CSIS2300

Database I

Faculty of Business Administration

Concurrency Control

Concurrency Control Techniques

- Objectives are…
  - Understand transactions.
  - Understand the need for concurrency control.
  - Understand serial and serializable schedules.
  - Understand the need for locks and locking protocols.

Introduction

- *Transaction Processing Systems* are systems with large DBs and hundreds of concurrent users that are executing DB transactions.
- They require high availability and fast response time.
- A *transaction* represents a logical unit of DB processing that must be completed in its entirety to ensure correctness.

Introduction (cont'd)

- "Ensure correctness" means preserving the consistency of the DB.
- Therefore, we must ensure that
  - All instructions in a transaction are executed to completion, OR
  - None are performed (rolled back).

Introduction (cont'd)

- A transaction must be in one of the following states:
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Committed

Schedules

- The order of execution of operations from various transactions executing concurrently in an interleaved fashion.

Serializability of Schedules

- Schedules that are considered correct when concurrent transactions are executing.

Two Sample Transactions

T1

read_item(X);

X:=X-N;

write_item(X);

read_item(Y);

Y:=Y+N;

write_item(Y);

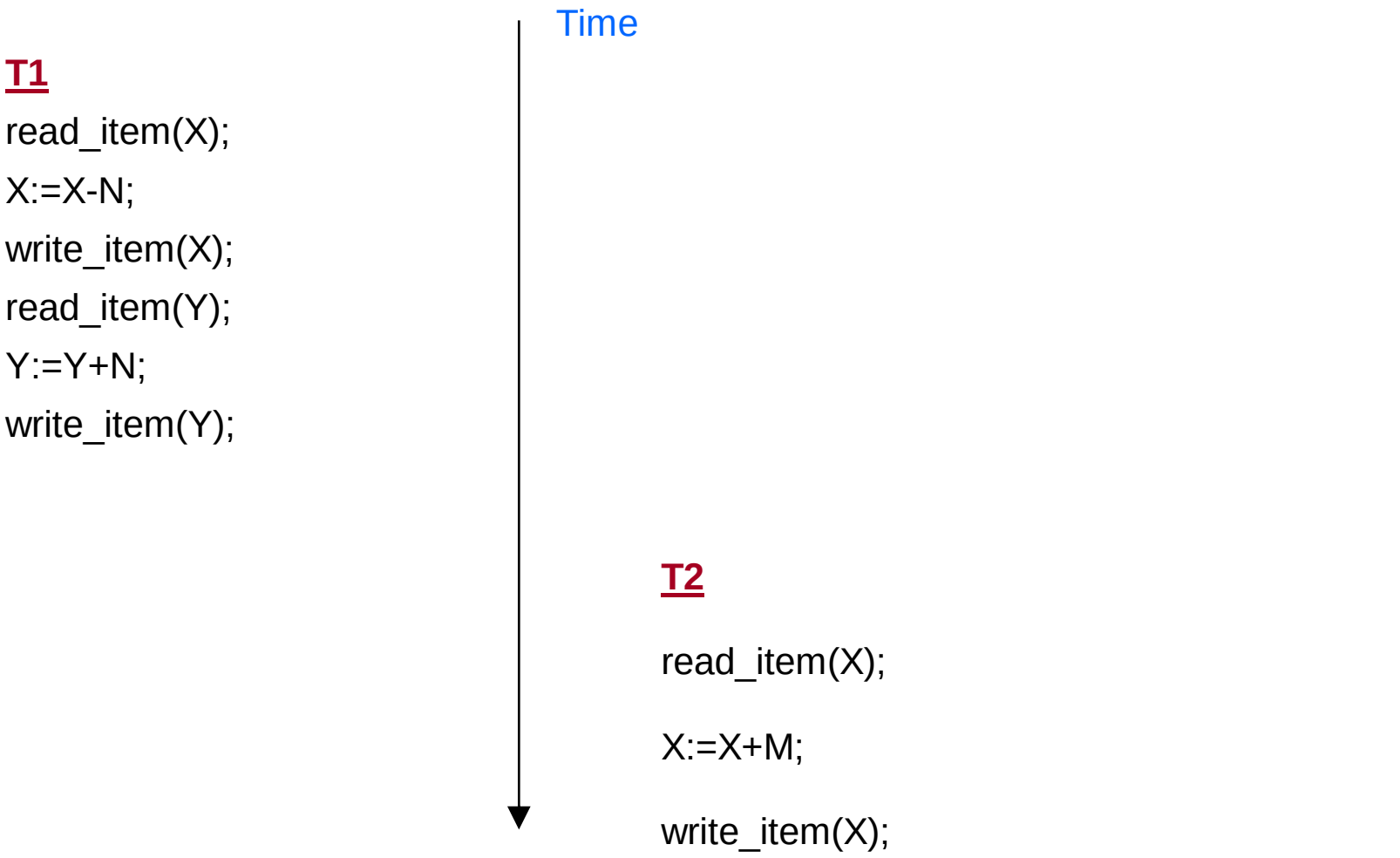
T2

read_item(X);

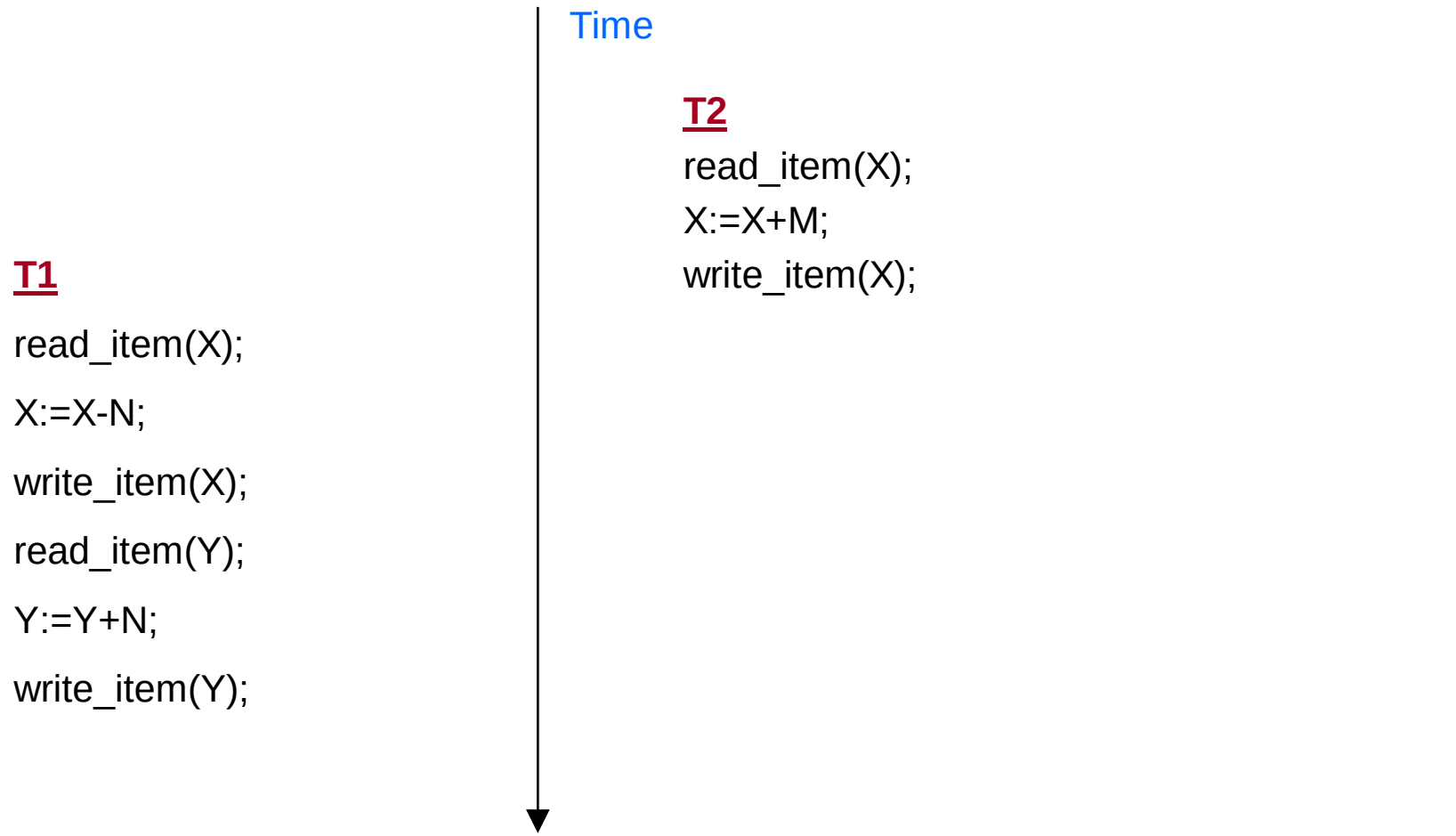
X:=X+M;

write_item(X);

No Interleaving Schedule A



No Interleaving Schedule B



Serial Schedules

- Lets try an example…
  - Suppose at the beginning X=90 and Y=90
  - And M=2 and N=3

Schedule B Example

Time

T1

read_item(X=92);

X:=(X=92)-(N=3);

write_item(X=89);

read_item(Y=90);

Y:=(Y=90)+(N=3);

write_item(Y=93);

T2

read_item(X=90);

X:=(X=90)+(M=2);

write_item(X=92);

Serial Schedules

- Schedule A and B were *serial* schedules.

