
# Transaction Processing

## Schedules, Conflicts and Serializability

Dr Stewart Blakeway
Lecturer in Computer Science

# Intro

# Learning Objectives

- Desirable Properties of Transactions (ACID)
- Schedules
  - Conflicting Operations in a Schedule
  - Serializable Schedules
  - Conflict Equivalence of Two Schedules
- Testing the Serializability of a Schedule

# Desirable Properties of Transactions

- Transactions should possess several properties which are called the ACID properties

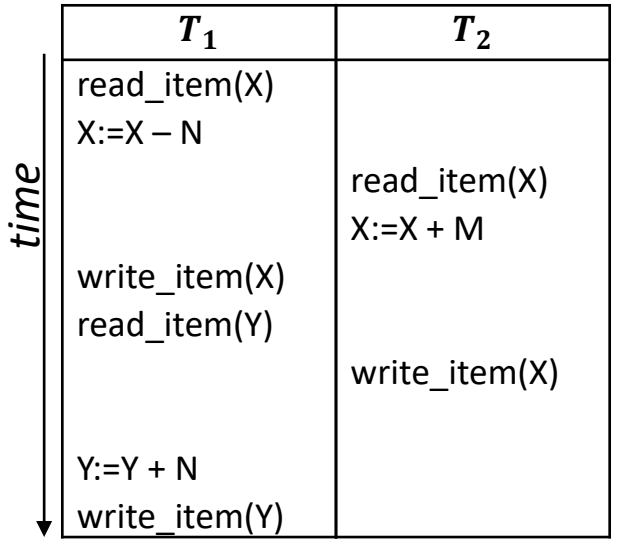
- Atomicity
- Consistency Preservation
- Isolation
- Durability or Permanency

# Schedules

- When executing transactions concurrently in an interleaved fashion we need to create a schedule
- A scheduled is defined as: $S$ of $n$ transaction $T_1, T_2, \ldots, T_n$
- A shorthand notation describing a schedule uses symbols $b, r, w, e, c$ and $a$

# Example

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

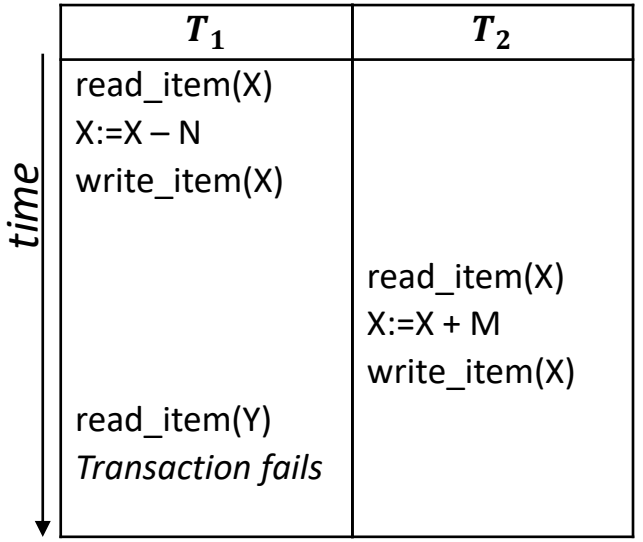


$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

# Your Turn

$S_b$:

$w, r, c$ and $a$



# Conflicting Operations in a Schedule

- Two operations in a schedule are said to conflict if they satisfy all three conditions
  - Condition 1: They belong to different transactions
  - Condition 2: They access the same item X
  - Condition 3: At least one of the operations is a write
- Conflicting Operations in a Schedule could result in inconsistent data!

# Example

1: They belong to different transactions
2: They access the same item X
3: At least one of the operations is a write

read-write conflict

$$S_a: \boldsymbol{r_1}(\boldsymbol{X}); r_2(X); w_1(X); r_1(Y); \boldsymbol{w_2}(\boldsymbol{X}); w_1(Y);$$

write-write conflict

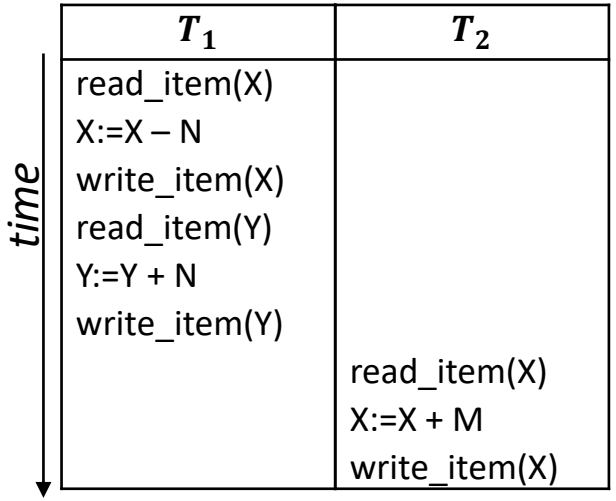
*time*

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| | read_item(X) |
| | X:=X + M |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| Y:=Y + N | |
| write_item(Y) | |

