SECTION 5

Logging and conflict serializability

February 3, 2010

Reminders

- Project 2 due tomorrow, Friday (2/4) at 11pm
- Homework 2 due next Friday (2/11) at 11pm

- Midterm Wednesday (2/9) in class

Notes on Project 2

- How do we handle concurrent transactions?
- What is Multi Version Concurrency Control (MVCC)?
- How can we test concurrent transactions?

Today

- Logging and recovery review
- Identifying conflict-serializable schedules

Why use logs to recover from crashes?

Helps satisfy 2 of the ACID constraints:

- Atomicity (all actions of txn happen or none happen)
  - How does log-based recovery keep TXen atomic?
  - How is this done in an undo log?
  - In a redo log?
- Durability (if a txn commits, its effects persist)
  - How does logging ensure that TXen persist?

Buffer Manager Policies

- Steal or No-Steal
  - Do we allow updates from uncommitted transactions to overwrite most recent committed values on disk?
    - If YES, then ‘Steal’
    - If NO, then ‘No-Steal’
- Force or No-Force
  - Do we force all updates of a transaction to disk before the transaction commits?
    - If YES, then ‘Force’
    - If NO, then ‘No-Force’

Buffer Manager Policies

- What are the performance tradeoffs of force/no-force and steal/no-steal?

	No-Steal	Steal
No-Force		Fastest
Force	Slowest	

- What logging policy is needed for each combination of force/no-force and steal/no-steal? (ex. Force + Steal)

	No-Steal	Steal
No-Force	Redo	Undo/Redo
Force		Undo

Our undo log notation

- <START T>
  - Transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T, X, v> - Update record
  - T has updated element X, and its *old* value was v

An undo logging problem

Given this undo log, when can each data item be output to disk?

- A: after 2
- B: after 3
- C: after 5, before 12
- D: after 7
- E: after 8, before 12
- F: after 10
- G: after 11

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<T2, E, e>**
9	**<START T4>**
10	**<T4, F, f>**
11	**<T3, G, g>**
12	**<COMMIT T2>**

Undo logging problem, continued

After writing these log entries, the DBMS crashes. What does it do when it restarts?

- Scan for transactions to undo: T1, T3, T4
- G, F, D, B, A reverted (in that order)
- <ABORT> written for T1, T3, T4

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<T2, E, e>**
9	**<START T4>**
10	**<T4, F, f>**
11	**<T3, G, g>**
12	**<COMMIT T2>**

What if it was a redo log?

Now, <T, X, v> means X’s *new* value is v!

… so *now* when can we output each item?

- C, E: after 12
- Others: never

(given log available)

1	<START T1>
2	<T1, A, a>
3	<T1, B, b>
4	<START T2>
5	<T2, C, c>
6	<START T3>
7	<T3, D, d>
8	<T2, E, e>
9	<START T4>
10	<T4, F, f>
11	<T3, G, g>
12	<COMMIT T2>

Redo log problem, continued

How do we recover from this redo log?

- Scan for transactions to redo: only T2
- C and E rewritten

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<T2, E, e>**
9	**<START T4>**
10	**<T4, F, f>**
11	**<T3, G, g>**
12	**<COMMIT T2>**

Why add (non-quiescent) checkpoints?

Checkpoints look different in undo and redo logs

Which is the undo log and which is the redo log?

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<COMMIT T1>**
9	**<START CKPT (T2, T3)>**
10	**<T2, E, e>**
11	**<START T4>**
12	**<T4, F, f>**
13	**<T3, G, g>**
14	**<COMMIT T3>**
15	**<END CKPT>**
16	**<COMMIT T2>**
17	**<COMMIT T4>**

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<COMMIT T1>**
9	**<START CKPT (T2, T3)>**
10	**<T2, E, e>**
11	**<START T4>**
12	**<T4, F, f>**
13	**<T3, G, g>**
14	**<COMMIT T3>**
15	**<COMMIT T2>**
16	**<END CKPT>**
17	**<COMMIT T4>**

Undo log recovery with checkpoints

The DBMS crashes with this undo log.

What do we do to recover?

- Which log entries are read?

  From end to 9: <START CKPT>
- Which transactions are undone?

  None; all have committed
- Which data do we change?