Serial Schedules

- Only one transaction is active at a time.
- Commit or abort starts next transaction.
- No interleaving.
- Transactions assumed independent and therefore considered correct.
- Does not matter which transaction is first as long as…
  - Executed from beginning to end.
  - NO interference from other operations.

Serial Schedules

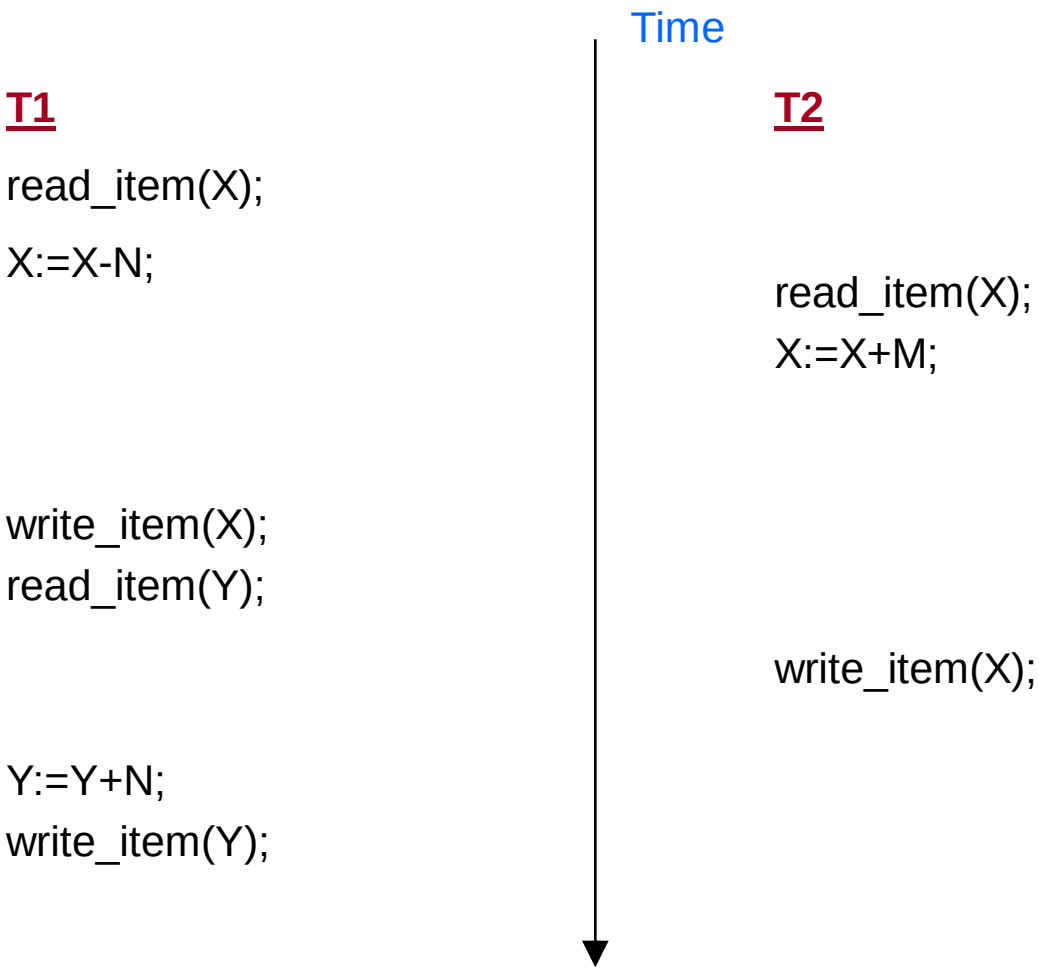
Disadvantages...

- Must wait (long time) for other transactions to end;
- Prevents CPU from time sharing while waiting (for I/O e.g.);
- Therefore generally NOT acceptable in practice.

Serial Schedules

- Schedule A and B were *serial* schedules.
- Now consider *non-serial* Schedules C and D coming up.

