UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 3380 Spring 2024

Database Systems

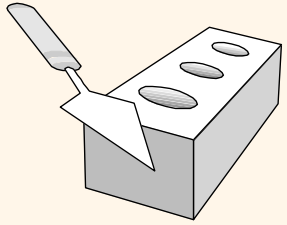
M & W 4:00 to 5:30 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON (must have)

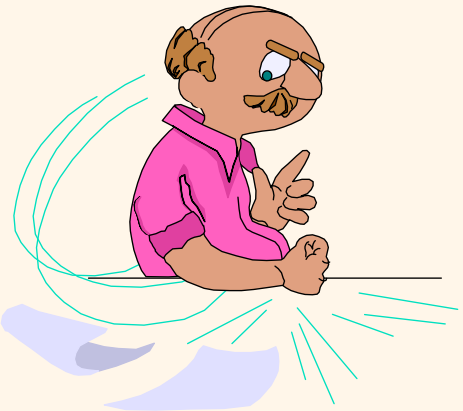
NO CHATTING during LECTURE

VH, UNhide Section 18



COSC 3380

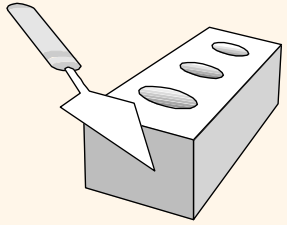
4 to 5:30



PLEASE
LOG IN
CANVAS

Please close all other windows.

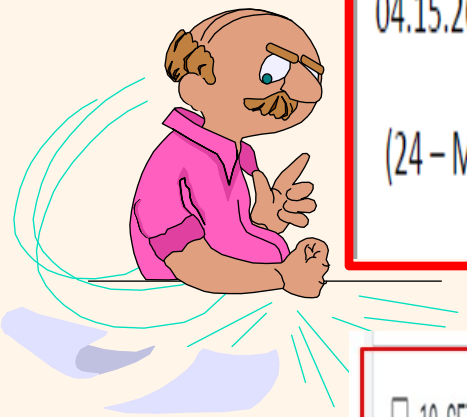
COSC 3380



04.15.2024 (24 – Mo)	**ZyBook SET 4 - 2**	**Set 4** LECTURE 17 CONCURRENCY CONTROL
04.17.2024 (25 – We)	**ZyBook SET 4 - 3**	**Set 4** LECTURE 18 CRASH RECOVERY LECTURE 19 SECURITY and AUTHORIZATION
04.22.2024 (26 – Mo)		**EXAM 4 Practice** **(PART of 20 points)**
04.24.2024 (27 – We)	**TA Download** **ZyBook SET 4 Sections** **(4 PM)** **(PART of 30 points)**	**EXAM 4 Review** **(PART of 20 points)**
04.29.2024 (28 –Mo)		**EXAM 4** **(PART of 50 points)**

COSC 3380

Class 24



04.15.2024	ZyBook SET 4 - 2	Set 4
(24 – Mo)		LECTURE 17 CONCURRENCY CONTROL

19. SET 4 - 2:CONCURRENCY CONTROL Hidden 0% v LECTURE 17 CONCURRENCY CONTROL

From 4:00 to 4:07 PM – 5 minutes.

04.15.2024 (24 – Mo)	ZyBook SET 4 - 2	Set 4 LECTURE 17 CONCURRENCY CONTROL

CLASS PARTICIPATION 20 points — 20% of Total

CONCURRENCY

Class 24 BEGIN PARTICIPATION
Not available until Apr 15 at 4:00pm | Due Apr 15 at 4:07pm

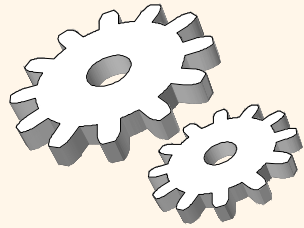
VH, publish

This is a synchronous online class.

Attendance is required.

Recording or distribution of class materials is prohibited.

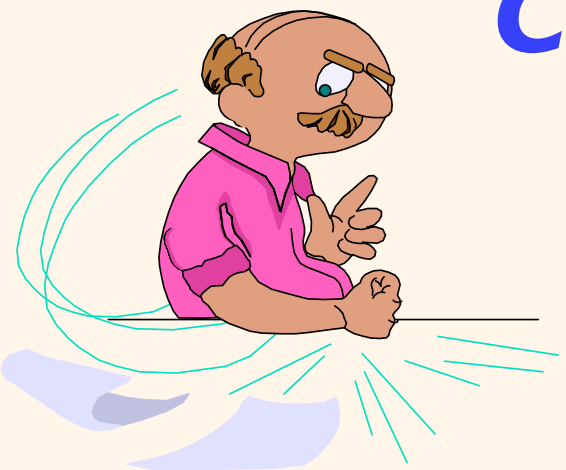
1. At the beginning of selected classes there is an **assessment in the first 10 minutes**. (**beige** BOX in the Detailed Syllabus)
2. At the end of selected classes there is an **assessment in the last 10 minutes**. (**blue** BOX in the Detailed Syllabus)
3. ZyBook sections will be downloaded and used for 30% of Total Score on the dates specified in the Detailed Syllabus.
4. EXAMS are in CANVAS. **No late EXAMS**.
5. I have to be **present in TEAMS in order to take any graded assignment assigned during that class.**



