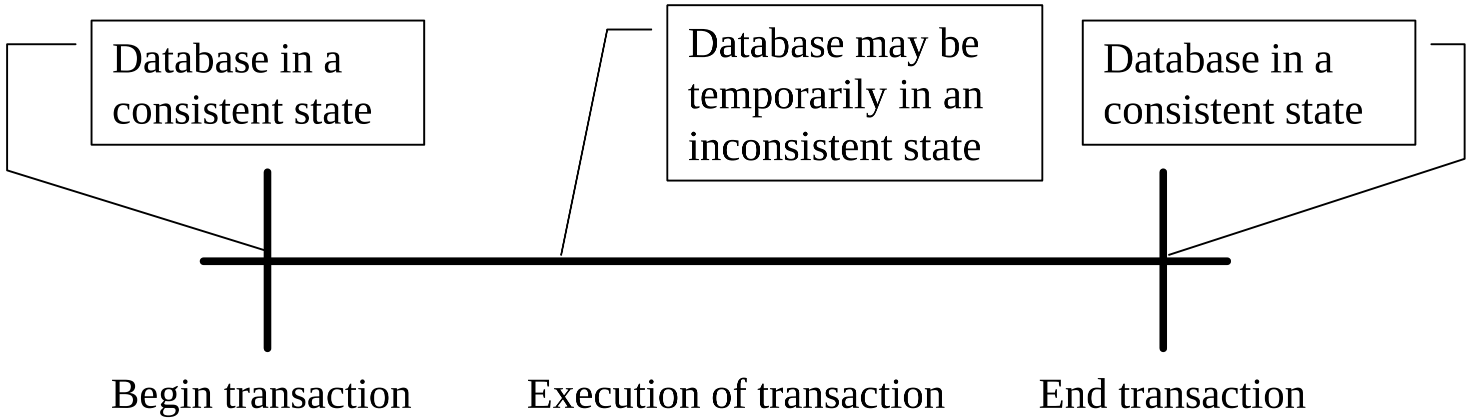
Transaction Models of DDBMS

- Topics covered:
  - Transactions
  - Characterization of transactions
  - Formalization of transactions
  - Serializability theory
  - Concurrency control models
  - Locks

Transactions

- The concept of transaction is a unit of consistent and reliable computation



- Transaction management: keeping the DB in consistent state even when concurrent accesses and failures occur

Definition of a transaction

- A transaction makes transformations of system states preserving consistency
- A transaction is a sequence of read and write operations together with computation steps, assuming that
  - the transaction may be executed concurrently with others: concurrency transparency must be provided
  - failures may occur during execution: failure transparency must be provided

Example of a transaction

- Example DB:

```
FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME, SPECIAL)
```

- Transaction

```
BEGIN_TRANSACTION RESERVATION
BEGIN
INPUT(flight_no, date, customer_name);
EXEC SQL UPDATE FLIGHT
   SET STSOLD = STSOLD + 1
   WHERE FNO = flight_no
   AND DATE = date;
EXEC SQL INSERT
   INTO FC(FNO, DATE, CNAME, SPECIAL)
   VALUES(flight_no, date, customer_name, null);
END
```

Properties of transactions

- **A**tomicity
  - all or nothing
- **C**onsistency
  - maps one consistent DB state to another
  - the ´correctness´of a transaction
- **I**solation
  - each transaction sees a consistent DB
- **D**urability
  - the results of a transaction must survive system failures
- Remember **ACID**ity

Atomicity

- Treated as a unit of operation
- Either all the actions of a transaction are completed or none of them
  - upon failure the DBMS can decide whether to terminate by completing the pending actions or terminate by undoing the actions that have been executed
- Maintainig atomicity requires recovery from failures
  - transaction failures: data errors, deadlocks, etc. $\rightarrow$ Transaction recovery
  - system failures: media, processor failures, communication breakages, etc. $\rightarrow$ Crash recovery

Classification of consistency (by Gray et al.)