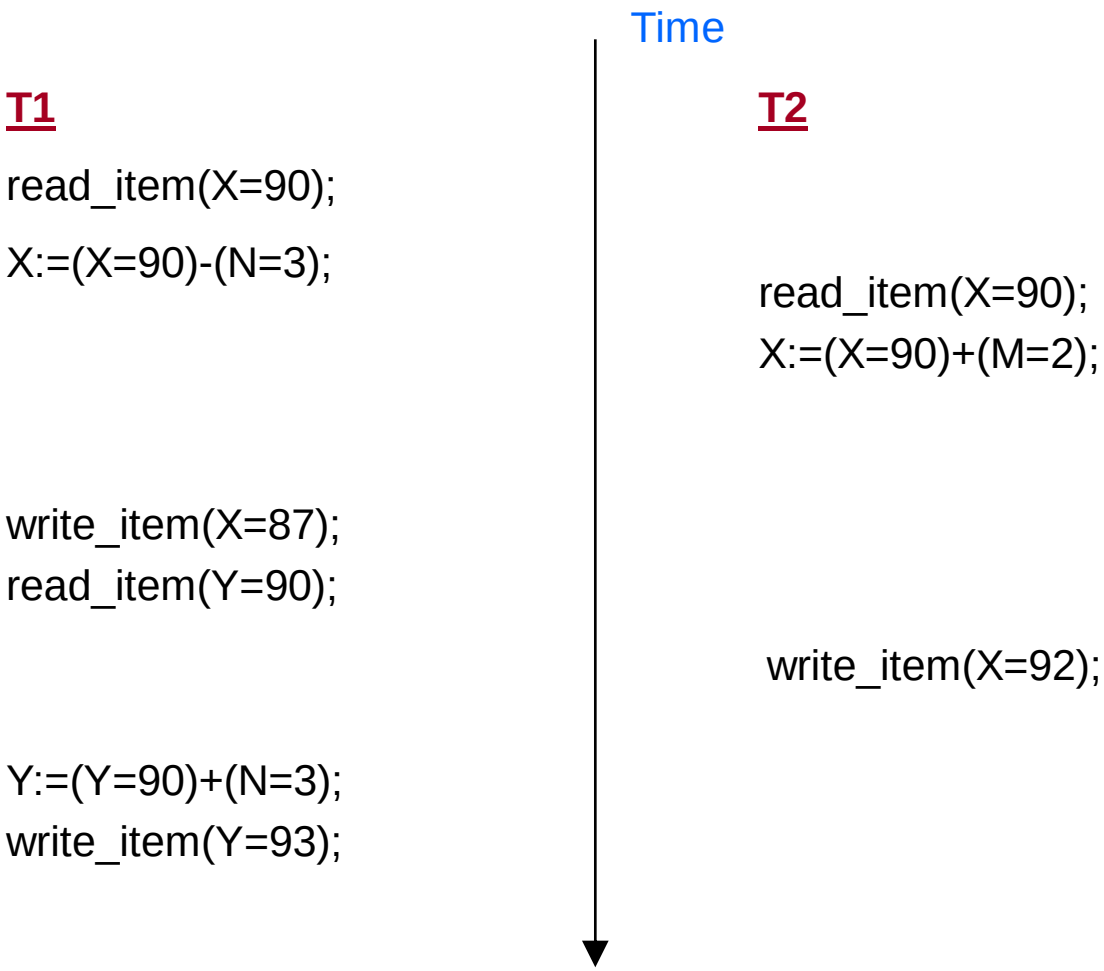
Interleaving Schedule C



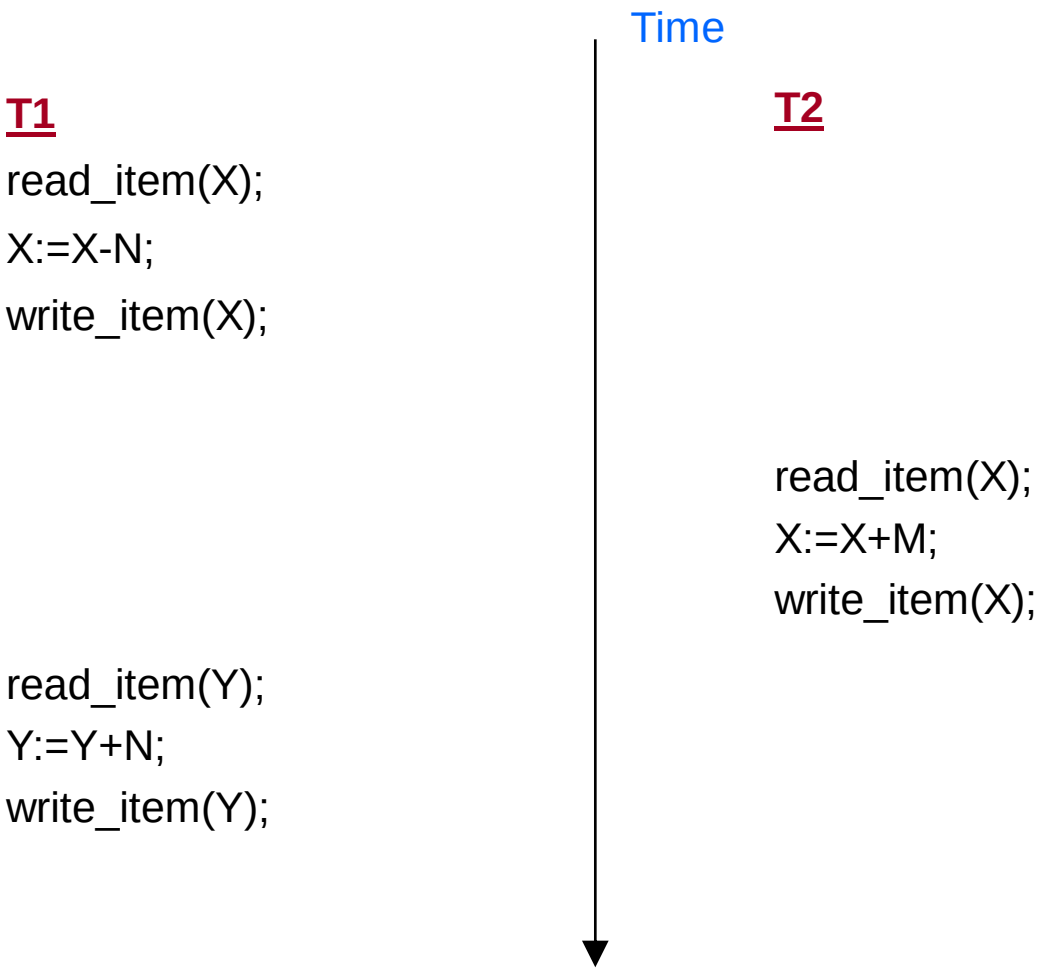
Interleaving Schedules

- Lets try an example with values…
  - Suppose at the beginning X=**90** and Y=**90**
  - And M=**2** and N=**3**

Interleaving Schedule C



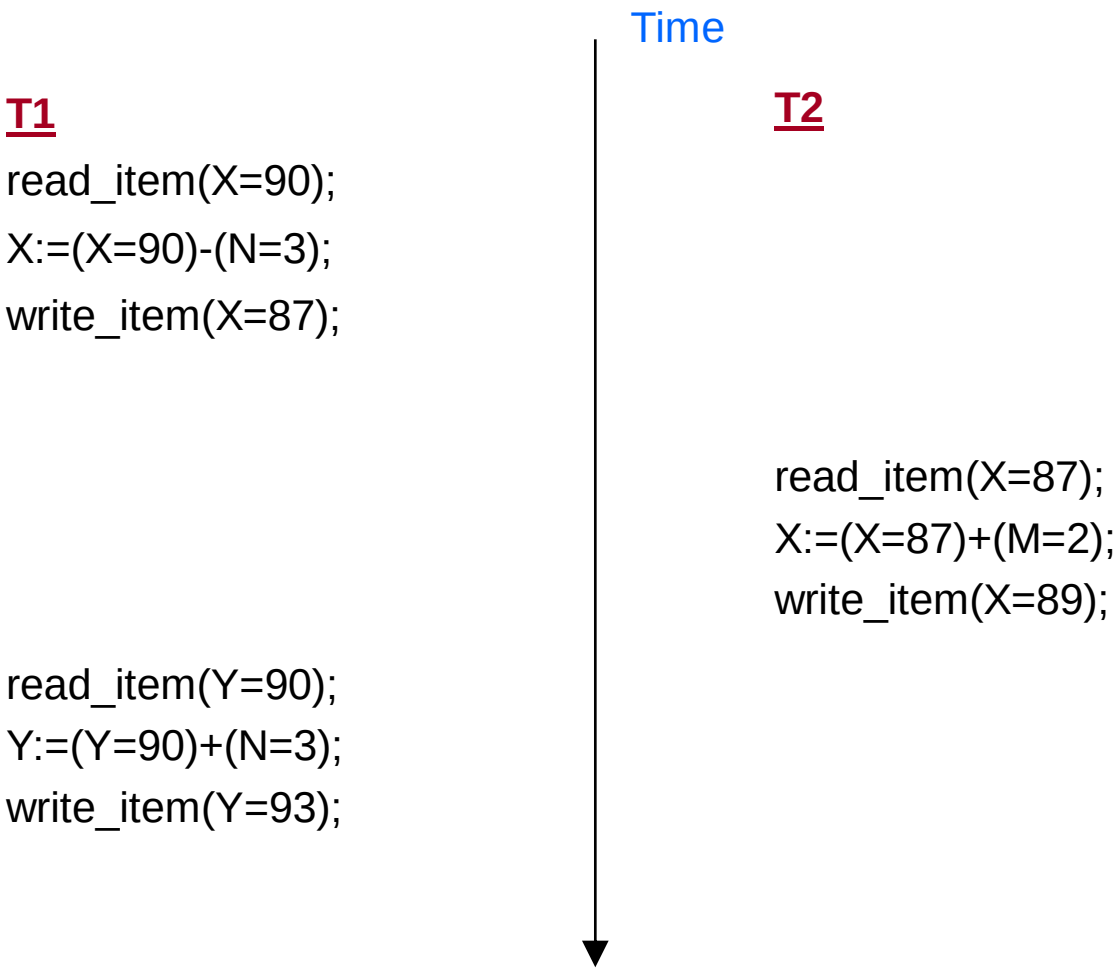
Interleaving Schedule D



Interleaving Schedules

- Lets try an example with values…
  - Suppose at the beginning X=**90** and Y=**90**
  - And M=**2** and N=**3**

Interleaving Schedule D



Serializability

- Schedule C has “lost update” problem.
- Schedule D is correct.
- How to determine which *non-serial* schedule always gives correct result ?
- Concept used is called *Serializability of Schedules*.

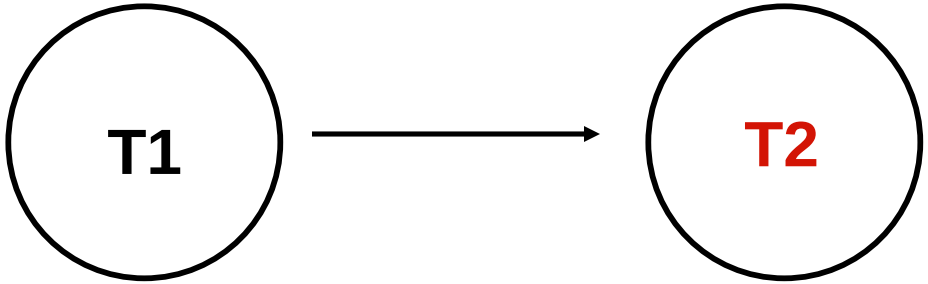
Serializability

- A schedule **S** of **n** transactions is *serializable* if it is equivalent to some serial schedule of the same **n** transactions.
- In other words if Schedule C were to produce the same results as either Schedule A or B then we would say Schedule C is *serializable*.
- *Serializable* is distinct from being *serial*.

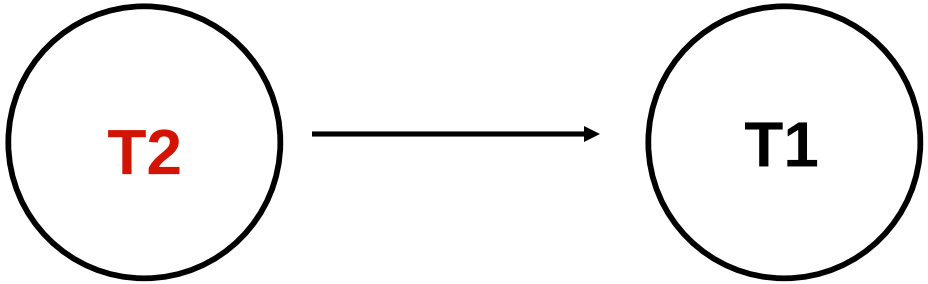
Precedence Graphs

- **Precedence graphs** are drawn to determine serializability.
  - If the PG has a cycle, then the schedule is NOT serializable
  - Otherwise, the serializability order can be obtained through topological sorting
- Test for Conflict Serializability is as explained below
- The set of edges consists of all edges **Ti → Tj** for which one of the following three conditions hold
  - Ti executes write(Q) before Tj executes read(Q)
  - Ti executes read (Q) before Tj executes write (Q)
  - Ti executes write(Q) before Tj executes write(Q)
- Next slide is an example. Schedule A - edge from T1 to T2 is shown bec all the instructions of T1 are executed before the first instruction of T2.