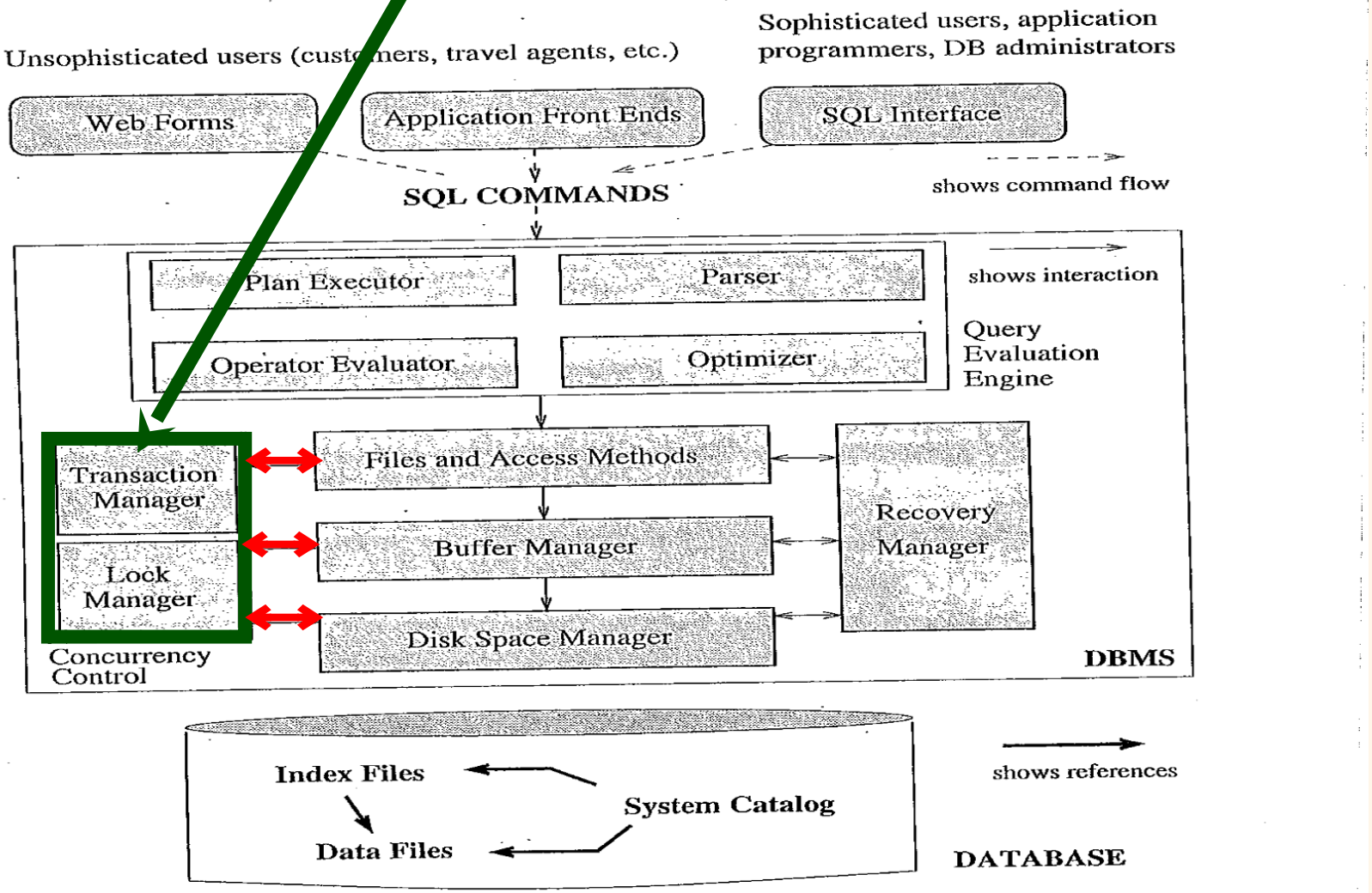
COSC 3380

Lecture 17

Concurrency Control



A DBMS – Concurrency Control



A Sample Transaction

Initial state <Consistent state>

```
1:  Begin_Transaction
2:      get (K1, K2, CHF) from terminal
3:      Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4:       S1 := S1 - CHF;
5:      Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6:      Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;
7:      S2 := S2 + CHF;
8:      Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9:      Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
                Values (K1, today, -CHF, 'Transfer');
10:     Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
                Values (K2, today, CHF, 'Transfer');
12:     If S1<0 Then Abort_Transaction
11: End_Transaction
```

Final state <Consistent state>

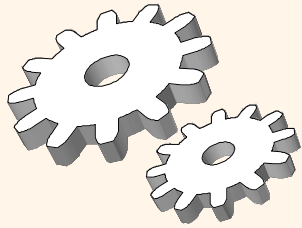
Transaction = Program that takes database from one consistent state to another consistent state

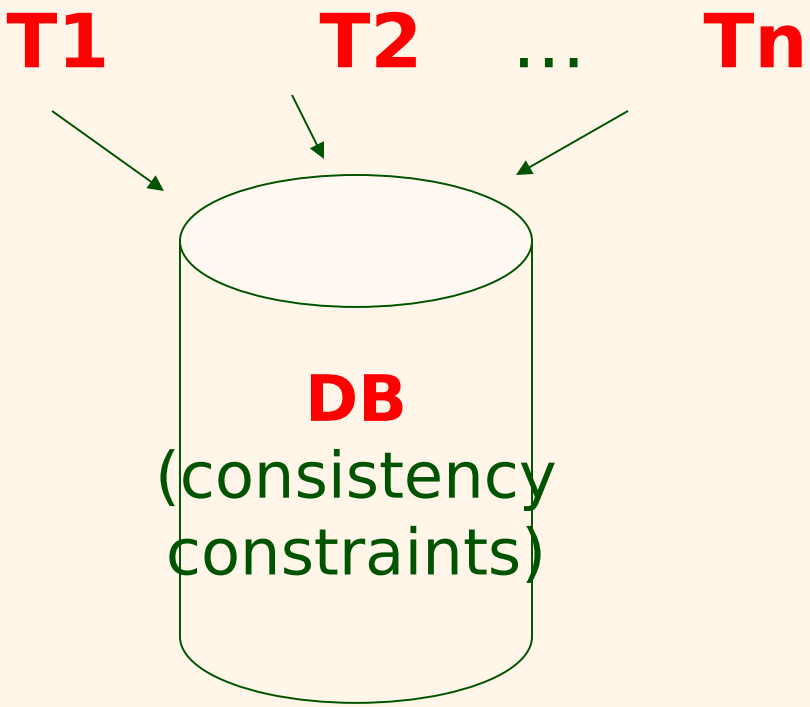
TA, Fernando (A – L).