# A Complete Schedule

- The goal is to establish a complete schedule, which is defined as:
  1. The operations in $S$ are exactly those operations in $T_1, T_2, \ldots, T_n$
  2. For any pair of operations from the same transaction, their relative order in $S$ is preserved
  3. For any two conflicting operations, one of the two must occur before the other in the schedule

# The easy approach




| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |

Schedule A


| $T_1$ | $T_2$ |
|---|---|
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |

Schedule B

These are examples of Serial Schedules

One runs after the other in series.

With two transactions there can only be two possible schedules using a serial schedule.

Unacceptable in practice.

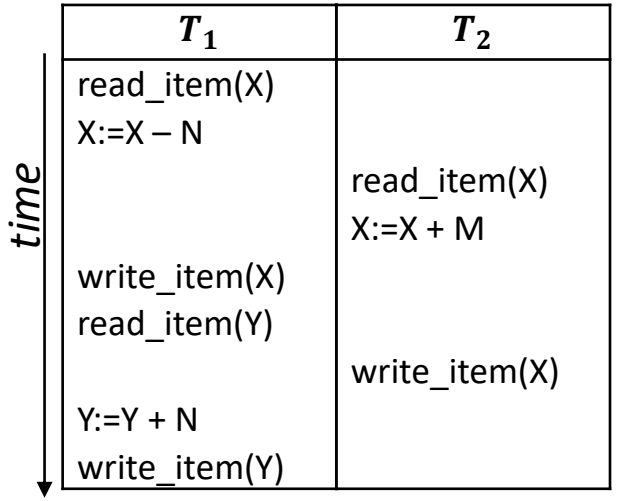
Long I/O waits (no switching)
A large transaction might hog the CPU

# Allowing Interleaving

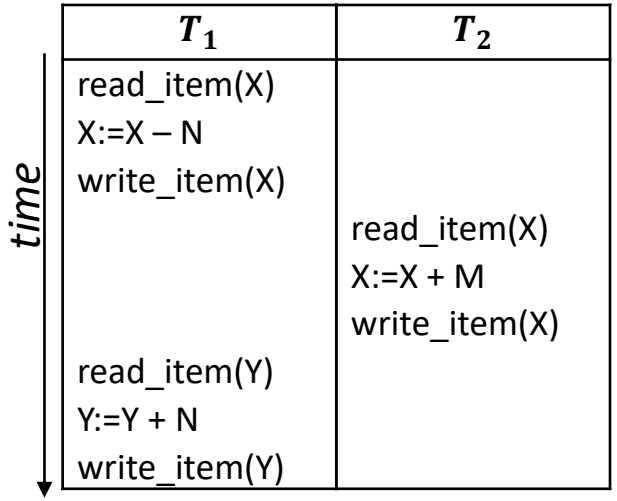
These are examples of NonSerial Schedules

They are interleaved and run concurrently.

- Ideally we want to allow interleaving where possible
- Interleaving of Transactions results in many possible schedules, here are two examples.




Schedule C




Schedule D

# Correct or not correct

Quickly help me run through the schedules with values: X=90, Y=90, N=3, M=2.

time

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |

Schedule A

time

| $T_1$ | $T_2$ |
|---|---|
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |

Schedule B

time

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| | read_item(X) |
| | X:=X + M |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| Y:=Y + N | |
| write_item(Y) | |

Schedule C

# So…

- Serial Transactions always give us the correct result
- Non Serial is a bit hit or miss
  - But we want non serial because of concurrency
- What if there was a way to determine if a non serial schedule gives us the correct result, then our problems would be solved
  - This brings us to Serializable Schedules

# Serializable Schedule

- A schedule $S$ of $n$ transactions is serializable if it is equivalent to some serial schedule of the same $n$ transactions
- There are $n!$ possible schedules of serial transactions
  - Many more for non serial schedules
- But what do we mean by equivalent?
  - **Result Equivalent**: In the simplest terms, compare the results of a serial schedule to that of a non serial schedule.
    - But… two different schedules may accidently produce the same final state

| $S_1$ |
|---|
| read_item(X) |
| X:=X + 10 |
| write_item(X) |

| $S_2$ |
|---|
| read_item(X) |
| X:=X * 1.1 |
| write_item(X) |

If X=100 they produce the same result. So are they equivalent? What if X=20?