Precedence Graphs

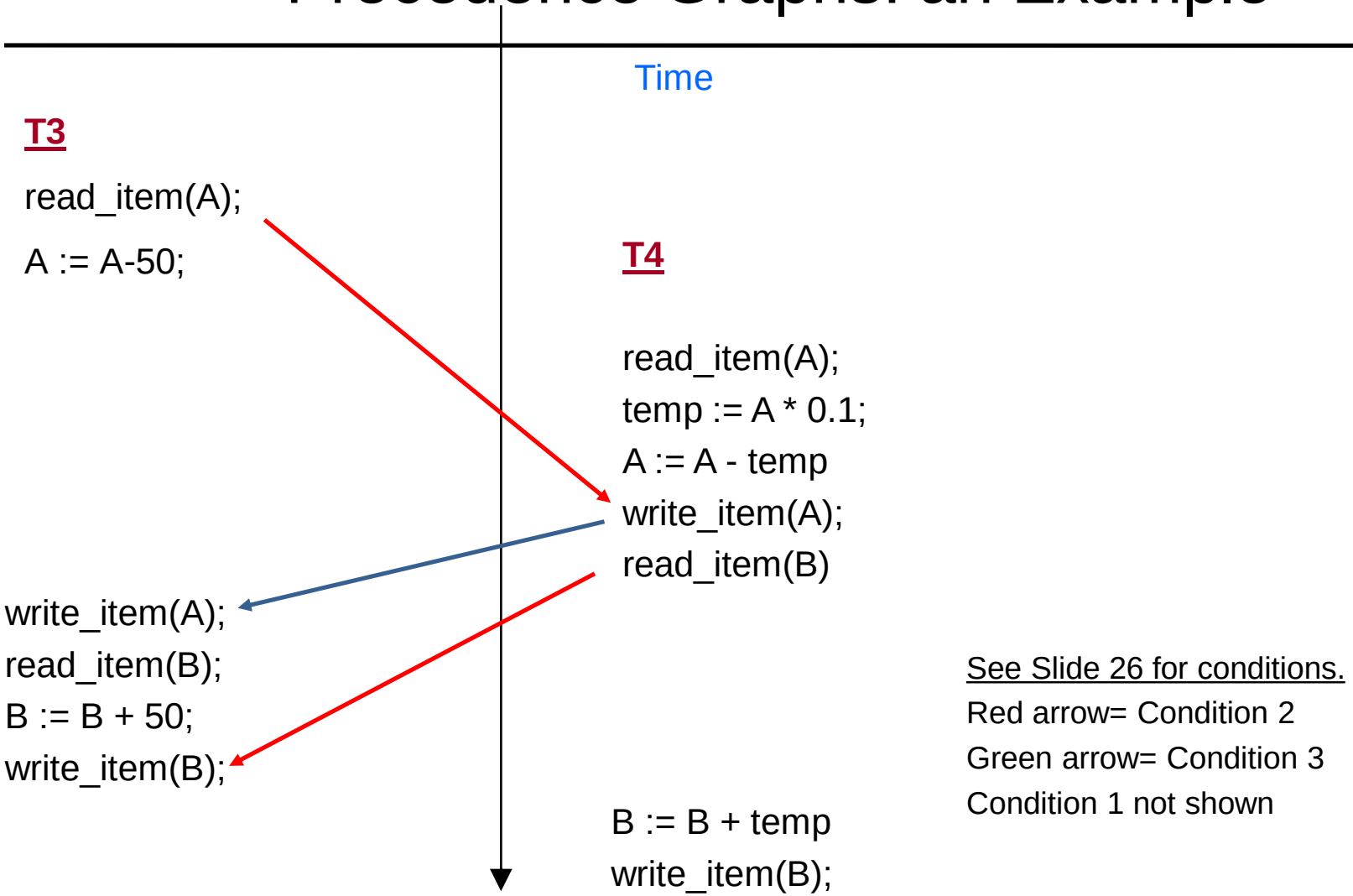


Schedule A



Schedule B

Precedence Graphs: an Example



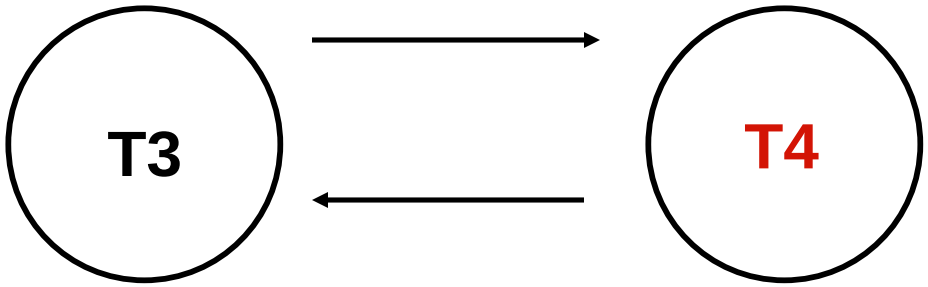
See Slide 26 for conditions.