- Dirty data: data values that have been written by a transaction prior to its commitment
- Degree 0 (Transaction T sees degree 0 consistency if)
  - T does not overwrite dirty data of other transactions
- Degree 1: Degree 0 plus
  - T does not commit any writes before end of transaction
- Degree 2: Degree 1 plus
  - T does not read dirty data from other transactions
- Degree 3: Degree 2 plus
  - Other transactions do not dirty any data read by T before T completes

Isolation (example)

- Possible execution schemes of T1 and T2

T1: Read(x)	T1: Read(x)
T1: x = x + 1	T1: x = x + 1
T1: Write(x)	T2: Read(x)
T1: Commit	T1: Write(x)
T2: Read(x)	T1: Commit
T2: x = x + 1	T2: x = x + 1
T2: Write(x)	T2: Write(x)
T2: Commit	T2: Commit

Reads 50 when x is 51

- Lost update: incomplete results can be seen by other transactions
- Cascading aborts: if T1 decides to abort, all transactions that have seen T1´s incomplete results must be aborted

Isolation

- An executing transaction cannot reveal it results to other concurrent transactions before its commitment
- Isolation is related to serializability: if several transactions are executed concurrently, the results must be the same as if they were executed serially in some order
- There is a strong relationship between isolation and degrees of consistency:
  - degree 0: low level of isolation, yet solves the problem of lost updates
  - degree 2: solves both lost updates and cascading aborts
  - degree 3: full isolation

Durability

- Once a transaction commits, its results are permanent and cannot be erased even if system failure occurs
- Database recovery

Termination of transactions

- A transaction <u>always terminates</u>
  - if the task is successful: commits
  - if the task is incomplete (for some reasons): aborts
    - either due to system failure or unsatisfied conditions
    - rollback: undone the actions and return the DB to its state before execution
- Commit
  - the point of no return
  - if a transaction is committed
    - its results are permanently stored in the DB $\rightarrow$ durability
    - its results can be made visible to other transactions $\rightarrow$ consistency, isolation

Example of termination

```
BEGIN_TRANSACTION RESERVATION
BEGIN
    INPUT(flight_no, date, customer_name)
    EXEC SQL SELECT STSOLD, CAP
        INTO temp1, temp2
        FROM FLIGHT
        WHERE FNO = flight_no
        AND DATE = date;
    IF temp1 = temp2 THEN
        BEGIN
            OUTPUT(„no free seats“);
            ABORT
        END
    ELSE BEGIN
        EXEC SQL UPDATE FLIGHT
            SET STSOLD = STSOLD + 1
            WHERE FNO = flight_no
            AND DATE = date;
        EXEC SQL INSERT
            INTO FC(FNO, DATE, CNAME, SPECIAL)
            VALUES(flight_no, date, customer_name, null);
        COMMIT;
        OUTPUT(„reservation completed“);
    END
END
```

Formalization of the transaction concept

- Characterization
  - Data items that a given transaction
    - reads: Read Set (RS)
    - writes: Write Set (WS)
    - they are not necessarily mutually exclusive
    - Base Set (BS): BS = RS $\cap$ WS
- Insertion and deletion are omitted, the discussion is restricted to static databases

Formalization of the transaction concept

- $O_{ij}(x)$: some atomic operation O_j of transaction T_i that operates on DB entity x
- $O_j \in$ {read, write}
- $OS_i = \cup_j O_{ij}$, i.e. all operations in T_i
- $N_i \in$ {abort, commit}, the termination condition for T_i
- Transaction T_i is a partial ordering over its operations and the termination condition

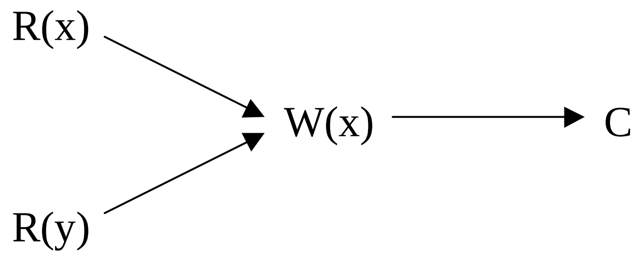
Formalization of the transaction concept