TA, Alvaro (M – Z).

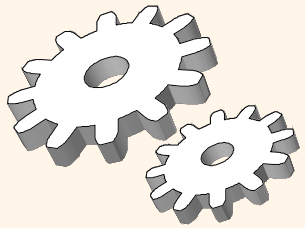
Please compare CANVAS vs. TEAMS Attendance.
Print screens of students in CANVAS but not in the TEAMS meeting.
(4.15.2024 Attendance X missing LastName.docx)

Concurrency *Control*





The Scheduler



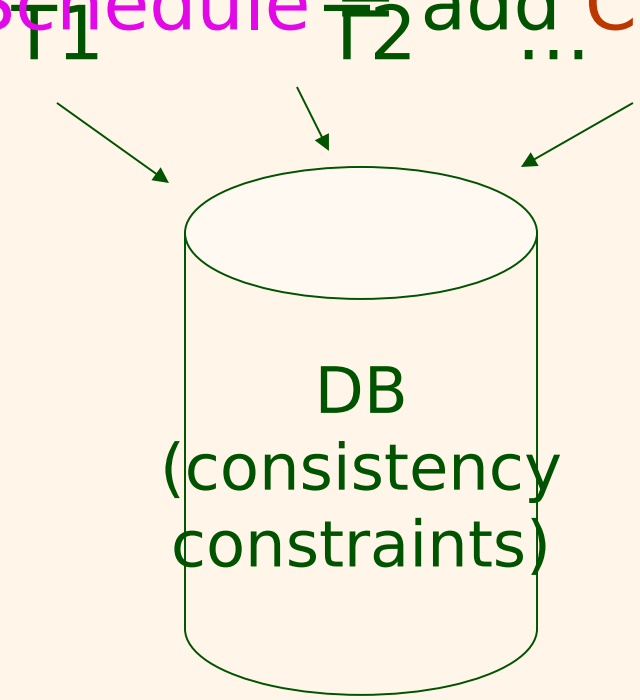
- Bases its actions on Concurrency Control Algorithms
- Ensures computer's central processing unit (**CPU**) is used **Efficiently**
- Facilitates **data Isolation** to ensure that two Transactions do not **update same data** **ıent** at **same time**

	TRANSACTIONS		
	T1	T2	RESULT
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

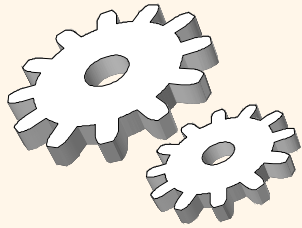
Schedules

Schedule = an interleaving of actions (**read**/**write**) from a set of transactions T1, T2, …, Tn, where the actions of **any single transaction** are in the original order

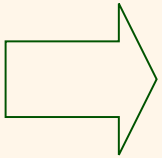
Complete Schedule = add Commit or Abort at the end



Complete Schedule



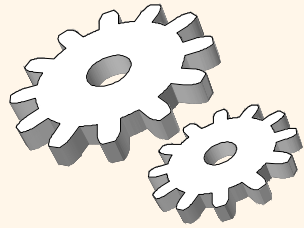
```
Transactions
T1:                     T2:
read(acc1)

                        read(acc1)
                        acc1 := acc1 + 20
                        write(acc1)
                        commit

read(acc1)
sum := sum + acc1
write(sum)
commit
```



```
Schedule

read1(acc1)

read2(acc1)

write2(acc1)
commit2
read1(acc1)

write1(sum)
commit1
```

Initial State of **DB** + **Schedule** → Final State of **DB**

Serial Schedule

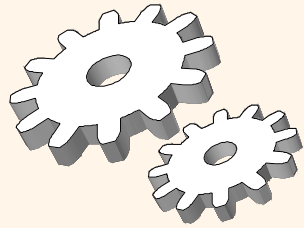


- One **Transaction** at a time, **no interleaving**

```
T1:                     T2:
                        read(acc1)
                        acc1 := acc1 + 20
                        write(acc1)
                        commit
read(acc1)
read(acc1)
sum := sum + acc1
write(sum)
commit
```

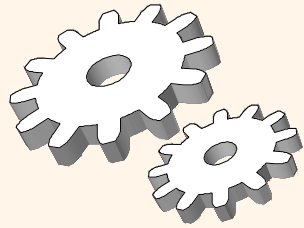
- Final state consistent (if **Transactions** are)
- **Different Serial Schedules** give different final states

Serializable Schedule



- **Schedule** with interleaved transactions that produces the same result as ***some* Serial Schedule**
- "Good" **Schedules**

Checking ***Serializability***

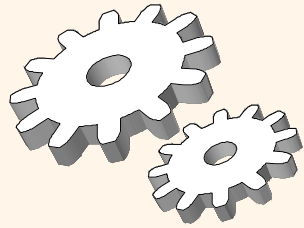


- Idea: which **actions** can be swapped in a **Schedule**?

- The following **cannot be swapped** without changing the result (conflict)
  - Actions within the **same Transaction**
  - Actions in **different Transactions *on the*** *same object* if at least one action is a **write** operation

- T

Serializability Graph



- Node for each **Transaction T_i**

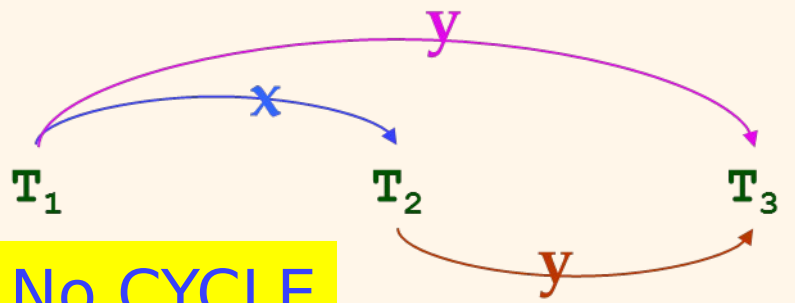
- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

- **Theorem:**

A **Schedule is Serializable** iff it's **Serializability Graph** is acyclic

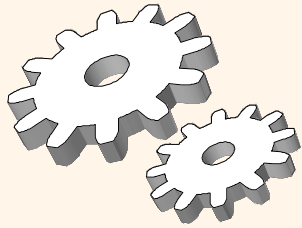
(NO CYCLE!).