  None; no transactions to undo

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<COMMIT T1>**
9	**<START CKPT (T2, T3)>**
10	**<T2, E, e>**
11	**<START T4>**
12	**<T4, F, f>**
13	**<T3, G, g>**
14	**<COMMIT T3>**
15	**<COMMIT T2>**
16	**<END CKPT>**
17	**<COMMIT T4>**

Redo log recovery with checkpoints

This similar log is a REDO log. (why?)

How do we recover this one?

- Which log entries are read?

  From end to 9: <START CKPT>

  Then from 4: <START T2> down to end
- Which transactions are redone?

  T2, T3, T4
- Which data do we change?

  C ← c, D ← d, E ← e, F ← f, G ← g

1	**<START T1>**
2	**<T1, A, a>**
3	**<T1, B, b>**
4	**<START T2>**
5	**<T2, C, c>**
6	**<START T3>**
7	**<T3, D, d>**
8	**<COMMIT T1>**
9	**<START CKPT (T2, T3)>**
10	**<T2, E, e>**
11	**<START T4>**
12	**<T4, F, f>**
13	**<T3, G, g>**
14	**<COMMIT T3>**
15	**<END CKPT>**
16	**<COMMIT T2>**
17	**<COMMIT T4>**

Next

- Identifying conflict-serializable schedules

Schedules and conflicts

For some transaction T_1:

- $r_1(X)$ means "T_1 reads the data element X"
- $w_1(X)$ means "T_1 writes the data element X"

Two actions from T_1, T_2 *conflict* iff one or both is a write, and they act on the same element

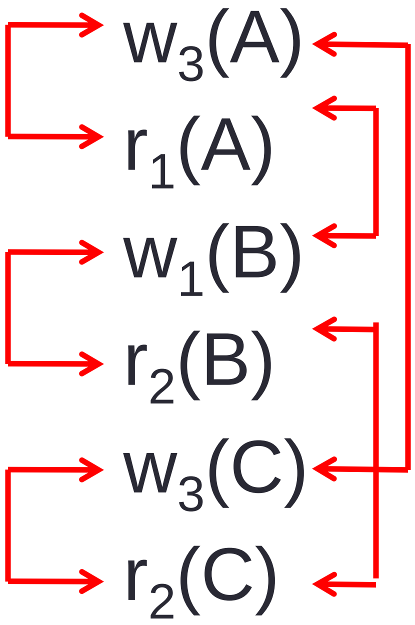
- $w_1(X)$; $r_2(X)$ or $r_2(X)$; $w_1(X)$
- $r_1(X)$; $w_2(X)$ or $w_2(X)$; $r_1(X)$
- $w_1(X)$;$w_2(X)$ or $w_2(X)$; $w_1(X)$

Executing T1 before T2 gives different results from executing T2 before T1

Two actions both from T_1 also conflict

- $r_1(X)$; $w_1(Y)$

Example 1: find all conflicts



$w_3(A)$

$r_1(A)$

$w_1(B)$

$r_2(B)$

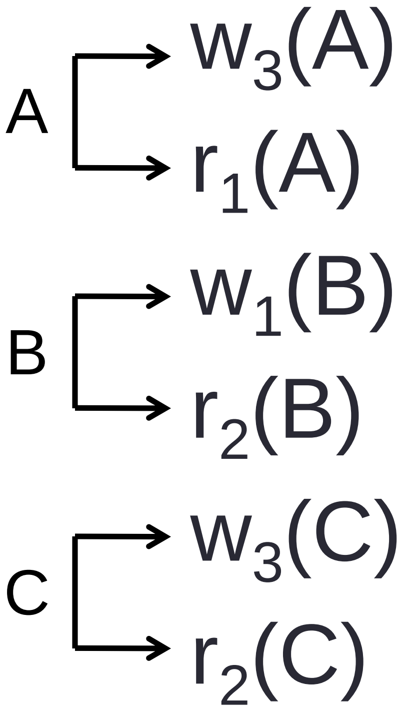
$w_3(C)$

$r_2(C)$

The precedence graph

- Recall: T_1 must *precede* T_2 iff an action from T_1 conflicts with a later action from T_2
  - Ignore conflicting actions from the same transaction
- Precedence graph shows the precedence relations

