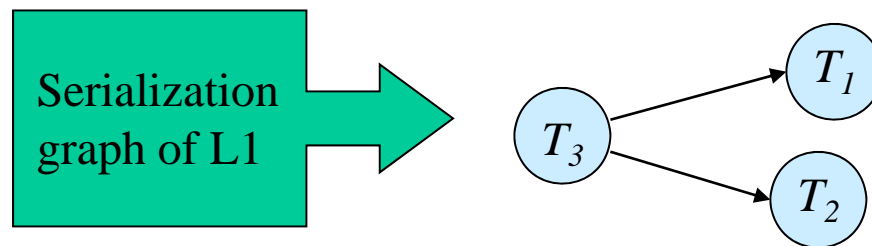


# Serial Logs

- *Serial* log:
  - a log in which all actions of a transaction terminate before any action of the next transaction starts
  - example: L2 is a serial log
- *Serializable* log:
  - a log in which actions from several transactions  $T_0, T_1, \dots, T_n$  are interleaved, and that has the same output and the same effect on the database as the serial execution of a permutation of  $T_0, T_1, \dots, T_n$
  - Example: log L1 is equivalent to serial log L2
- A serializable log is equivalent to a serial log, thus represents a correct execution
  - question: is there some criterion for telling if a log is serializable?

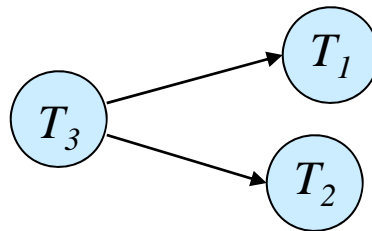
# Serialization Graph

- Let  $L$  be a log over a set of transactions  $T_0, T_1, \dots, T_n$ 
  - the *serialization graph*  $SG(L)$  of  $L$  is a directed graph in which the nodes are the transactions  $T_i$ , and whose edges satisfy the condition:
    - edge  $T_i \rightarrow T_j$  then for some  $x$  either  $ri[x] < wj[x]$  or  $wi[x] < rj[x]$  or  $wi[x] < wj[x]$
    - i.e. an edge  $T_i \rightarrow T_j$  denotes a conflict between actions of  $C$  and  $T_j$
  - example:



# The Serializability Theorem

- Th.: A log  $L$  is serializable iff  $SG(L)$  is acyclic
  - no cycles in  $AG(L) \Leftrightarrow L$  is serializable
- The serial log corresponding to  $L$  can be determined by topologically sorting  $AG(L)$ 
  - example:



$T_3 \rightarrow T_1 \rightarrow T_2$  OR  $T_3 \rightarrow T_2 \rightarrow T_1$

# Concurrency Control Algorithms

- Given a number of conflicting transactions, the serializability theory provides
  - criteria to study the correctness of a possible schedule
  - it does not provide a practical way to produce the schedule
- Need concurrency control algorithms
- Two main classes of algorithms:
  - lock based
  - timestamp based

# Lock Based Algorithms

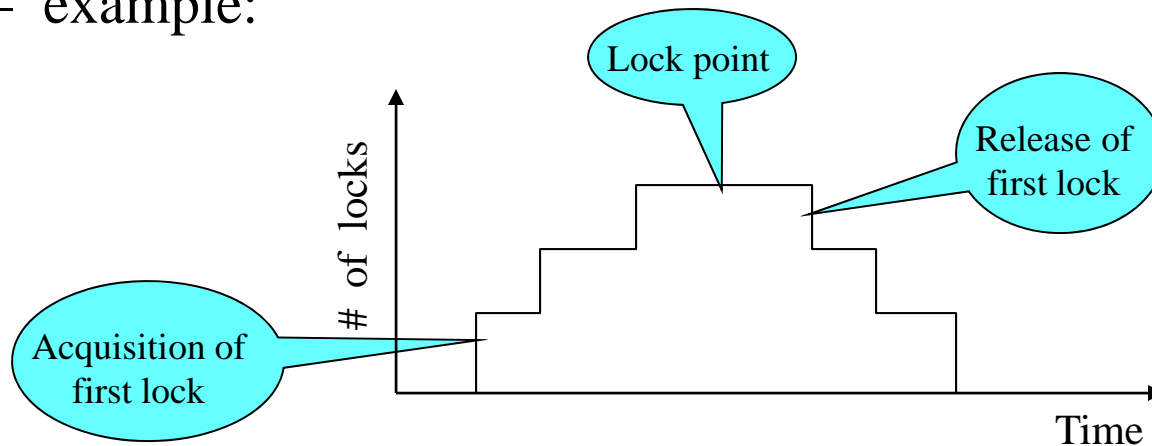
- In these algorithms, transactions must lock data objects before accessing them
- A transaction is *well-formed* if it
  - locks a data before accessing it
  - does not lock a data object more than once, and
  - unlocks all the locked data objects before completing