Red arrow= Condition 2

Green arrow= Condition 3

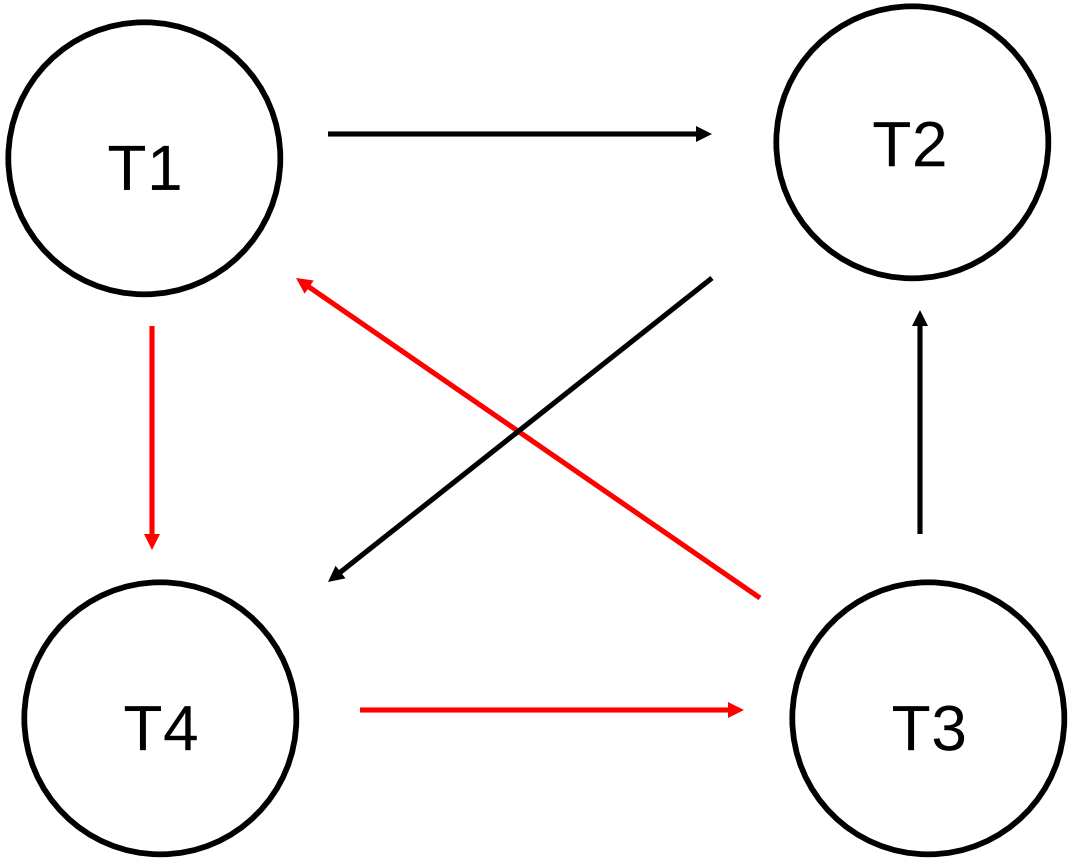
Condition 1 not shown

Precedence Graphs



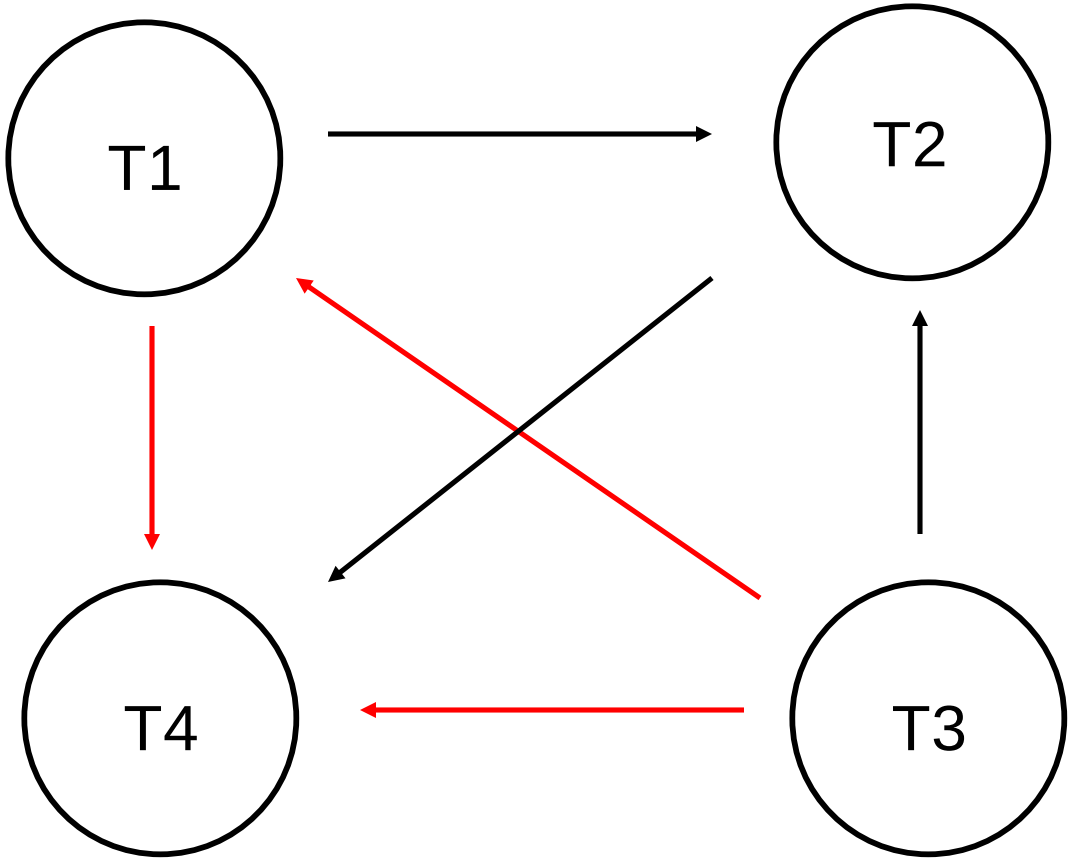
Previous schedule has a cycle so not serializable

