Scheduling and Concurrency Control

- Objectives
  - atomic execution of transactions on shared data by controlling the interleaving of concurrent accesses
- Conflicts
  - a request to access a data object meets other request from another transaction
  - one of the requests is a write access request
  - RW conflict, WR conflict, WW conflict
- Algorithms
  - two-phase locking
  - timestamp ordering
  - certifier schemes
  - integrated schemes
  - hybrid schemes

Scheduling Approaches

- Transaction Manager ↔ Scheduler ↔ Data Manager
- Options for a scheduler
  - when receiving a request from transaction manager
    1) immediately schedule it
    2) delay it (insert it into a queue)
    3) reject it (causing abort)
- Aggressive vs conservative approaches
  - optimistic vs pessimistic
  - aggressive favors immediate action (option 1);
    if impossible to finish T, abort some (option 3)
  - conservative favors option 2
  - performance trade-offs between the two
- Syntactic vs semantic correctness

Two-Phase Locking (2PL)

Assumption: each data object has a lock associated with it.

- Two locking modes
  - shared (read) lock
  - exclusive (write) lock
- Well-formed transaction
  - locks data object before accessing it
  - does not lock the same data object twice
  - unlocks all the locked objects before completion
- Some notations
  - rl(x): read lock on x
  - ru(x): unlock (release) x
  - wl(x): write lock on x
  - wu(x): unlock (release) x

Basic 2PL

1. For a request p_l(x), check if pl_l(x) conflicts with other ql_j(x) that already exists.
   - if so, delay p_l(x), forcing T_l to wait
   - if not, set pl_l(x) and send p_l(x) to data manager
2. Once pl_l(x) is set, it is not released until after data manager acknowledges that p_l(x) is processed
3. Two-phasedness
   - growing phase and shrinking phase cannot be mixed
   - once a transaction T_l starts releasing a lock, it cannot set another lock on any data object
   - to guarantee all pairs of conflicting operations of two transactions are scheduled in the same order
     (to guarantee consistency)
**Example of Simple Locking and 2PL**

\[ T_1: \quad A + 100 \rightarrow A \quad \text{and} \quad T_2: \quad A \times 2 \rightarrow A \]
\[ B + 100 \rightarrow B \quad \text{and} \quad B \times 2 \rightarrow B \]

Correctness assertion: \( A = B \)

- Well-formed, not two-phased version of \( T_1: T_1' \)
  - lock A
  - \( A + 100 \rightarrow A \)
  - unlock A
  - lock B
  - \( B + 100 \rightarrow B \)
  - unlock B

- Well-formed two-phased version of \( T_1 \) and \( T_2 \)
  \[ T_1: \quad \text{lock A} \quad \text{A + 100} \rightarrow \text{A} \quad \text{lock B} \quad \text{B + 100} \rightarrow \text{B} \]
  \[ T_2: \quad \text{lock A} \quad \text{A} \rightarrow \text{A} \times 2 \rightarrow \text{A} \quad \text{lock B} \quad \text{B} \rightarrow \text{B} \times 2 \rightarrow \text{B} \]
  \[ \text{unlock A} \quad \text{unlock A} \quad \text{unlock B} \quad \text{lock A} \quad \text{A} \rightarrow \text{A} + 100 \rightarrow \text{A} \]
  \[ \text{lock B} \quad \text{unlock B} \quad \text{unlock B} \quad \text{unlock A} \quad \text{lock B} \]

**Inconsistent Execution**

\[ T_1': \quad \text{lock A} \]
\[ T_1': \quad A + 100 \rightarrow A \]
\[ T_1': \quad \text{unlock A} \]
\[ T_2: \quad \text{lock A} \]
\[ T_2: \quad A \times 2 \rightarrow A \]
\[ T_2: \quad \text{lock B} \]
\[ T_2: \quad B \times 2 \rightarrow B \]
\[ T_2: \quad \text{unlock A} \]
\[ T_2: \quad \text{unlock B} \]
\[ T_1': \quad \text{lock B} \]
\[ T_1': \quad B + 100 \rightarrow B \]
\[ T_1': \quad \text{unlock B} \]

A: \( T_1' \rightarrow T_2 \)
B: \( T_2 \rightarrow T_1' \)

**Consistent Execution**

\[ T_1: \quad \text{lock A} \]
\[ T_1: \quad A + 100 \rightarrow A \]
\[ T_1: \quad \text{lock B} \]
\[ T_1: \quad \text{unlock A} \]
\[ T_2: \quad \text{lock A} \]
\[ T_2: \quad A \times 2 \rightarrow A \]
\[ T_2: \quad \text{lock B} \]
\[ T_2: \quad B \times 2 \rightarrow B \]
\[ T_2: \quad \text{unlock A} \]
\[ T_2: \quad \text{unlock B} \]
\[ T_1: \quad \text{lock B} \]
\[ T_1: \quad B \times 2 \rightarrow B \]
\[ T_1: \quad \text{unlock B} \]