# Safe and General Approach to Equivalence

- The safe approach is to focus only on the read and write operations of the transactions.
  - Don’t make assumptions about the other internal operations included within the transactions.
- Two schedules are equivalent if the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order
- The most common approach to equivalence of schedules is the conflict equivalence

# Conflict Equivalence of Two Schedules

- Two schedules are said to be conflict equivalent if the relative order of any two conflicting operations is the same in both schedules.
  - Different transactions
  - Access the same data item
  - Both or one is a write operation
- Not Conflicting Equivalence is when conflicting operations are applied in different orders in two schedules

# Serializable Schedules

- S is serializable if its (conflict) equivalent to some serial schedule
- Therefore, by reordering the operations in a schedule it could become serializable

Schedule C (Non Serial)

time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| | read_item(X) |
| | X:=X + M |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| Y:=Y + N | |
| write_item(Y) | |

Schedule A (Serial)

time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |

Schedule D (Non Serial)

time ↓

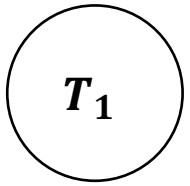
| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| X:=X – N | |
| write_item(X) | |
| | read_item(X) |
| | X:=X + M |
| | write_item(X) |
| read_item(Y) | |
| Y:=Y + N | |
| write_item(Y) | |

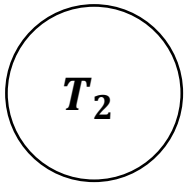
# Testing the Serializability of a Schedule

- We can use graph theory to easily test if a schedule is serializable or not
- Recall that we only focus on the read and write operations

Schedule C (Non Serial)

| $T_1$ | $T_2$ |
|---|---|
| | read_item(X) |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| write_item(Y) | |



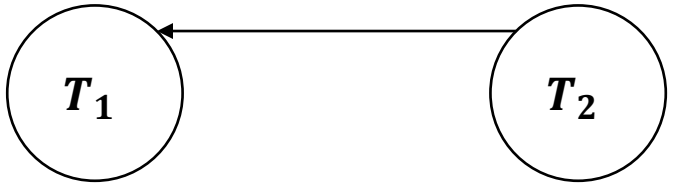




These two transactions conflict, we have a read operation and a write operation

# Testing the Serializability of a Schedule

Schedule C (Non Serial)

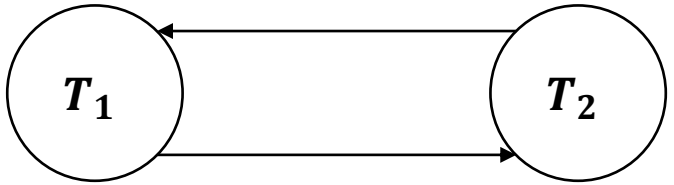
| $T_1$ | $T_2$ |
|---|---|
| | read_item(X) |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| write_item(Y) | |



# Testing the Serializability of a Schedule

Schedule C (Non Serial)

| $T_1$ | $T_2$ |
|---|---|
| | read_item(X) |
| write_item(X) | |
| read_item(Y) | |
| | write_item(X) |
| write_item(Y) | |




These two transactions conflict, we have a write operation and a write operation

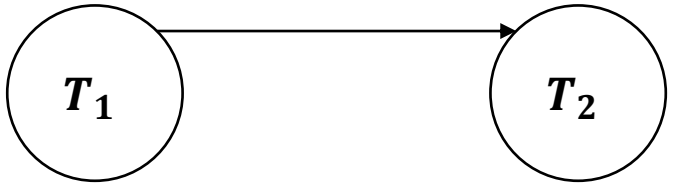
Because we have cycle, the schedule is not serializable

# Testing the Serializability of a Schedule

Schedule D (Non Serial)

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| write_item(X) | |
| | read_item(X) |
| | write_item(X) |
| read_item(Y) | |
| write_item(Y) | |

*time* ↓



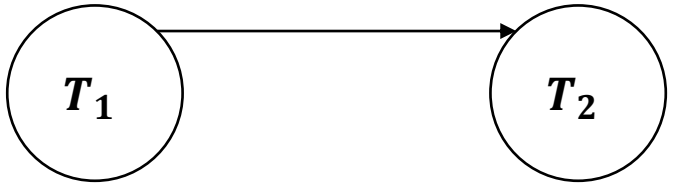

We can now skip $T_1$ because we have already found a conflict

# Testing the Serializability of a Schedule

Schedule D (Non Serial)

| $T_1$ | $T_2$ |
|---|---|
| read_item(X) | |
| write_item(X) | |
| | read_item(X) |
| | write_item(X) |
| read_item(Y) | |
| write_item(Y) | |

*time* ↓




No conflict from $T_2$ to $T_1$ so this schedule is serializable

# Let look at a more complicated example

Schedule

time ↓

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item(B) | | |
| | | read_item(C) |
| read_item(A) | | |
| | write_item(A) | |
| write_item(A) | | |
| | write_item(B) | |
| | | write_item(A) |
| write_item(B) | | |
| | | write_item(B) |
| | | write_item(C) |