Example 1: precedence graph



A

$w_3(A)$

$r_1(A)$

B

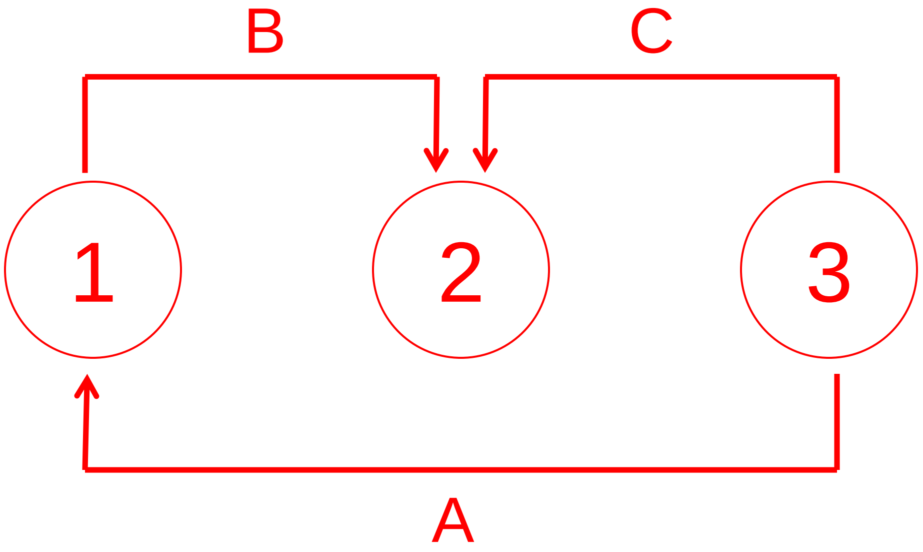
$w_1(B)$

$r_2(B)$

C

$w_3(C)$

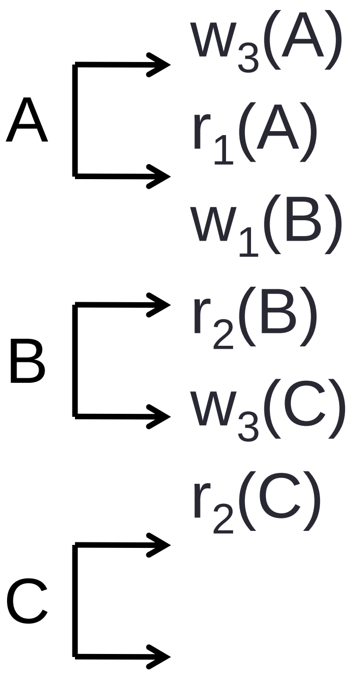
$r_2(C)$



Is it conflict serializable?

- **YES:** if no cycles in the precedence graph
  - Any transaction order which follows the precedences shown is an equivalent serial schedule
- **NO:** if there are cycles in the precedence graph

Example 1: conflict serializable?