Precedence Graphs
More examples



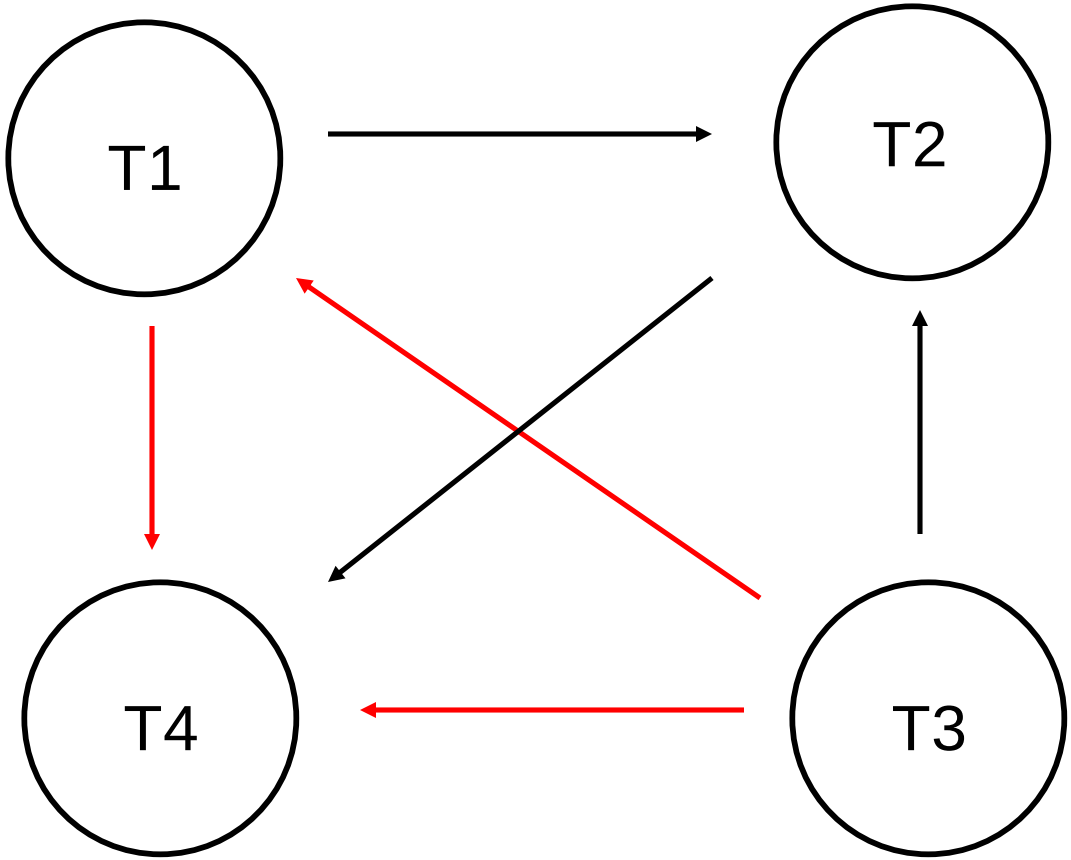
PG has a cycle so not serializable

Precedence Graphs
More examples



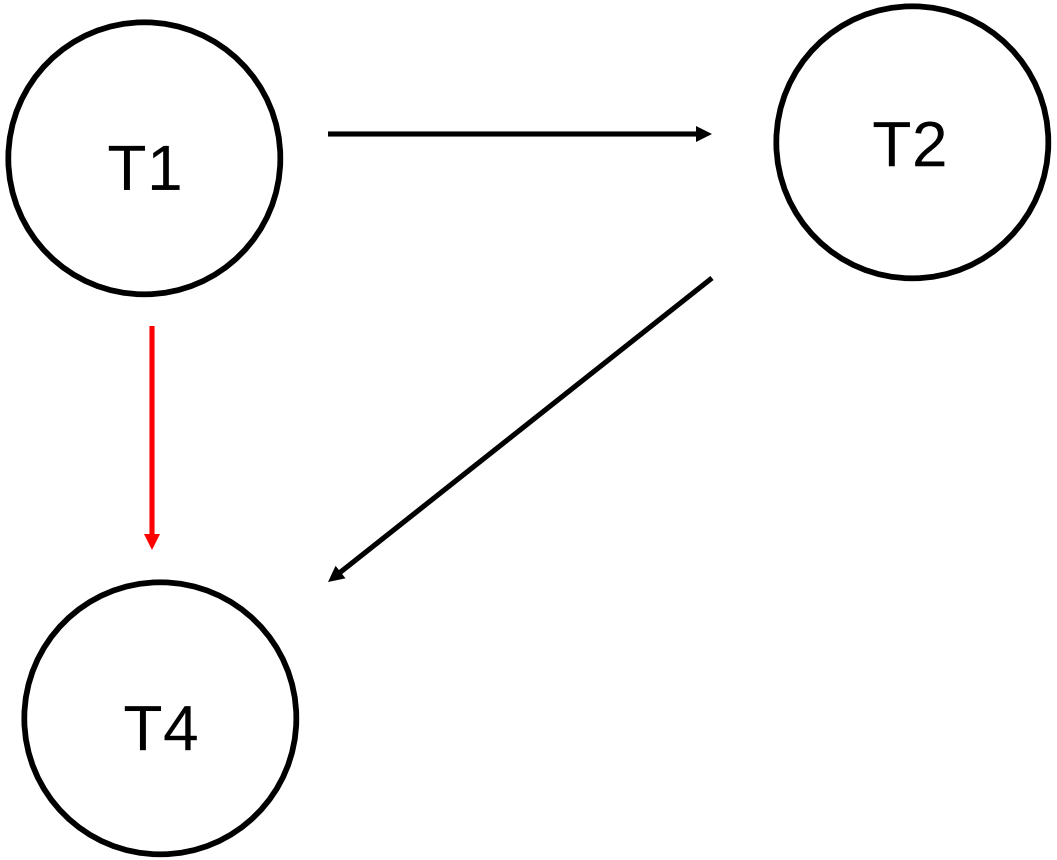
PG is serializable

Precedence Graphs
Topological Sorting



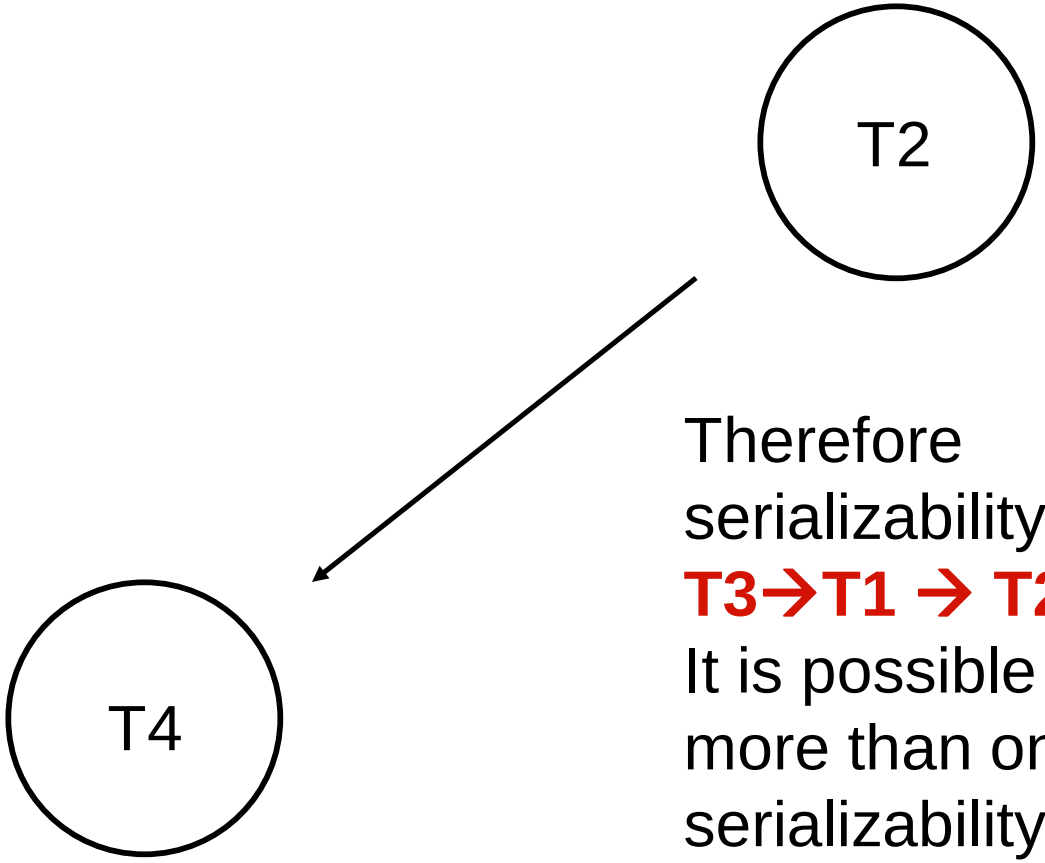
Pick a node with NO edges going into it, left with…

Precedence Graphs
Topological Sorting



Pick a node with NO edges going into it, left with…

Precedence Graphs
Topological Sorting



Therefore serializability order is : **T3→T1 → T2 →T4**.
It is possible to have more than one serializability order.

Transaction Support in SQL

- A single SQL statement is always considered atomic – either it completes execution without error or it fails.
- ANSI SQL has no explicit *Begin_Transaction* statement.
- But every transaction must have *Commit* or *Rollback*.

Concurrency Control Techniques

- Techniques used to ensure noninterference or isolation property of concurrently executing transactions.
- Most techniques ensure serializability of schedules using ***protocols***.
- ***Locking protocols*** are used in most commercial DBMSs.
- Also ***Timestamping*** protocol, et al.

Locking Protocol

- A **Lock** is a variable associated with a data item
- It describes the status of the item with respect to possible operations that can be applied to it e.g. read-only, write, lock
- Generally, one lock per data item.

Manual Locking

- In (Oracle) SELECT query rows are NOT locked
- During query changes, rows remain locked until COMMITed
- To view rows and lock them for future changes, you use,

  SELECT – FOR UPDATE

Manual Locking (cont'd)

- Syntax is:

```
SELECT columnnames
FROM tablenames
WHERE condition
FOR UPDATE OF columnnames
[NOWAIT]
```

- columnname does NOT imply that the entire column is locked, just the rows
- NOWAIT is optional and informs user that the rows requested have already been locked by another user.

Binary Locks (Auto)

- Have TWO states: Locked (1) & Unlocked (0)
- Locked data item cannot be accessed by a requesting DB operation
- No interleaving allowed once Lock or Unlock operation started until operation terminates or the transaction waits.

Binary Locks (cont'd)

- A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T
- A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.

Binary Locks (cont'd)

- A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X
- A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

Binary Locks: an Example (pseudo-code)

T1

```
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
```

T2

```
read_lock(X);
read_item(X);
unlock(X);
write_lock(Y);
read_item(Y);
Y:=X+Y;
write_item(Y);
unlock(Y);
```

Binary Locks

- Initial Values: X=**20**, Y=**30**.
- Execute T1 then T2 (serial schedule):
  - X=50
  - Y=80
- Execute T2 then T1 (serial schedule):
  - X=70
  - Y=50
- Ans: Y in T1 and X in T2 unlocked too early.

Shared/Exclusive Locks (Auto)

- Locks by themselves are insufficient as previous example shows
- Furthermore, Binary Lock too restrictive; at most one transaction can hold a lock on a given item
- S/E Lock have three states: Read(Shared) Lock, Write(Exclusive) Lock, Unlock
- But even this is insufficient.

Non-serializable Schedule

Time

T1

read_lock(Y=30);

read_item(Y);

unlock(Y);

T2

read_lock(X=20);

read_item(X);

unlock(X);

write_lock(Y=30);

read_item(Y);

Y:=(X=20)+(Y=30);

write_item(Y=50);

unlock(Y);

cont'd next page

Non-serializable Schedule

from prev page

Time

T1

write_lock(X=20);

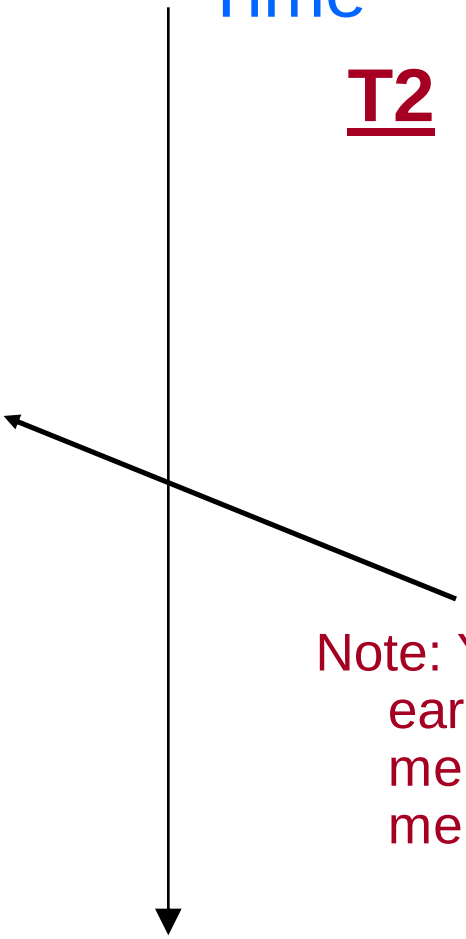
read_item(X);

X:=(X=20)+(Y=30);

write_item(X=50);

unlock(X);

T2



Note: Y was read into memory very early by T1 and remains in memory. Unlock does not free memory.

Two-Phase Locking (Auto)

- ALL locking operations (read, write) precede the first unlock operation
- Growing Phase
  - New locks acquired, none released
- Shrinking Phase
  - Acquired locks released, no new locks can be acquired.

Two-Phase Locking: Rewritten

T1

```
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
```

T2

```
read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
Y:=X+Y;
write_item(Y);
unlock(Y);
```