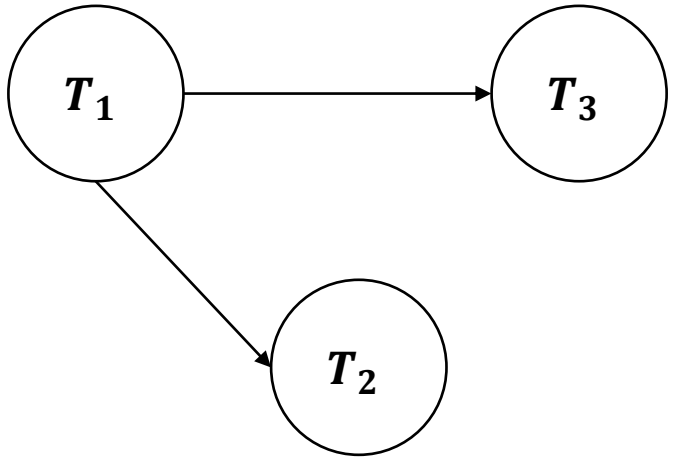
Using the approach we have seen. Determine if this schedule is serializable or not

# Let look at a more complicated example

Schedule

time

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item(B) | | |
| | | read_item(C) |
| read_item(A) | | |
| | write_item(A) | |
| write_item(A) | | |
| | write_item(B) | |
| | | write_item(A) |
| write_item(B) | | |
| | | write_item(B) |
| | | write_item(C) |

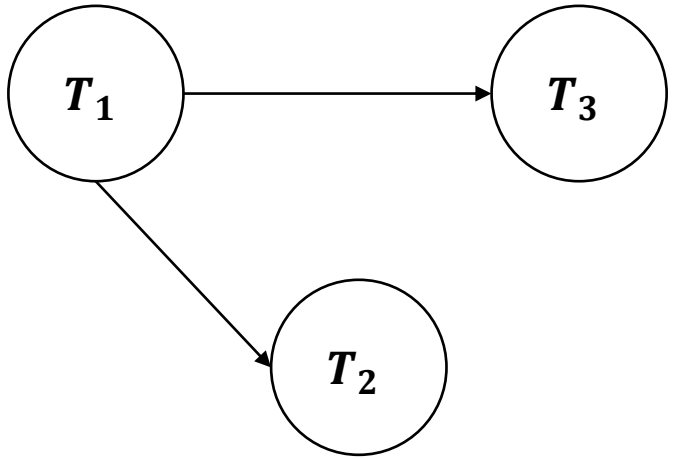


# Let look at a more complicated example

Schedule

time ↓

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item(B) | | |
| | | read_item(C) |
| read_item(A) | | |
| | write_item(A) | |
| write_item(A) | | |
| | write_item(B) | |
| | | write_item(A) |
| write_item(B) | | |
| | | write_item(B) |
| | | write_item(C) |



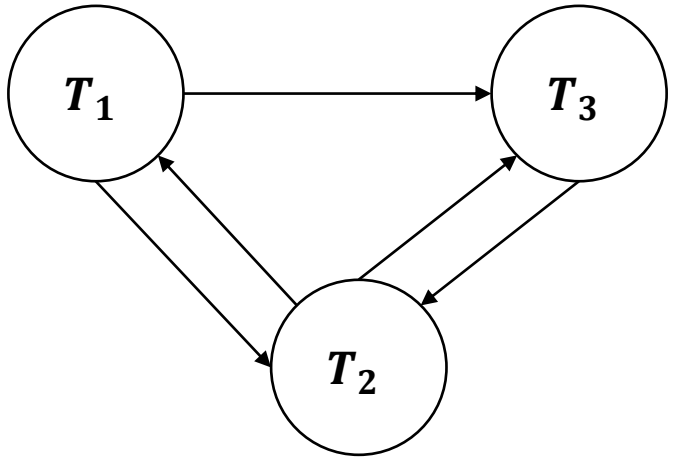

All good, no conflicts

# Let look at a more complicated example

Schedule

time ↓

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item(B) | | |
| | | read_item(C) |
| read_item(A) | | |
| | write_item(A) | |
| write_item(A) | | |
| | write_item(B) | |
| | | write_item(A) |
| write_item(B) | | |
| | | write_item(B) |
| | | write_item(C) |




Cycle create between $T_2$ and $T_1$, $T_2$ and $T_3$

# Conclusion

- Desirable Properties of Transactions (ACID)
- Schedules
  - Conflicting Operations in a Schedule
  - Serializable Schedules
  - Conflict Equivalence of Two Schedules
- Testing the Serializability of a Schedule