Acyclic?



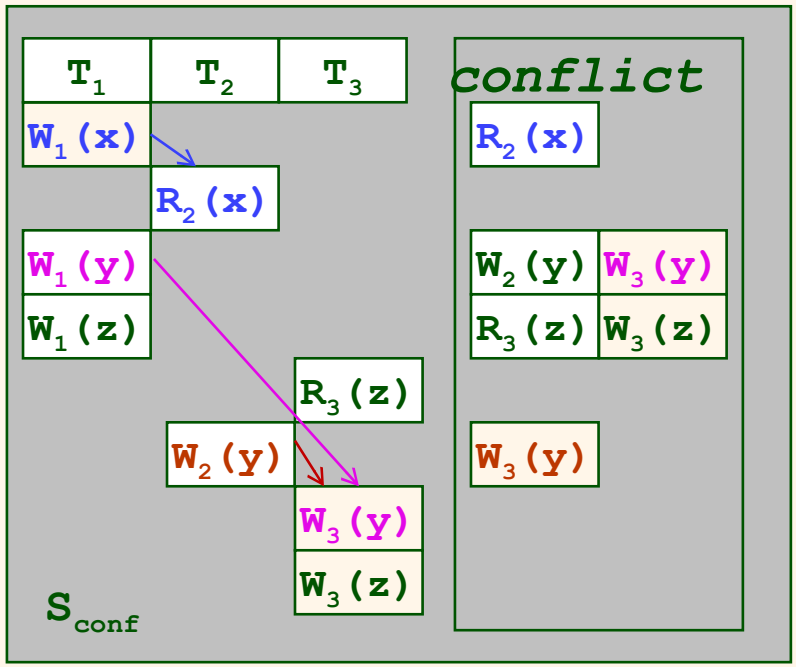
YES! No CYCLE

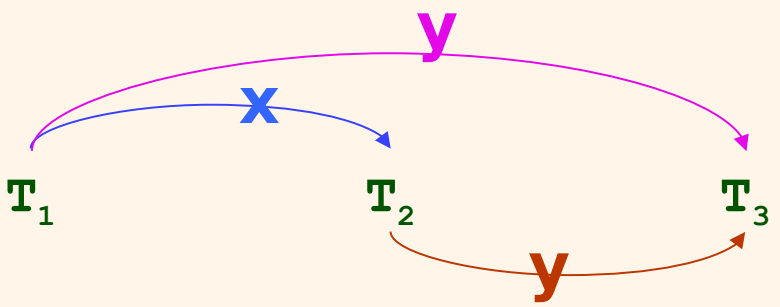
Example: Serializable Schedule ?



Node for each **Transaction T_i**

Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j





Serializability Graph

ACYCLIC – i.e., NO CYCLE

SERIALIZABLE

1. Build the **Serializability Graph.**
2. Is it **acyclic**?
3. Is this schedule **Serializable**?

Schedules and concurrency

- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

A ***serial schedule*** is a schedule in which transactions are executed one at a time. Serial schedules have no concurrent transactions. Every transaction begins, executes, and commits or rolls back before the next transaction begins. All transactions in a serial schedule are isolated.

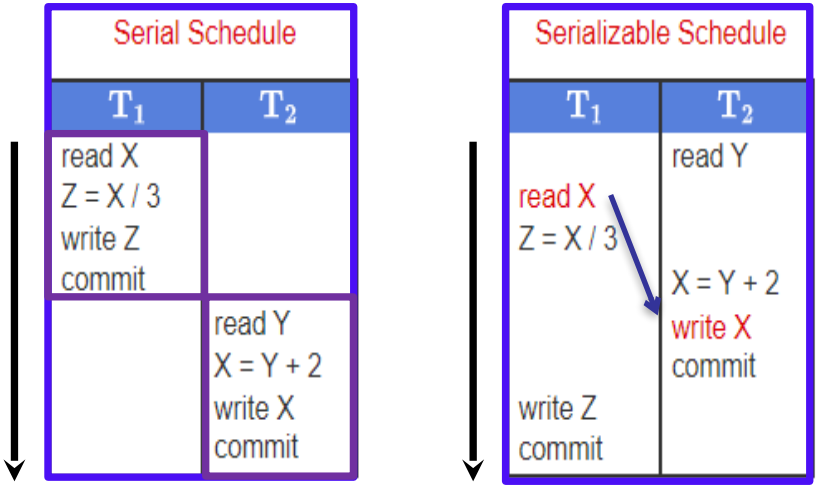
Any schedule that is equivalent to a serial schedule is a ***serializable schedule***. A serializable schedule can be transformed into a serial schedule by switching the relative order of reads in different transactions. The order of all operations within a single transaction, and reads and writes in different transactions, cannot be changed.

Serializable schedules generate the same result as the equivalent serial schedule. Therefore, concurrent transactions in a serializable schedule are isolated.

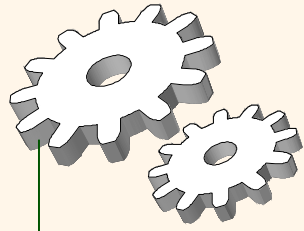
PARTICIPATION ACTIVITY 18.2.3: Serial and serializable schedules.