- Partial order $P = \{\Sigma, \prec\}$ where
  - Σ is the domain
  - $\prec$ is an irreflexive and transitive relation
- Transition T_i is a partial order $\{\Sigma_i, \prec_i\}$ where
  - $\Sigma_i = OS_i \cup N_i$
  - For any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij}=R(x)$ and $O_{ik}=W(x)$ for any data item x then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$, i.e. ´there must be an order between conflicting operations´
  - $\forall O_{ij} \in OS_i, O_{ij} \prec_i N_i$, i.e. áll operations must precede the termination´
- The ordering relation $\prec_i$ is application dependent

Formalization of the transaction concept

- Example

  Read(x)
  Read(y)
  x = x + y
  Write(x)
  Commit



  - $\Sigma = \{R(x), R(y), W(x), C\}$
  - $\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$ where (O_i, O_j) means $O_i \prec O_j$
- Partial order: the ordering is not specified for every pair of operations

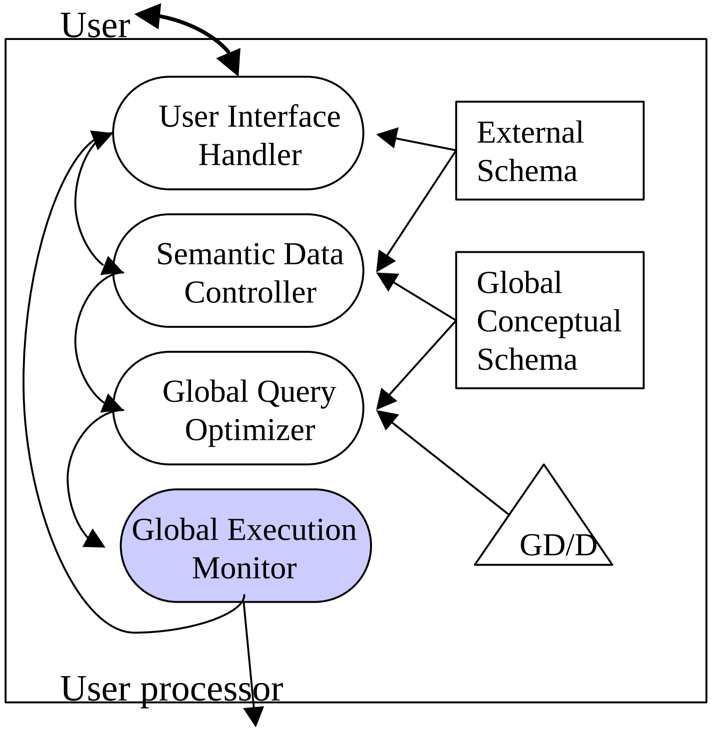
Characterization of transactions

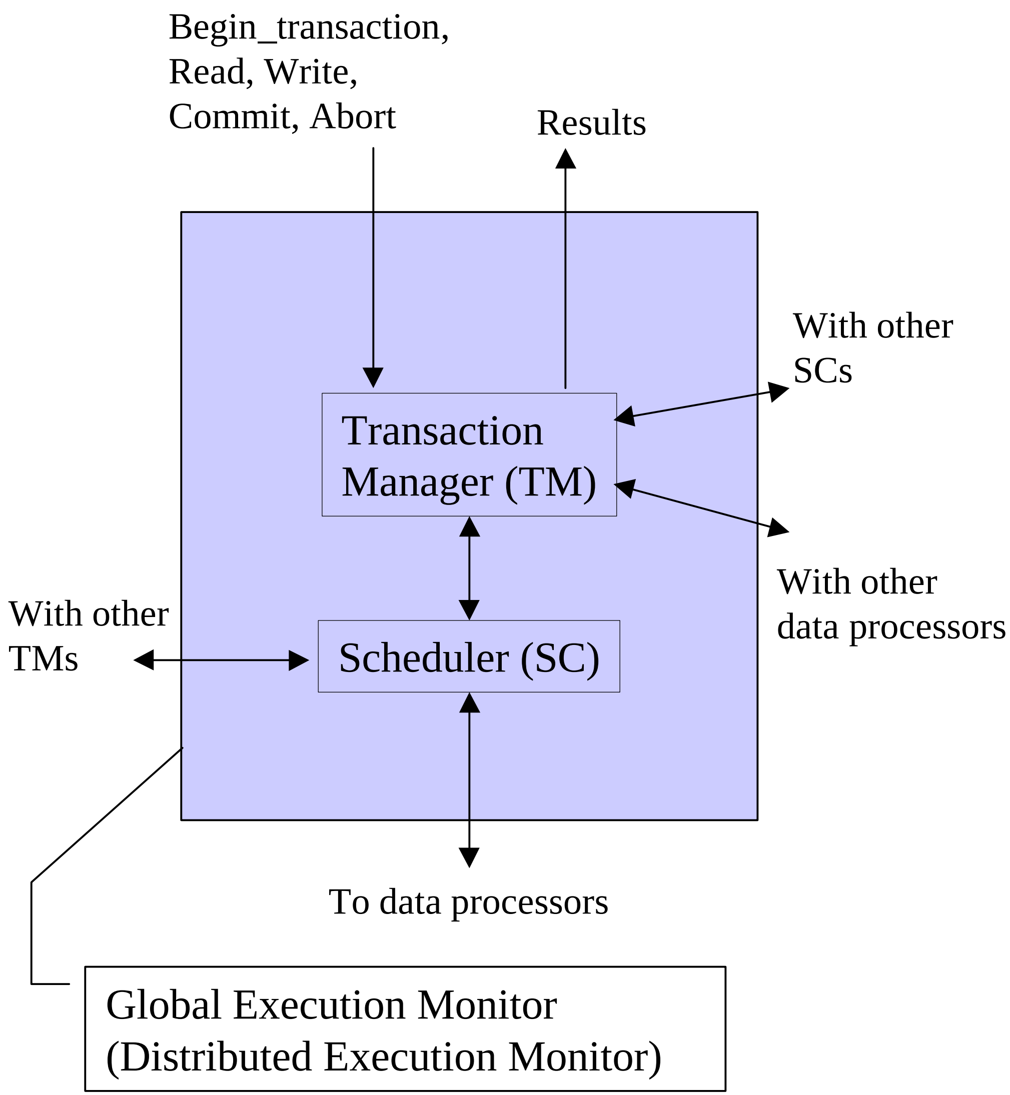
- According to application type
  - regular or distributed
  - compensating
  - heterogeneous
- According to duration
  - on-line (short life) or batch (long life)
- According to structure
  - flat, nested or workflow
- According to the order of read and write operations
  - general
  - two-step: all read ops before any write ops
  - restricted: a data item must be read before written
  - restricted two-step
  - action: restricted where each read-write pair is atomic

Structural types of transactions

- Flat
  - a sequence of primitive operations between begin and end markers