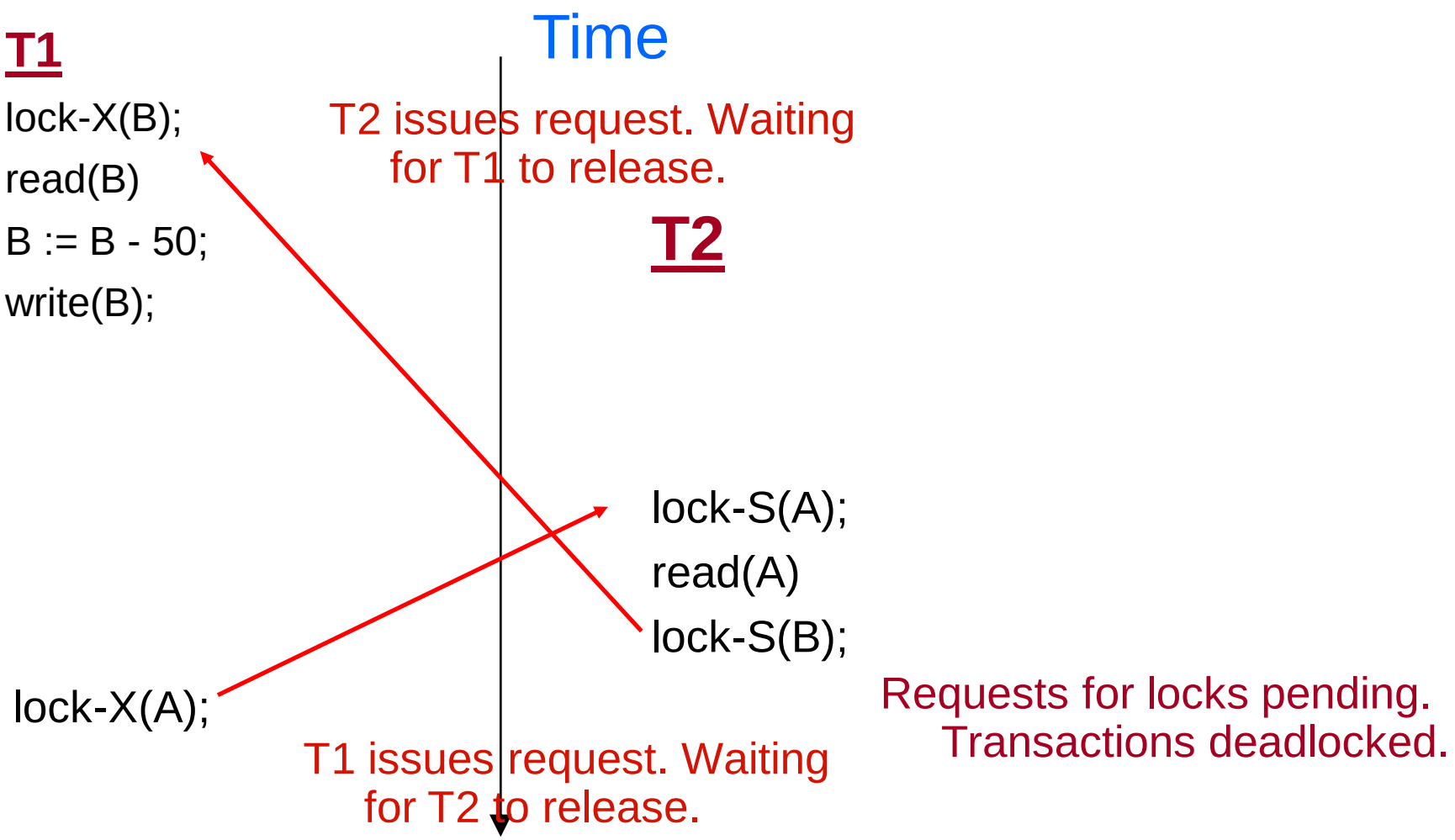
Two-Phase Locking

- Lock point : point in schedule S where transaction obtained its final lock
- Thus ordering of lock points becomes serializability ordering for the transactions
- Locking in two phases guarantees serializability
- 2 PL does NOT ensure freedom from deadlocks (e.g. previous slide).

Deadlocks

- Deadlock occurs when two (or more) transactions cannot proceed with normal execution due to locking
- Example follows.

Deadlock Example



Deadlocks

- Deadlocks are a necessary evil
- Deadlocks preferable to inconsistent state
- To avoid must follow set of rules called *locking protocol*
- When deadlock occurs must rollback one of the two transactions.

Timestamping

- In 2PL, waiting occurs for requested item if already locked
- TS do not use locks so deadlocking cannot occur
- A timestamp is a unique identifier created by the DBMS to identify a transaction
- Serializability is enforced by ordering transactions based on their timestamps.

Timestamping (cont'd)

- TS uses current date/time value of the system clock
- Equivalent serial schedule has the transactions in timestamp value order; called **timestamp ordering** (TO)
- **TO** algorithm must protect serializability order of item access by conflicting operations
- The timestamps of the transactions determine their serializability order. Thus if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj.

Timestamping (cont'd)

- To do this each DB item associated with two **timestamp** (TS) values:
  - **Read_TS(X):** The read timestamp
  - **Write_TS(X):** The write timestamp
- The timestamp of transactions determine the serializability order.
  - If TS(Ti) < TS(Tj), then the scheme ensures that the schedule is equivalent to Ti->Tj.
  - If the order is violated, then the transaction is rolled back.

Timestamping (cont'd)

- The following are kept for each data item X:
  - **Read_TS(X):** the largest timestamp (most recent) of any transaction that successfully executed read(Q)
  - **Write_TS(X):** the largest timestamp (most recent) of any transaction that successfully executed write(Q)

Timestamping (cont'd)

- The protocol operates as follows:
  - When Ti issues read(X)
    - If TS(Ti) < Write_TS(X), reject read, roll back Ti
    - If TS(Ti) >= Write_TS(X), execute read,
      - set (X) to MaxOf(read(X) or TS(Ti))
  - Example
    - Ti $^{00:02}$ < (X $_{w00:03}$), reject read, roll back Ti
    - Ti $^{00:02}$ >= (X $_{w00:01}$), execute read,

      set X $^{r00:02 \text{ or systime}}$
  - It means that an earlier transaction is trying to read a value of an item that has been updated by a (younger) later transaction
  - MaxOf() means that may be someone else who read it after you started – you must preserve that value for

Timestamping (cont'd)

- The protocol operates as follows:
  - When Ti issues read(X)
    - If TS(Ti) < Write_TS(X), reject read, roll back Ti and reject the operation. This should be done because some younger transaction with timestamp greater (more recent) than TS(Ti) – and hence after Ti in the timestamp ordering – has already written the value of item X before Ti had a chance to read X.
    - When Ti rolled back, Ti is issued new time stamp.

Timestamping (cont'd)

- The protocol operates as follows:
  - When Ti issues write(X)
    - If TS(Ti) < Read_TS(X), reject write, roll back Ti
    - If TS(Ti) < Write_TS(X), reject write, roll back Ti
    - Otherwise execute write set (X) to TS(Ti)
  - Example
    - Ti $^{00:02}$ < (X $^{r00:03}$), reject write, roll back Ti
    - Ti $^{00:02}$ < (X $_{w00:03}$), reject write, roll back Ti
    - Ti $^{00:02}$ >= (X $_{r00:001\ \ w00:01}$),
      - write,
      - set (X $_{w00:02}$).

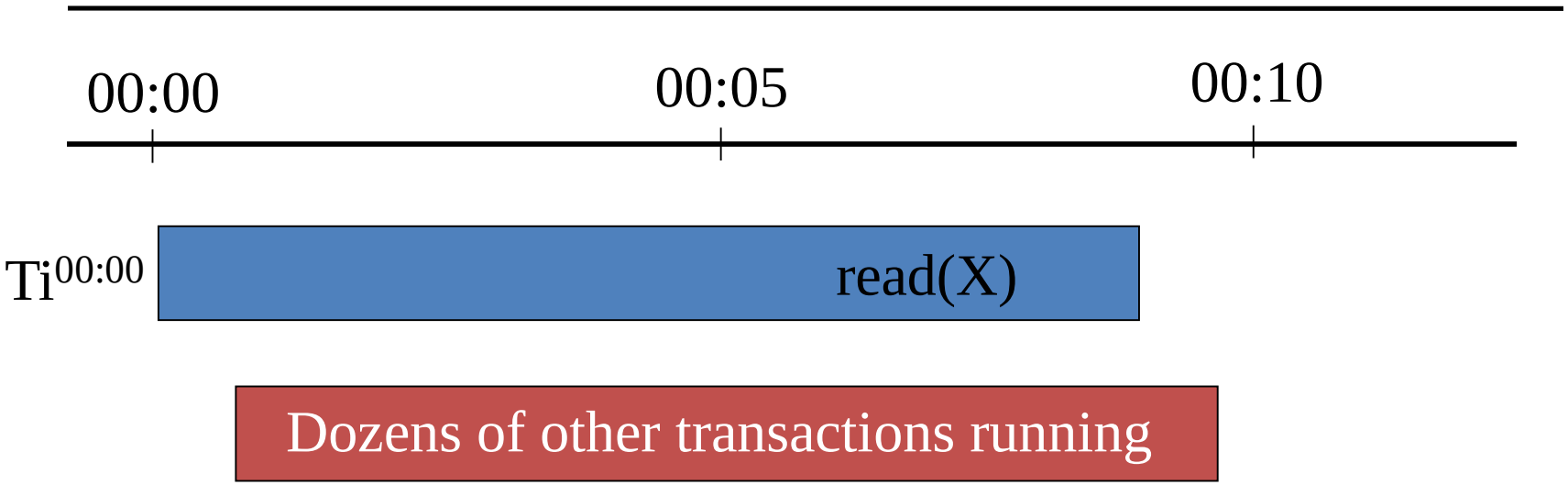
Timestamping (cont'd)

- The protocol operates as follows:
  - When Ti issues write(X)
    - If TS(Ti) < Read_TS(X), reject write, roll back Ti
    - If TS(Ti) < Write_TS(X), reject write
    - Otherwise execute write set Write_TS(X)
  - It means that some younger transaction with timestamp greater TS(Ti) – and hence after Ti in the timestamp ordering – has already read or written the value of item X before Ti had a chance to write X thus violating the time stamp ordering.

What Do You Care About?

- When You Read?
  - When was it last written to?
- When You Write?
  - When was it last written to?
    - Someone wrote to it after I started
  - When was it last read?
    - Somebody else is holding a copy of it

Why MaxOf(Read)?



- Let's say Ti successfully reads X at 00:07 and now needs to set the timestamp
- Should Ti set the timestamp to -00:00, 00:00, or 00:07 ?