ACYCLIC – i.e., NO CYCLE

SERIALIZABLE



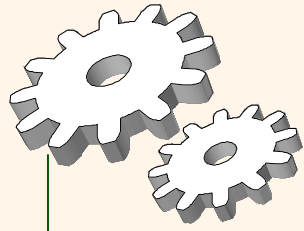
Schedule a (SERIAL)



T1	T2	A	B
Read(A); A ← A+100		25	25
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ←		
A×2;		250	
	Write(A);		
	Read(B);B ←		250
B×2;		250	250
	Write(B);		

Constraint: A=B

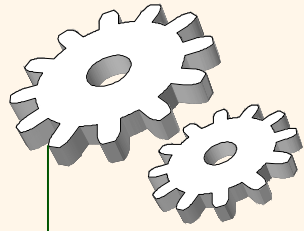
Schedule b (SERIAL)



T1	T2	A	B
	Read(A);A ←	25	25
A×2;		50	
	Write(A);		
	Read(B);B ←		50
B×2;			
	Write(B);	150	
Read(A); A ← A+100			
Write(A);			150
Read(B); B ← B+100;		**150**	**150**
Write(B);			

Constraint: A=B

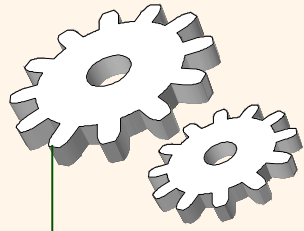
Schedule c (Good or B… Good



T1	T2	A	B
Read(A); A ← A+100		25	25
Write(A);		125	
	Read(A);A ←		
A×2;		250	
	Write(A);		
Read(B); B ← B+100;			125
Write(B);			
	Read(B);B ←		250
B×2;		**250**	**250**
	Write(B);		

Constraint: A=B

Schedule d (Good or Bad)



T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ←		
A×2;		250	
	Write(A);		
	Read(B);B ←		50
B×2;			
	Write(B);		150
Read(B); B ← B+100;		250	150
Write(B);			

Constraint: A=B

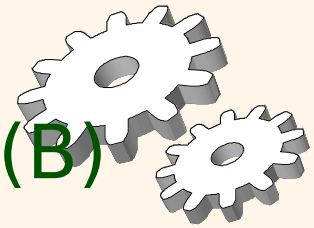
- Want **Schedules** that are "**good**", regardless of
  - initial state and
  - Transaction semantics

- Only look at **order of** read and writes

Example:

Sc=$r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

Any schedule that is equivalent to a serial schedule is a ***serializable schedule***. A serializable schedule can be transformed into a serial schedule by switching the relative order of reads in different transactions. The order of all operations within a single transaction, and reads and writes in different transactions, cannot be changed.



$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$Sa = r_1(A)w_1(A)\ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2($ **Serial T_1T_2**

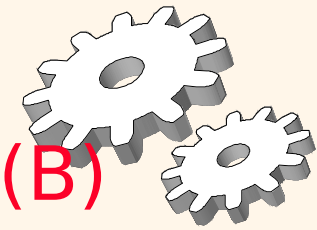
T_1 T_2

- The following **cannot be swapped** without changing the result (conflict)
  - Actions within the **same Transaction**
  - Actions in **different Transactions *on the* *same object*** if at least one action is a **write** operation

Any schedule that is equivalent to a serial schedule is a ***serializable schedule***. A serializable schedule can be transformed into a serial schedule by switching the relative order of reads in different transactions. The order of all operations within a single transaction, and reads and writes in different transactions, cannot be changed.

However, for **Sd**:

Sd=$r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$



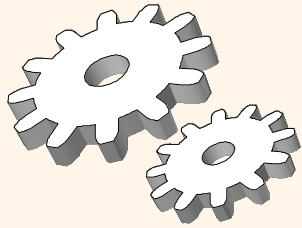
Schedule d (Good or Bad?)

Bad

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

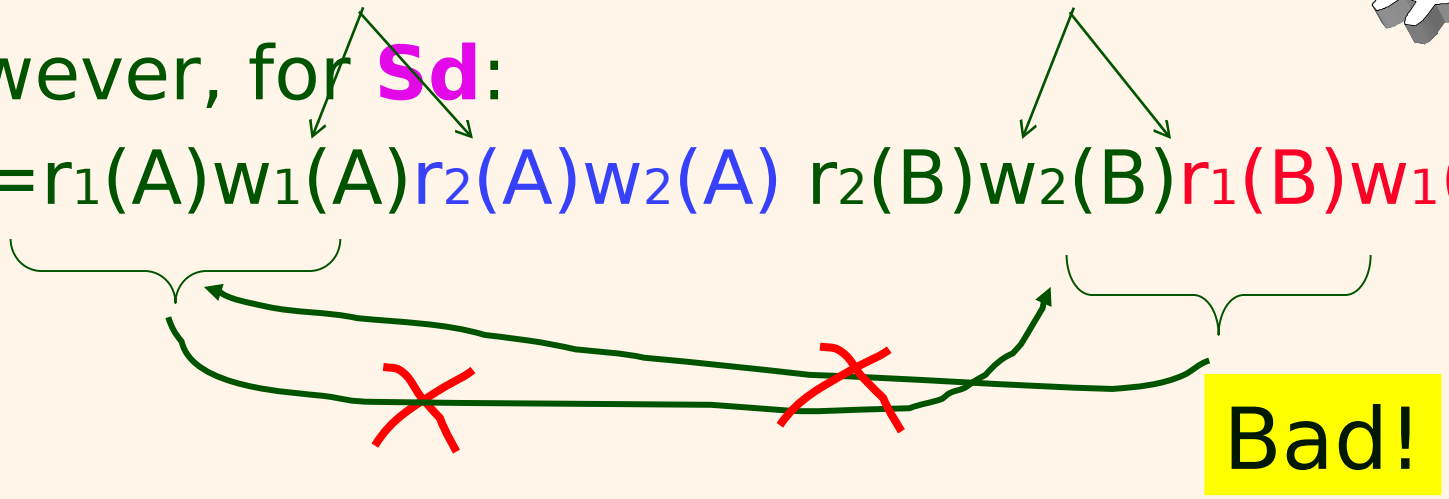
Constraint: A=B

least one action is a write op



However, for **Sd**:

Sd=$r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$



The following cannot be swapped without changing the result:

- **Actions within the same Transaction**
- **Actions in** different **Transactions** ***on the same object*** **if at**

> Any schedule that is equivalent to a serial schedule is a ***serializable schedule***. A serializable schedule can be transformed into a serial schedule by switching the relative order of reads in different transactions. The order of all operations within a single transaction, and reads and writes in different transactions, cannot be changed.