- Nested
  - a transaction may include other transactions with their own commit points
    - more concurrency introduced
    - recovery is possible independently for each subtransaction
  - a subtransaction can be a nested one too
  - nesting
    - open
      - subtransactions begin after their parents and finish before them
      - commitment is conditional upon the commitment of the parent
    - closed
      - subtransactions can execute and commit independently
      - compensation may be necessary

Architecture revisited





Serializability theory

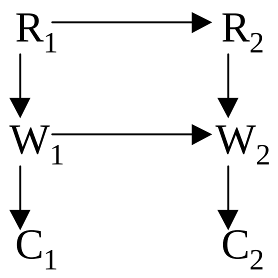
- Schedule (history) S: specifies an interleaved execution order over a set of transactions $T=\{T_1, T_2, \ldots T_n\}$
- Complete schedule S_T^c: is a partial order $S_T^c = \{\Sigma_T, \prec_T\}$ over a set of transactions $T=\{T_1, T_2, \ldots T_n\}$ that defines the execution order of all operations in its domain
  - $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$
  - $\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$
  - for any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$ or $O_{kl} \prec_T O_{ij}$

Serializability theory

- Schedule (example): a possible complete schedule
  - T1: T2:

T1	T2
Read(x)	Read(x)
x = x + 1	x = x + 1
Write(x)	Write(x)
Commit	Commit

  - $\Sigma_1 = \{R_1(x), W_1(x), C_1\}$, $\Sigma_2 = \{R_2(x), W_2(x), C_2\}$
  - $\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$
  - $\prec_T = \{(R_1, R_2), (R_1, W_1), (R_1, C_1), (R_1, W_2), (R_1, C_2), (R_2, W_1), (R_2, C_1), (R_2, W_2), (R_2, C_2), (W_1, C_1), (W_1, W_2), (W_1, C_2), (C_1, W_2), (C_1, C_2), (W_2, C_2)\}$