- Locked point
  - the point at the end of the growing phase at which the transaction owns all the locks

- Equivalence
  - an execution \( L \) is equivalent to a serial execution \( L' \) in which every transaction executes at its locked point

**Correctness of Schedulers**

- Need to prove
  - all schedules representing executions that could be produced by the scheduler are serializable (SR)

- How to prove it?
  - enumerate all the possible schedules and check SR is infeasible
  - two step approach
    - characterize properties of its schedules
    - prove that any schedule with such properties are serializable

- How to characterize the properties?
  - from the specification of scheduling algorithms
Properties of Schedules by 2PL

1. If \( o_i(x) \) is in the schedule, then \( o_l_i(x) \) and \( o_u_i(x) \) are also in the schedule and \( o_l_i(x) < o_i(x) < o_u_i(x) \)

2. If \( p_i(x) \) and \( q_j(x) \) (i \( \neq \) j) are conflicting operations in the schedule, then either \( p_l_i(x) < q_l_j(x) \) or \( q_l_j(x) < p_l_i(x) \)

3. If \( p_i(x) \) and \( q_j(y) \) are in the schedule, then \( p_l_i(x) < q_l_j(y) \)
   --- from two-phaseseness

Correctness of 2PL

**Theorem:** 2PL is correct (i.e., SR)

Proof:

1. If \( T_i \rightarrow T_j \) in the schedule, then \( p_l_i(x) < q_l_j(x) \) for some \( x \).

2. If \( T_i \rightarrow T_j \rightarrow T_k \) in the schedule, then \( T_i \) releases some lock before \( T_j \) set the lock, and the same for \( T_j \) and \( T_k \).
   By induction, same for \( T_1 \) and \( T_n \) if \( T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \)

3. If the schedule has a cycle in the serialization graph \( T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1 \)
   then \( T_1 \) releases some lock before \( T_1 \) sets a lock
   --- violation of two-phaseseness, cannot be a 2PL schedule

Hence a cycle cannot exist.

Deadlocks

- Unfortunate property of locking

  \( T_1: r_1(X) \rightarrow w_1(Y) \rightarrow c_1 \)
  \( T_2: w_2(Y) \rightarrow w_2(X) \rightarrow c_2 \)
  schedule: \( r_l_1(X) \) \( w_l_2(Y) \) delay \( w_l_2(X) \) delay \( w_1(Y) \)

- Four necessary conditions for deadlock
  - mutual exclusion: one request is in exclusive mode
  - wait-for condition: holding a resource while waiting
  - no preemption
  - circular wait

- Approaches
  - prevention
  - avoidance
  - detection and resolution

Issues in Deadlock Detection and Resolution

- Time-out
  - no detection (by guessing)
  - chances of aborting transactions not involved in deadlock

- Wait-for graph (WFG) maintenance
  - precise detection
  - large overhead
  - how often should we check for a cycle in WFG?

- Victim selection
  - select the one with minimum cost
  - avoid cyclic restart
Deadlock Prevention

- Priority-based scheme
  Allow $T_i$ to be blocked (wait for) $T_j$, if $T_i$ has higher priority than $T_j$.
  Otherwise, $T_i$ is aborted.
  - deadlock is impossible: $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_l$
    implies priority($T_1$) > priority($T_2$) > \ldots > priority($T_l$)

- Potential problem of livelock (cyclic restart)
  - if a transaction uses higher priority when restarted
  - livelock is different from deadlock in that it does not prevent a transaction from execution, but it prevents the transaction from completing because of continuous abort/restart

- Avoiding livelock
  - by ensuring that a transaction will eventually have a priority high enough to complete

Wait-Die and Wound-Wait

- Timestamp
  - monotonically increasing number
  - unique
  - finite number of smaller timestamps
  - priority of a transaction is the inverse of its timestamp: older transaction $\rightarrow$ higher priority

- Scenario: $T_i$ requests a lock on which $T_j$ has a conflicting lock
  Wait-die: if $ts(T_i) < ts(T_j)$ then $T_i$ waits else abort $T_1$
  Wound-wait: if $ts(T_i) < ts(T_j)$ then abort $T_j$ else $T_1$ waits
  - terms wound, wait, and die are used from $T_i$’s viewpoint
  - in both schemes, younger transaction is aborted
  - wait-die favors younger transaction, while wound-wait favors older transactions