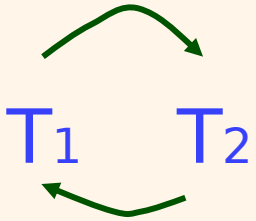
Serializability Graph

❖ Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

$Sd = r_1(A) w_1(A) r_2(A) w_2(A)\ r_2(B) w_2(B) r_1(B) w_1(B)$

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
r(B)	
w(B)	

- $T_1 \rightarrow T_2$
- Also, $T_2 \rightarrow T_1$



CYCLIC – i.e., A CYCLE

NOT SERIALIZABLE

⇨ Sd cannot be rearranged into a Serial Schedule

⇨ Sd is not "equivalent" to any Serial Sch

⇨ Sd is "Bad"

Serializability Graph

- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

Returning to **Sc**

Sc=$r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$T_1 \rightarrow T_2$

$T_1 \rightarrow T_2$

T1	T2
r(***A***)	
w(***A***)	
	r(***A***)
	w(***A***)
r(***B***)	
w(***B***)	
	r(***B***)
	w(***B***)

no cycles ⇒ **Sc** is "equivalent" to a **Serial Schedule**

(in this case T_1, T_2)

T_1 T_2

ACYCLIC – i.e., NO CYCLE

SERIALIZABLE

Question

Draw the schedules in a vertical timeline manner.

Draw the **Serializability Graph** for the schedule S below:

$w1(x)\ r1(x)\ w3(z)\ r2(x)\ r2(y)\ r4(z)\ r3(x)\ w3(y)$

- Node for each **Transaction** T_i
- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j
- **Theorem:** A **Schedule is Serializable** iff it's **Serializability Graph** is acyclic (NO CYCLE!).

If schedule S is **serializable** give an equivalent **serial schedule**, if not explain why.

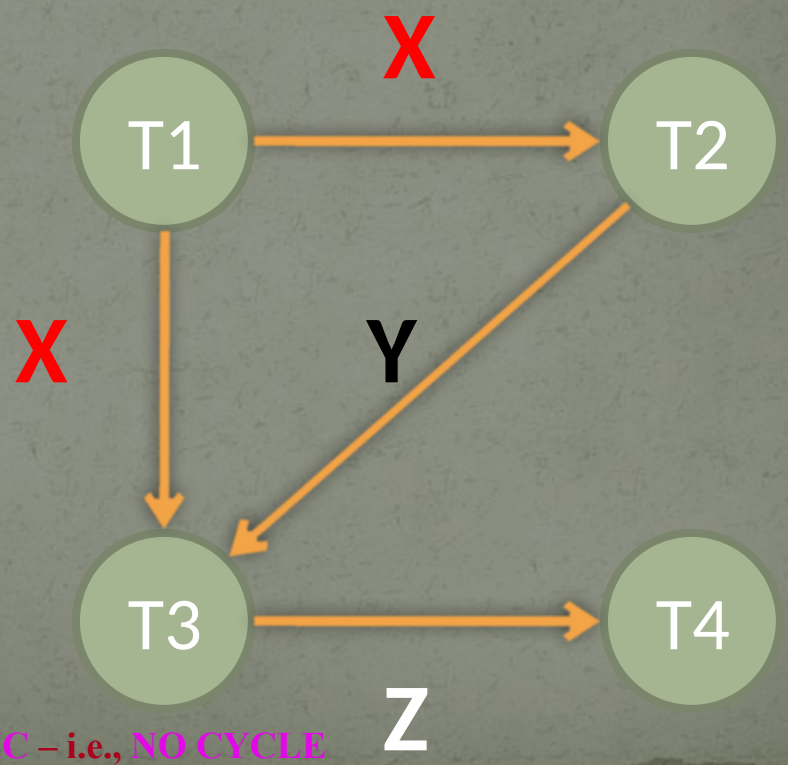
Transactions: T1, T2, T3, T4

Directed edges: (T1, T2), (T2, T3), (T1, T3), (T1, T4)

S is **Serializable** with **equivalent serial schedule**: T1, T2, T3, T4

T1	T2	T3	T4
$w(x)$			
$r(x)$			
		$w(z)$	
	$r(x)$		
	$r(y)$		
			$r(z)$
		$r(x)$	
		$w(y)$	

time



ACYCLIC – i.e., NO CYCLE

SERIALIZABLE

PARTICIPATION ACTIVITY | 18.2.4: Serial and serializable schedules.

Match the schedule type to the example schedule.

Serial schedule | **Serializable schedule** | **Non-serializable schedule**

Serial schedule

T_1	T_2
read X	
Y = X + 4	
write Y	
commit	
	read X
	X = X / 8
	write X
	commit

T_1 ends before T_2 begins. Schedules that contain no concurrent transactions are serial.

Correct

PARTICIPATION ACTIVITY | 18.2.4: Serial and serializable schedules.

- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

Match the schedule type to the example schedule.

Serial schedule | **Serializable schedule** | **Non-serializable schedule**

Serializable schedule

T_1	T_2
read X	
Y = X + 4	
write Y	
commit	
	read X
	X = X / 8
	write X
	commit

'write Y' in T_1 does not conflict with any T_2 operations. Therefore 'write Y' and 'commit' in T_1 can move prior to all T_2 operations, resulting in an equivalent serial schedule.