$w_3(A)$

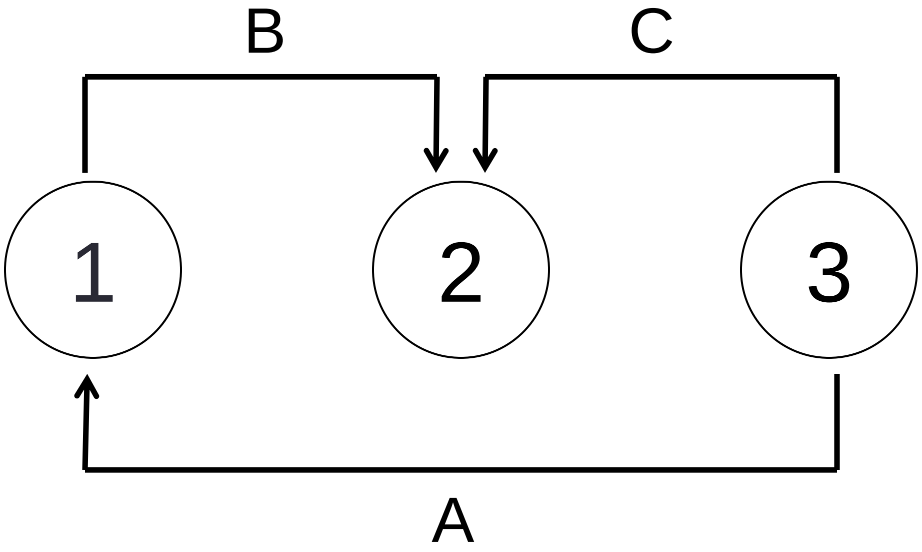
$r_1(A)$

$w_1(B)$

$r_2(B)$

$w_3(C)$

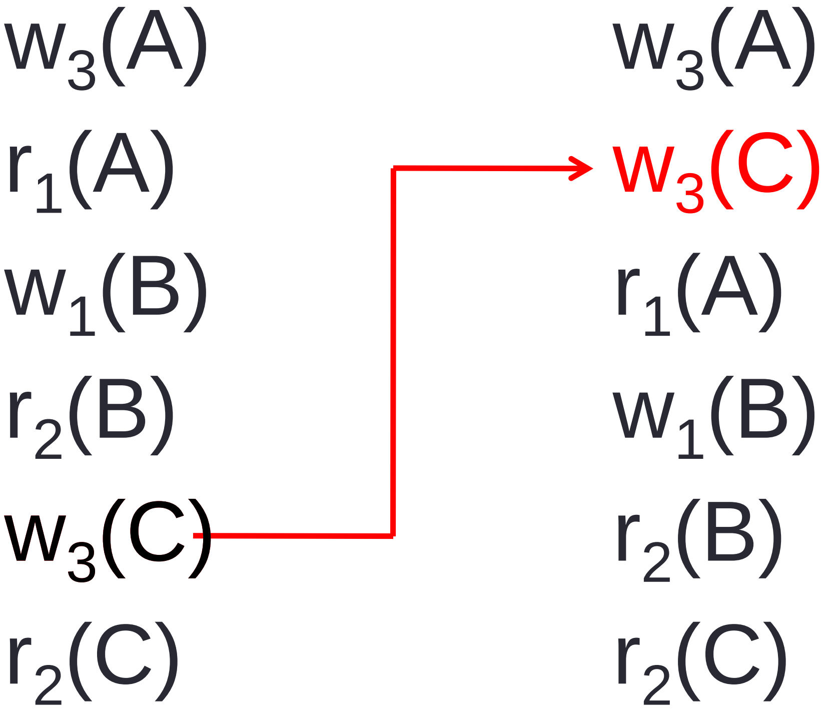
$r_2(C)$



No cycles: **YES,** conflict serializable

Only serial equivalent schedule: T_3, T_1, T_2

Example 1: serial equivalent

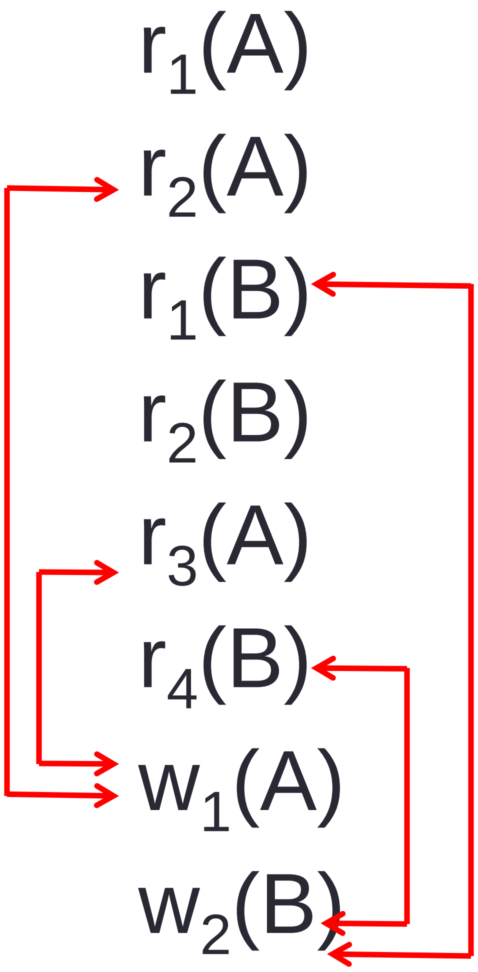


$w_3(A)$
$r_1(A)$
$w_1(B)$
$r_2(B)$