Serializability theory

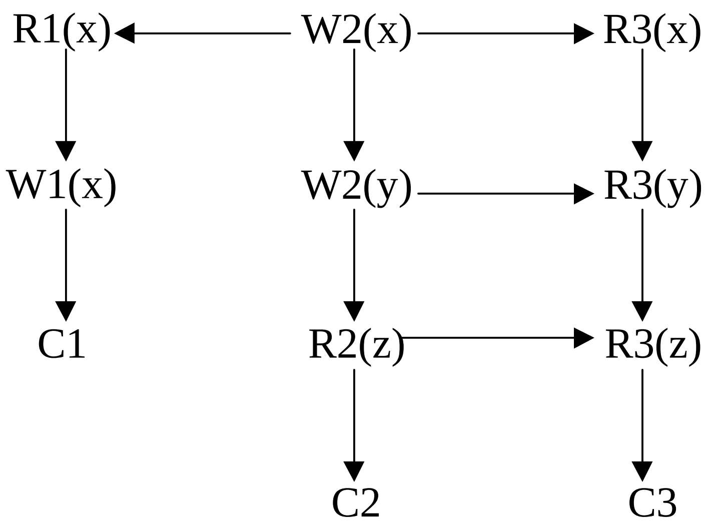
- Prefix: $P' = \{\Sigma', \prec'\}$ is a prefix of partial order $P = \{\Sigma, \prec\}$ if
  - $\Sigma' \subseteq \Sigma$
  - $\forall e_i \in \Sigma'$, $e_1 \prec' e_2$ iff $e_1 \prec e_2$
  - $\forall e_i \in \Sigma'$, if $\exists e_j \in \Sigma$ and $e_j \prec e_i$, then $e_j \in \Sigma'$
- Only the conflicting operations are relevant at scheduling - redefine schedule:
- Schedule (incomplete) S: is a prefix of complete schedule S_T^c

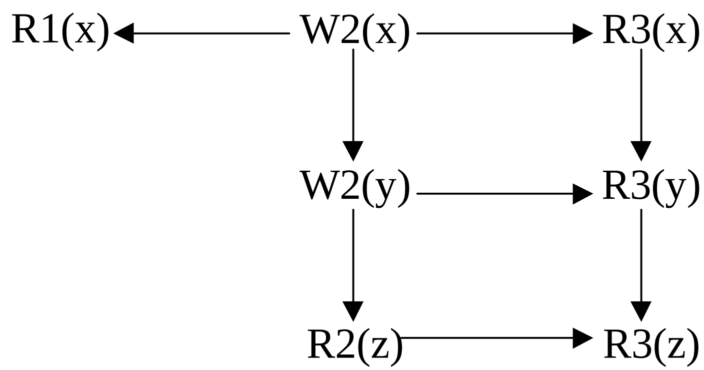
Serializability theory

- Incomplete schedule (example)
  - T1: T2: T3:

T1	T2	T3
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

- Complete schedule Partial schedule
  - the partial schedule is a prefix of complete schedule and equivalent to it





Serializability theory

- **Serial schedule** (serial history): if in a schedule S the operations of various transactions are not interleaved, the schedule is serial
  - $S = \{W_2(x), W_2(y), R_2(z), C_2, W_1(x), R_1(x), C_1, R_3(x), R_3(y), R_3(z), C_3\}$
  - $T_2 \prec_S T_1 \prec_S T_3$
- Two schedules S_1 and S_2 are **equivalent** if for each pair of conflicting operations O_{ij}, O_{kl} ($i \neq k$) whenever $O_{ij} \prec_1 O_{kl}$ then $O_{ij} \prec_2 O_{kl}$. (*conflict equivalence*)
- Schedule S is **serializable** if it is conflict equivalent to a serial schedule (*conflict-based serializability*)

Serializability theory

- Transactions execute concurrently but the overall effect of the resulted history upon the database is equivalent to some serial scheduling
- Primary goal of concurrency control: generate a serializable schedule for the pending transactions
- Two histories must be taken into account:
  - local schedule (at each site)
  - global schedule