Correct

PARTICIPATION ACTIVITY | 18.2.4: Serial and serializable schedules.

- Edge from T_i to T_j if there is an action of T_i that precedes and "conflicts" with an action of T_j

Match the schedule type to the example schedule.

Serial schedule | **Serializable schedule** | **Non-serializable schedule**

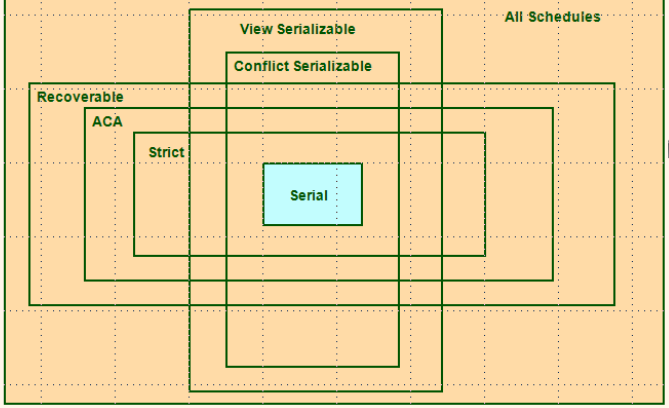
Non-serializable schedule

T_1	T_2
read X	
X = X + 4	
	read X
	X = X / 8
	write X
write X	
commit	
	commit

'write X' in T_2 conflicts with both 'read X' and 'write X' in T_1 and therefore cannot change relative order with either. Since a T_2 operation must occur between two T_1 operations, the schedule cannot be serialized.

Correct

Conflicts



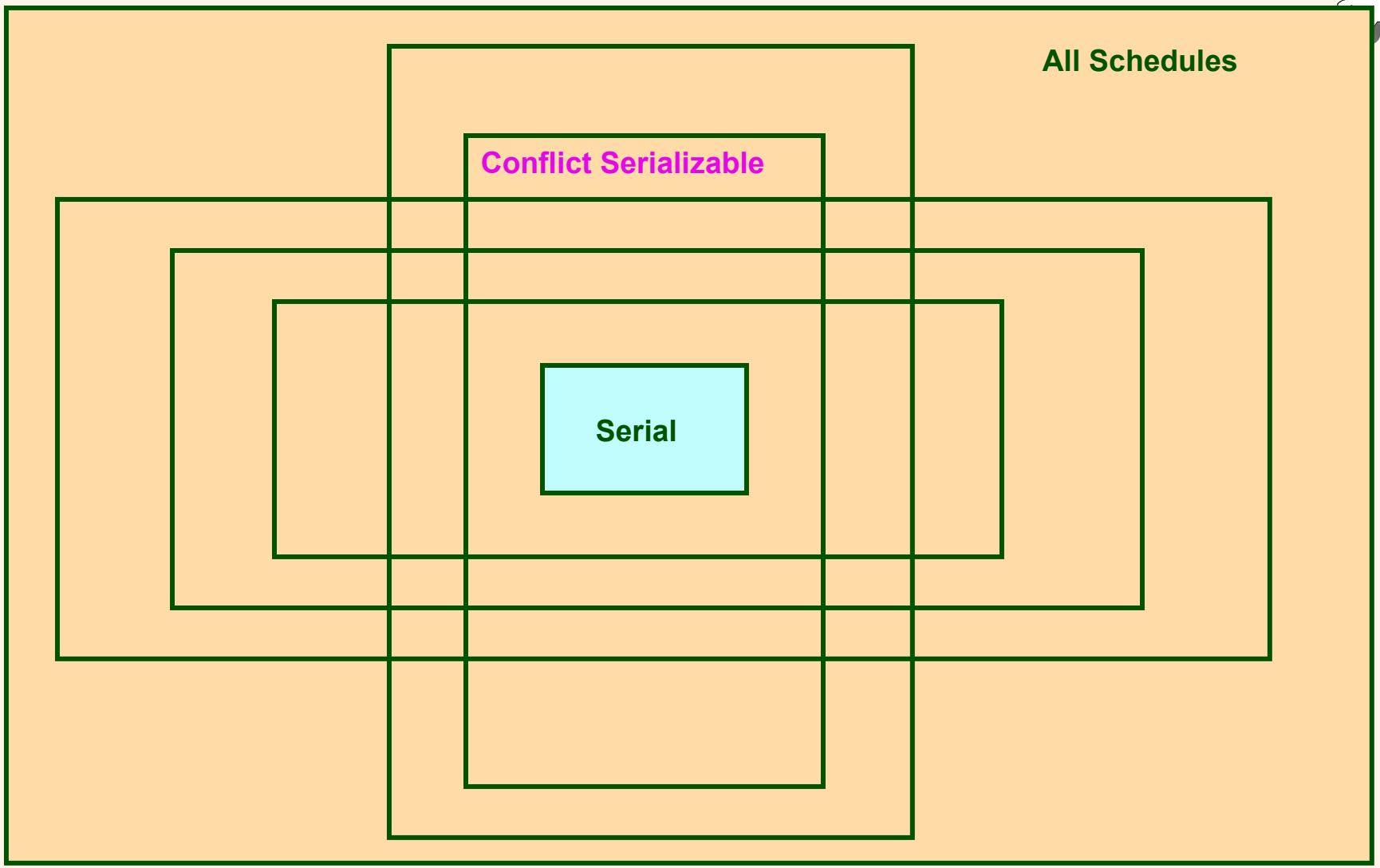
Conflicting **actions**: pairs of **actions** on **same object** from **different Transactions** where at **least one is write**

Two Schedules are ***conflict-equivalent*** if they have the same conflicts

A **Schedule is Conflict Serializable** if it is **conflict-equivalent** to a **Serial Schedule**

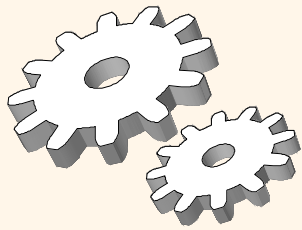
Every ***Conflict Serializable Schedule*** is