$w_3(C)$
$r_2(C)$

$w_3(A)$
$w_3(C)$
$r_1(A)$
$w_1(B)$
$r_2(B)$
$r_2(C)$

Only serial equivalent schedule: T_3, T_1, T_2

Example 2: find non-self conflicts



$r_1(A)$

$r_2(A)$

$r_1(B)$

$r_2(B)$

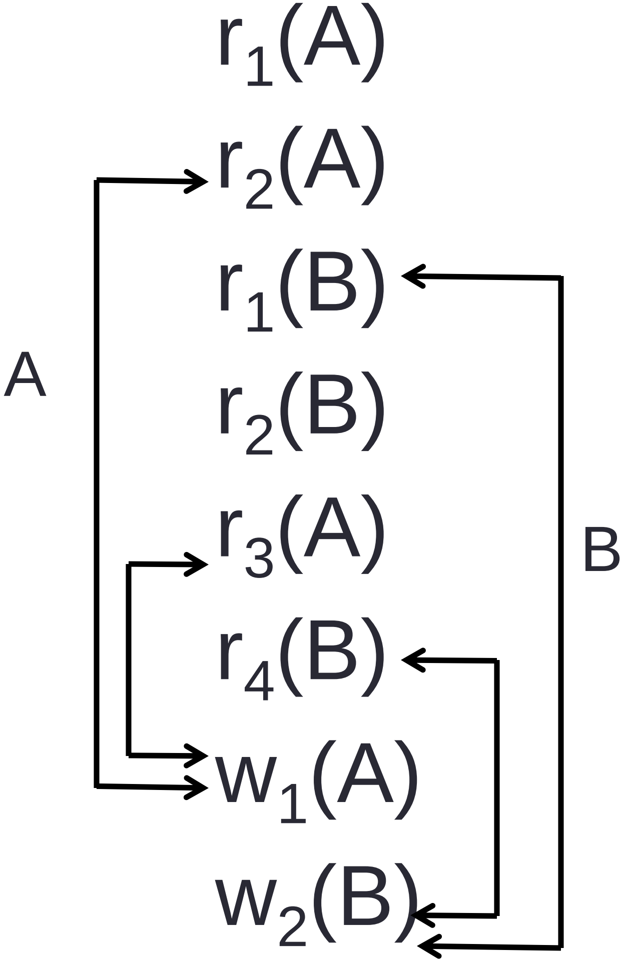
$r_3(A)$

$r_4(B)$

$w_1(A)$

$w_2(B)$

Example 2: precedence graph



$r_1(A)$

$r_2(A)$

$r_1(B)$