Serializability theory

- When the DB is partitioned, if each local schedule is serializable then the global schedule is serializable
- When the DB is replicated, the global schedule is serializable (one-copy serializable) if
  - local schedules are serializable
  - two conflicting operations are in the same relative order in each local schedule where they appear

Replica control protocol

- Consistency in presence of replication: one-copy serializability must be provided
  - concurrency control plus
  - replica control
- Assume data item x (logical data) is replicated as x_1, x_2, ... x_n (physical data items)
  - each read(x) is mapped to one of the physical items
  - each write(x) is mapped to a subset of the physical data copies
- If read(x) is mapped to one and write(x) is mapped to all physical copies, it is a read-once/write-all (ROWA) protocol

Concurrency control models

- Pessimistic
  - 2-Phase Locking based (2PL)
    - Centralized
    - Primary copy
    - Distributed
  - Timestamp Ordering (TO)
    - Basic
    - Multiversion
    - Conservative
  - Hybrid
- Optimistic
  - Locking
  - Timestamp ordering

Locks

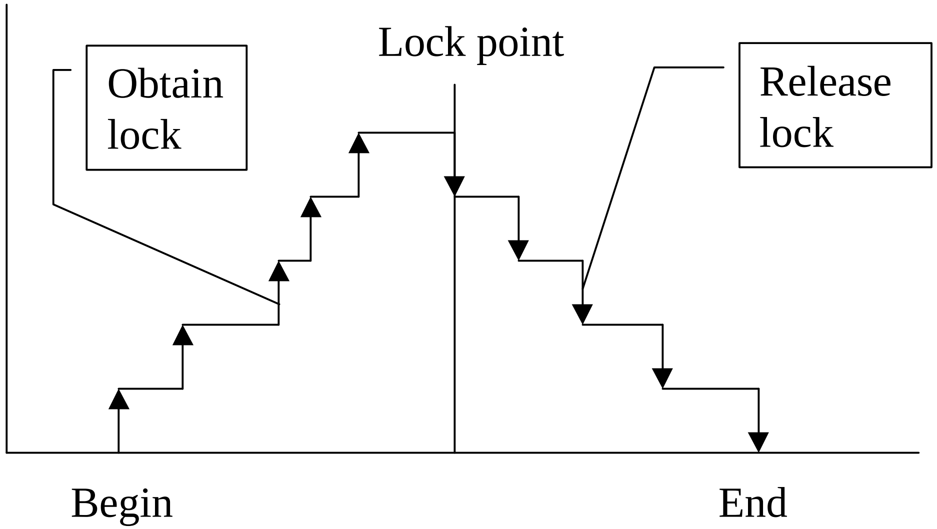
- Locks ensure that data shared by conflicting operations are accessed by one operation at a time - a simple way of serialization
- The lock is
  - set by a transaction before the lock unit is accessed
  - reset at the end of the operation
  - if the lock is set already, the lock unit cannot be accessed
- Lock modes
  - read lock (shared lock)
  - write lock (exclusive lock)

	Read lock (x)	Write lock (x)
Read lock (x)	compatible	not compatible
Write lock (x)	not compatible	not compatible

- Locks are controlled by the Lock Manager (LM) which is a part of the Scheduler (see architecture revisited)