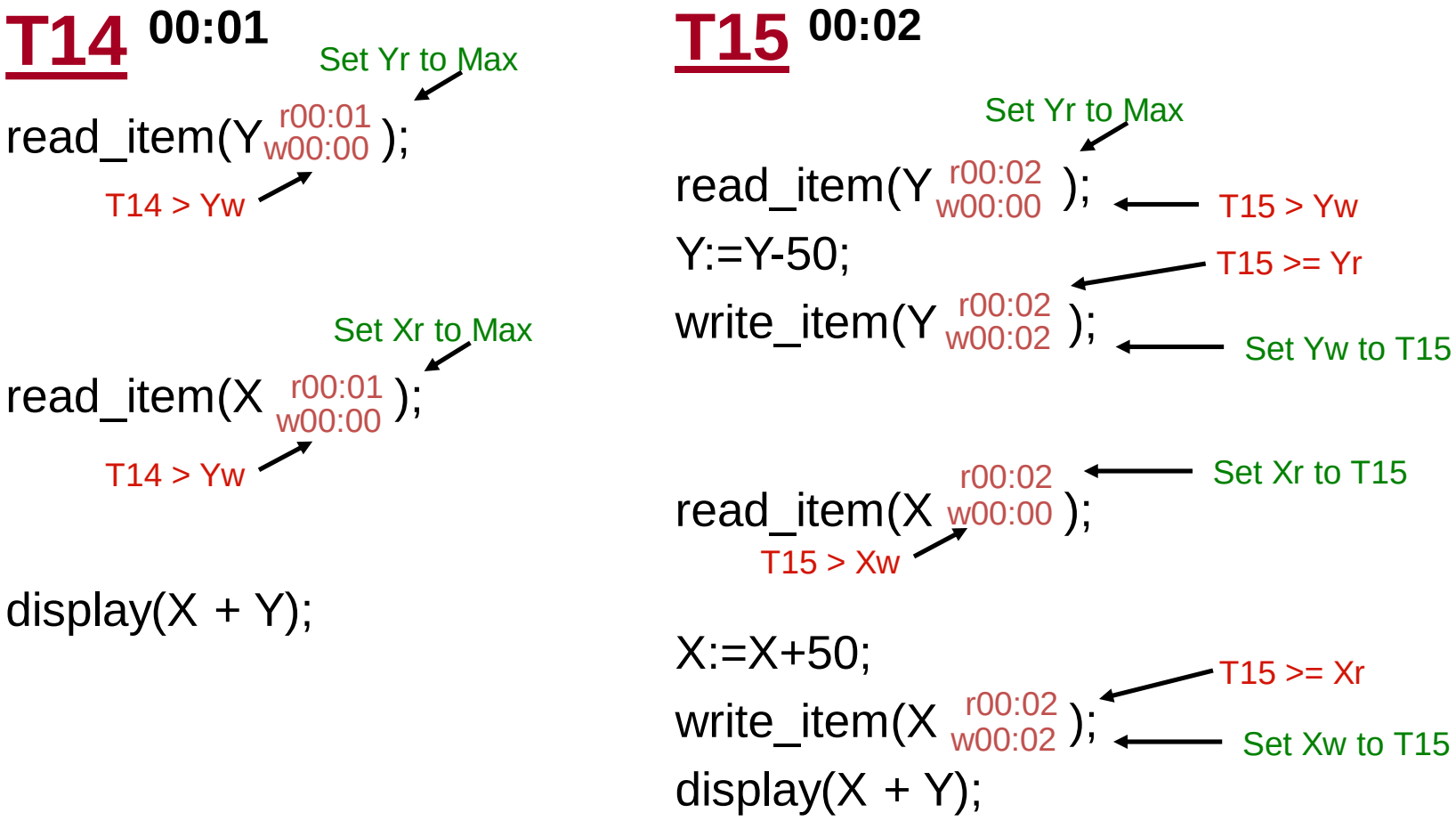
Why MaxOf(Read)? Cont'd

- If set to -00:00
  - You lied. It was read at 00:07
- If set to 00:07
  - You missed the point
  - We are trying to order transactions, not individual items (or operations)
  - Consider the cigarettes in the supermarket example
  - If two customers at the checkout want to buy the last carton of cigarettes, who should get it? The one who noticed there was only one carton left and shouts "its mine!" (real time) or the one at the head of the lineup?
- If set to 00:00 then it's correct because it is more recent than -00:00.

Timestamping

- In presenting schedules we shall assume that a transaction is assigned a timestamp immediately before its first instruction.
- “All database transactions (Read and Write) within the same transaction must have the same time stamp”. p. 476
- Guarantees that transactions are conflict serializable, and the results are equivalent to a serial schedule in which the transactions are executed in chronological order of the timestamps.

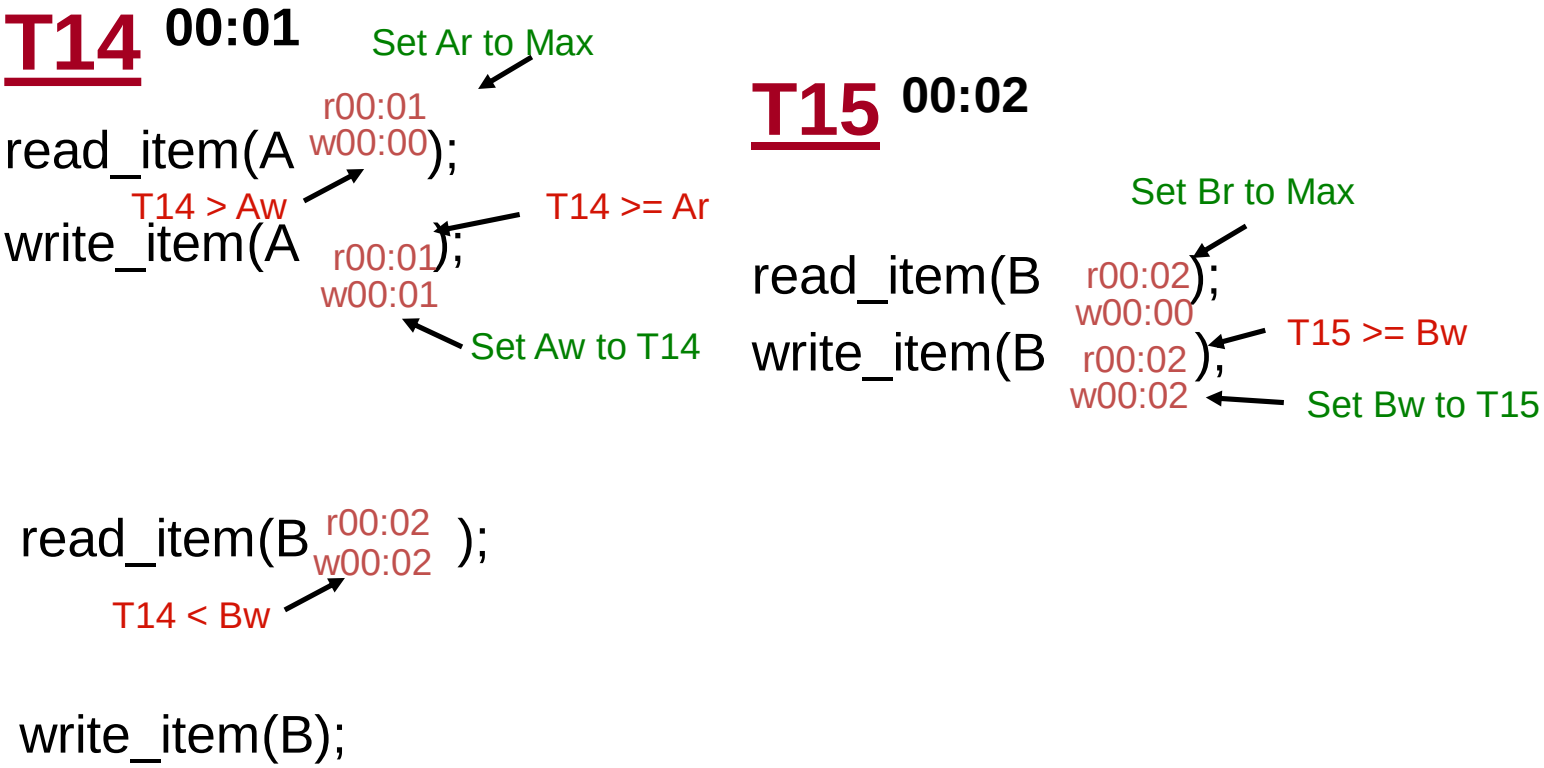
Feasible TS Schedule



Infeasible TS Schedule

- Supposing A is $20 and B is $8
  - Maybe A is savings account B is current account. OK?
- T1 transfers ALL money from A to B.
  - That is from savings to current account
  - Outcome? A has $0. B has $28. OK?
- T2 withdraws ALL money from B.
  - That is withdraw everything from current account
  - Outcome B has $0. You have $8 (should be $28)

Infeasible TS Schedule



INFeasible TS Schedule

T14 **00:01**

read_item(A $20.00);

write_item(A $20 - $20 $0);

read_item(B $0);

write_item(B A$20 + B$0);

T15 **00:02**

read_item(B $8.00);

write_item(B $0);

Time Stamping (last word)

- The time stamping protocol checks that T14 and T15 can be executed as if they were one after the other.
- A transaction that is rolled back by the system is assigned a new timestamp and is restarted.