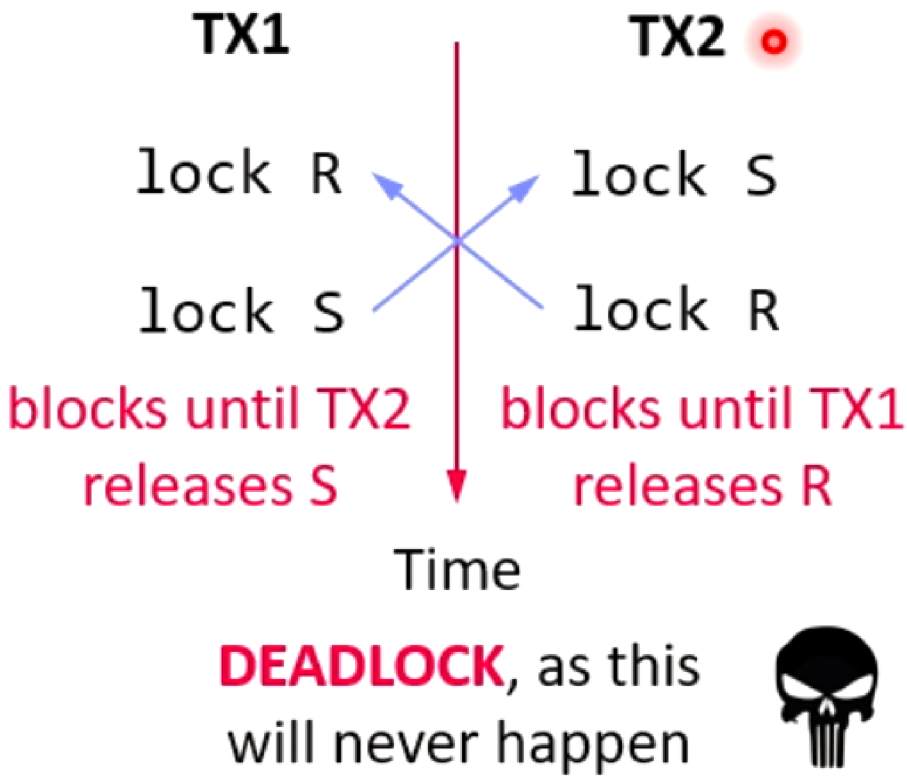
  -
  - deadlock

Strict 2PL w/ pre-claiming (aka conservative 2PL)

- Problem: incremental expanding can cause deadlocks for interleaved TXs
- **Pre-claim all necessary locks** (only possible if entire TX known + **latches**)

*





*

- **#1 Deadlock Prevention**
  - Guarantee that deadlocks can't happen
  - E.g., via pre-claiming (but overhead and not always possible)

DEADLOCK, as this will never happen

- **#2 Deadlock Avoidance**
  - Attempts to avoid deadlocks before acquiring locks via timestamps per TX
  - Wound-wait (T1 locks something held by T2 → if T1<T2, restart T2)
  - Wait-die (T1 locks something held by T2 → if T1>T2, abort T1 but keep TS)
- **#3 Deadlock Detection**
  - Maintain a wait-for graph of blocked TX (similar to serializability graph)
  - Detection of cycles in graph (on timeout) → abort one or many TXs

*

Timestamp Ordering

- optimistic concurrency control
- low overhead scheme if conflicts are rare
- TXs get timestamp at BOT
- each data object has read and write timestamp
- timestamp comparison to validate access/abort
- no locks but latches

- **Read Protocol $T_j(A)$**
  - If $TS(T_j)$ >= writeTS(A): allow read, set readTS(A) = max(TS(T_j), readTS(A))
  - If $TS(T_j)$ < writeTS(A): **abort T_j** (older than last modifying TX)
- **Write Protocol $T_j(A)$**
  - If $TS(T_j)$ >= readTS(A) AND $TS(T_j)$ >= writeTS(A): allow write, set writeTS(A)=TS(T_j)
  - If $TS(T_j)$ < readTS(A): **abort T_j** (older than last reading TX)
  - If $TS(T_j)$ < writeTS(A): **abort T_j** (older than last modifying TX)
- **BEWARE:** Timestamp Ordering requires additional handling of dirty reads, and concurrent transactions in general (e.g., via abort or versions)

Optimistic Concurrency Control

- **Read Phase**
  - Initial reads from DB, repeated reads and writes into TX-local buffer
  - Maintain **ReadSet(T_j)** and **WriteSet(T_j)** per transaction T_j
  - TX seen as read-only transaction on database
- **Validation Phase**
  - Check read/write and write/write conflicts, **abort on conflicts**
  - BOCC (Backward-oriented concurrency control) – check all older TXs T_i that finished (EOT) while T_j was running ($EOT(T_i) \geq BOT(T_j)$)
    - Serializable: if $EOT(T_i) < BOT(T_j)$ or $WSet(T_i) \cap RSet(T_j) = \emptyset$
    - Snapshot isolation: $EOT(T_i) < BOT(T_j)$ or $WSet(T_i) \cap WSet(T_j) = \emptyset$
  - FOCC (Forward-oriented concurrency control) – check running TXs
- **Write Phase**
  - Successful TXs: propagate TX-local buffer into the database and log
  - Unsuccessful TXs: discard the TX-local buffer