Locks

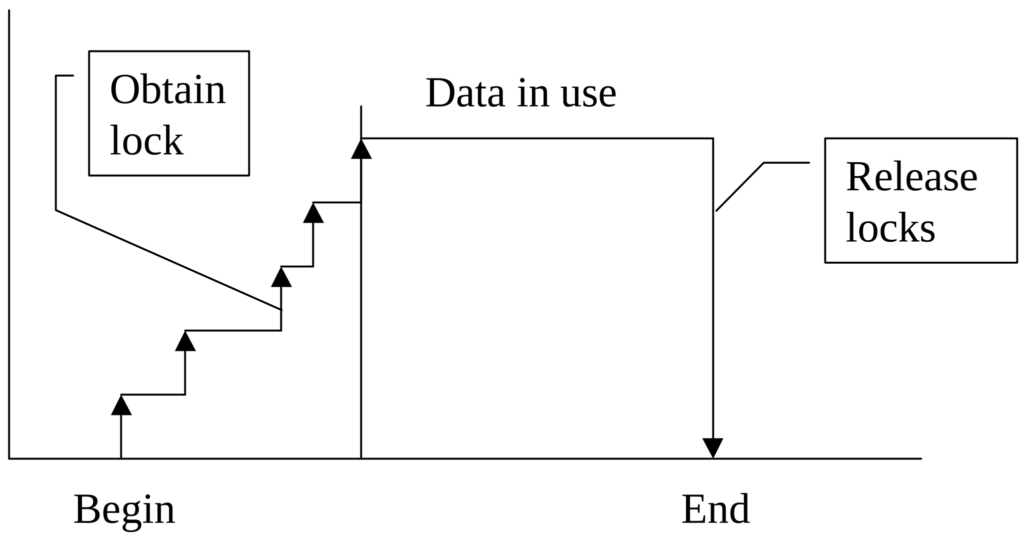
- Two-phase locking (2PL): no transaction should request a lock after it releases one of its locks
- Transactions have
  - growing phase
  - lock point
  - shrinking phase



- Theorem: any schedule that obeys 2PL rule is serializable (Eswaran et al.)
- Difficult to implement Transaction Manager (among others due to cascading aborts)

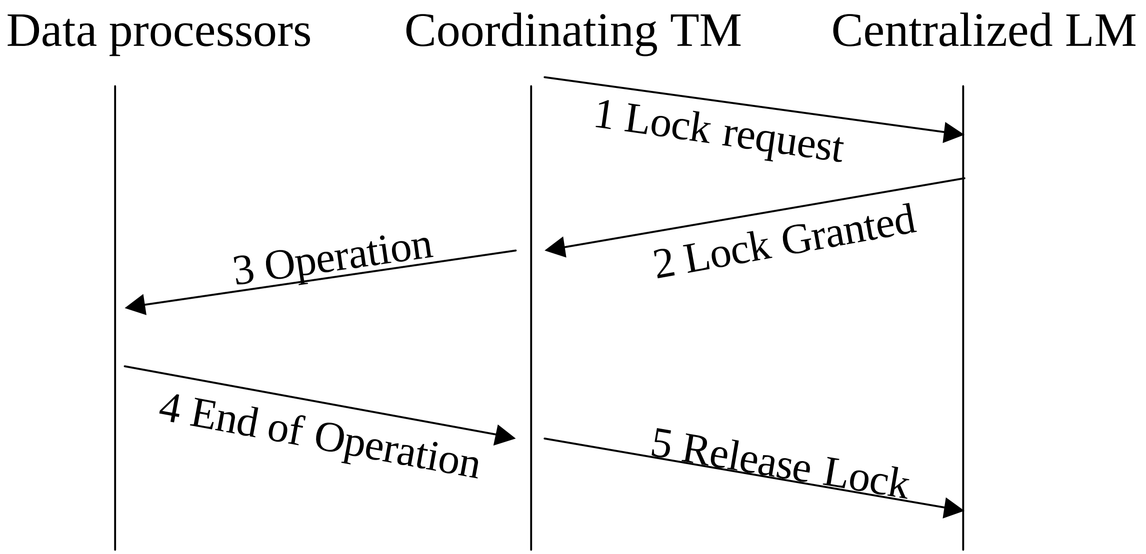
Locks

- Strict two-phase locking (S2PL): locks are released if the operation is a commit or an abort



Locks in distributed DBSs: Centralized 2PL

- There is only one 2PL scheduler (lock manager) in the distributed system
- All lock requests are addressed to it



- Important: TM must implement the replica control protocol

Locks in distributed DBSs: Primary copy 2PL

- The centralized 2PL scheduler may form a bottleneck
- In PC2PL lock managers are implemented at a number of sites
  - they are responsible for a given set of lock units
  - TMs send lock and unlock requests to the scheduler that is responsible for the given lock unit
  - one copy of the data item is treated as a primary copy
  - the location of the primary copy must be determined prior to sending lock and unlock requests - a directory design issue

Locks in distributed DBSs: Distributed 2PL

- LMs are available at each site in D2PL
  - if the DB is not replicated, it is the same as PC2PL
  - if replicated, it implements the ROWA protocol
  - operations are passed via LMs - there is no lock granted message