# Static locking

- A transaction acquires the lock on all the objects it needs at start of execution
  - release all the locks at end of execution
- Pros:
  - very simple
- Cons:
  - zero concurrency between transactions with a conflict
  - requires a priori knowledge of all the data needed

# Two-Phase locking

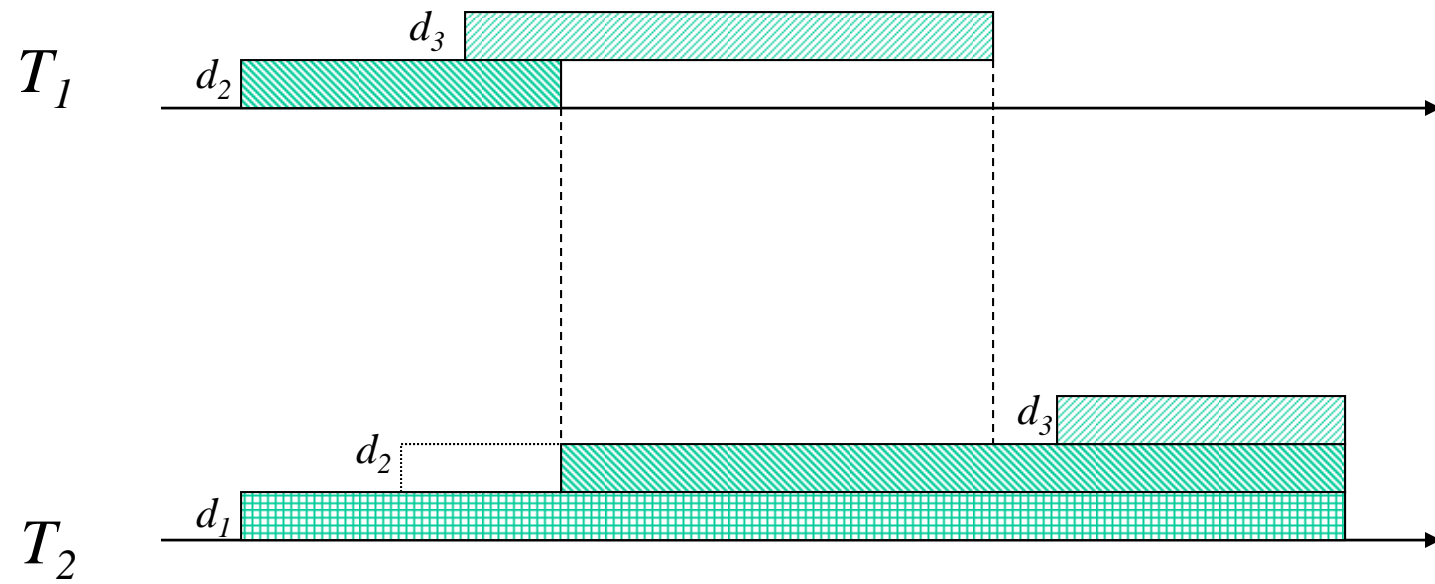
- Dynamic algorithm in which a transaction
  - requests a lock on a data object when it needs the object
  - cannot request a lock anymore after it has unlocked an object
- Thus algorithm has *growing phase* plus a *shrinking phase*
  - intermediate phase is called *lock point*
  - example:



# Two-Phase Locking: properties

- It can be shown that if a set of transactions
  - are well-formed, and
  - follow a two-phase scheme to get/release data objectsthen all legal logs are serializable
  - (legal log: a log in which a transaction trying to lock an already locked object waits on the lock)
- Two-Phase locking increases concurrency over the static scheme because objects are locked for a shorter period

# 2PL example



# Two-Phase Locking: problems

- Deadlocks due to circular wait.
  - Possible solutions:
    - kill one blocked transaction, then restore and unlock data
    - either acquire all locks or none
    - assign priorities to transactions