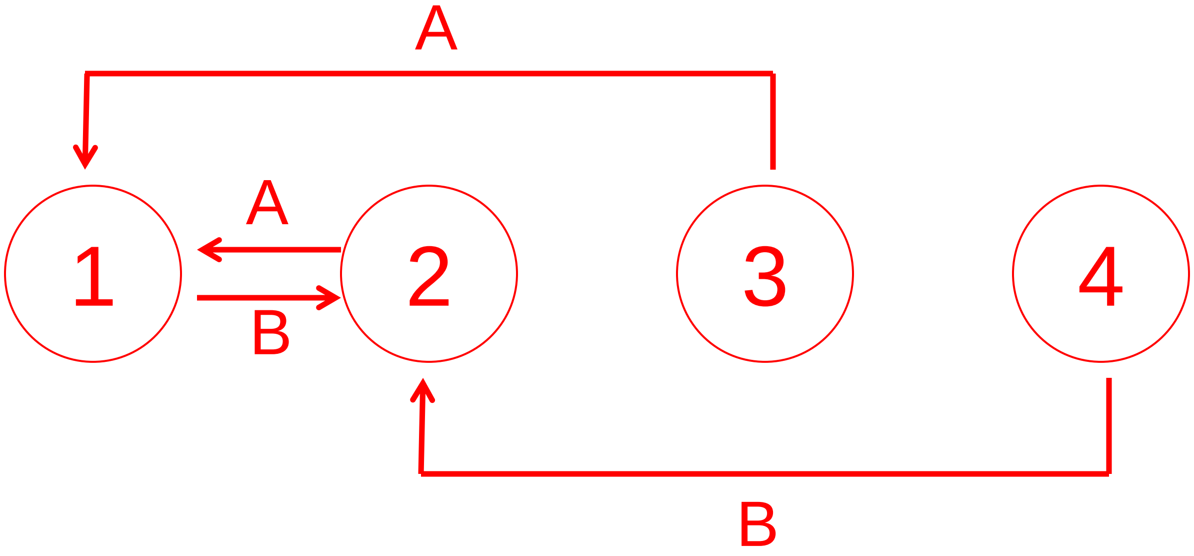
$r_2(B)$

$r_3(A)$

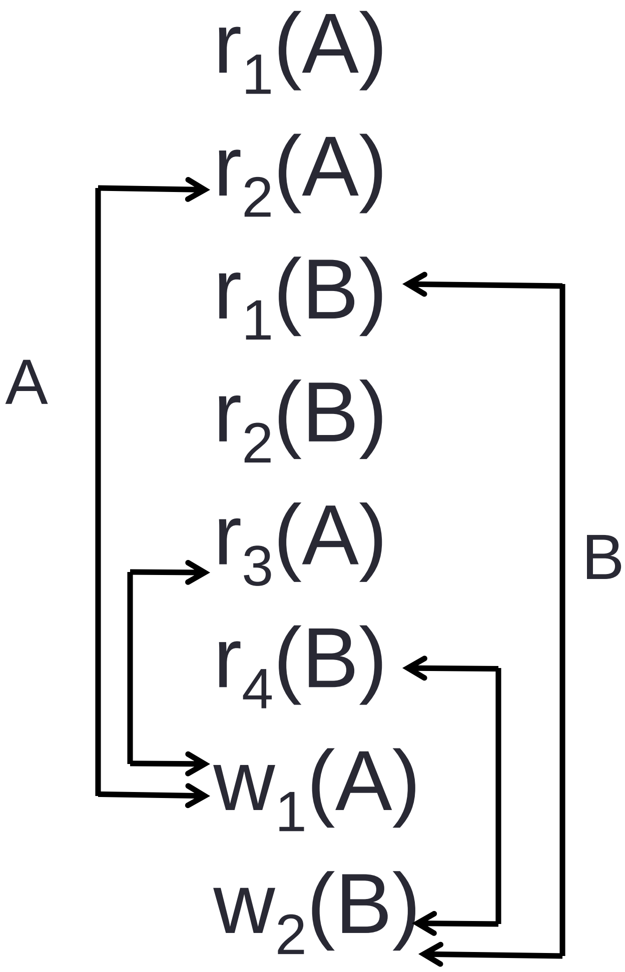
$r_4(B)$

$w_1(A)$

$w_2(B)$



Example 2: conflict serializable?



$r_1(A)$

$r_2(A)$

$r_1(B)$

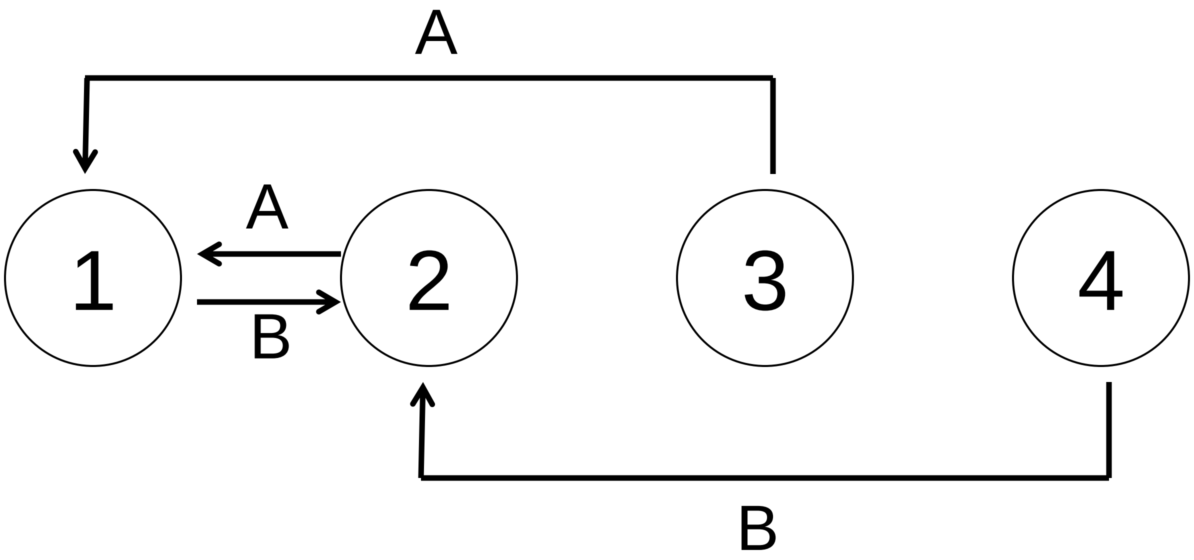
$r_2(B)$

$r_3(A)$

$r_4(B)$

$w_1(A)$

$w_2(B)$



Cycle between T_1 and T_2:

NO, not conflict serializable