Venn Diagram for **Schedules**



A **Schedule is Conflict Serializable** if it is **conflict-equivalent** to a **Serial Schedule**

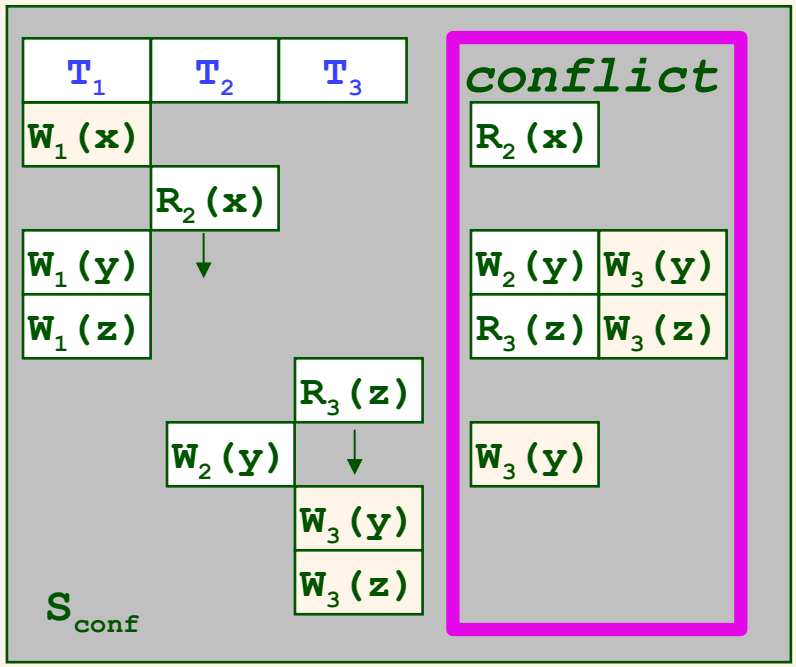
Example – Conflict Serializable

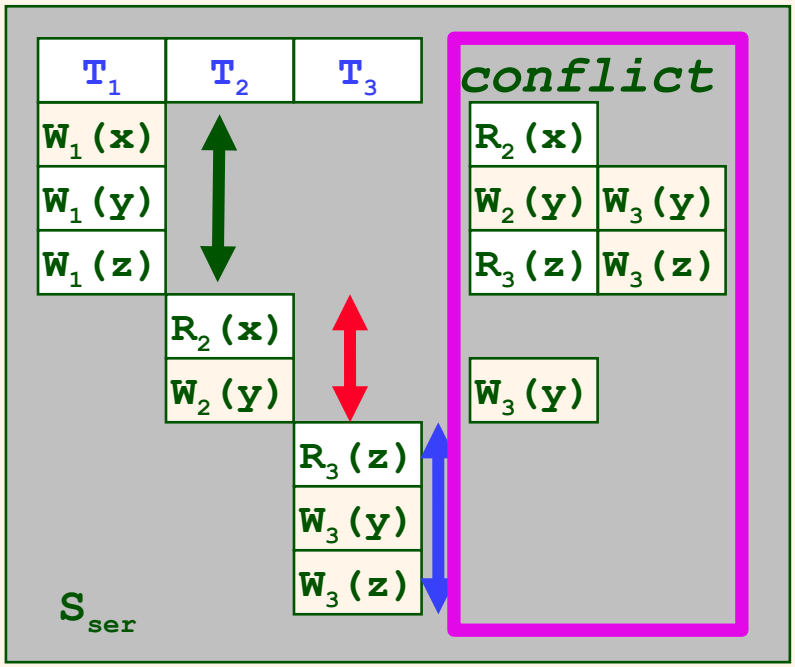


How many objects? 3: x, y, and y

How many transactions? 3: T_1, T_2, and T_3

same conflicts, thus **conflict equivalent**



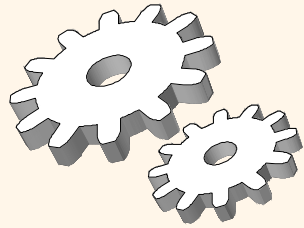


Conflict Serializable

Serial schedule

A **schedule is *conflict-serializable*** if it is conflict-equivalent to a **serial schedule**

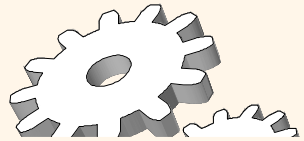
Dependency Graph



- ***Dependency graph***: One node per **Xact**; edge from ***Ti*** to ***Tj*** if ***Tj*** **reads/writes** an object last **written** by ***Ti***.

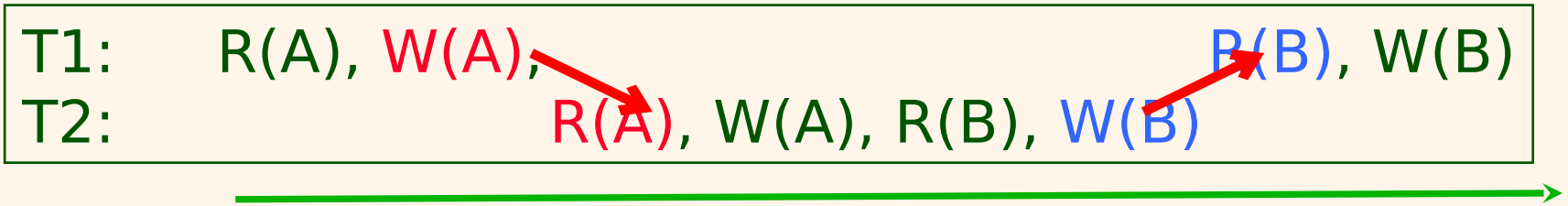
- **Theorem**: Schedule is **Conflict Serializable** if and only if its **Dependency Graph** is **acyclic**

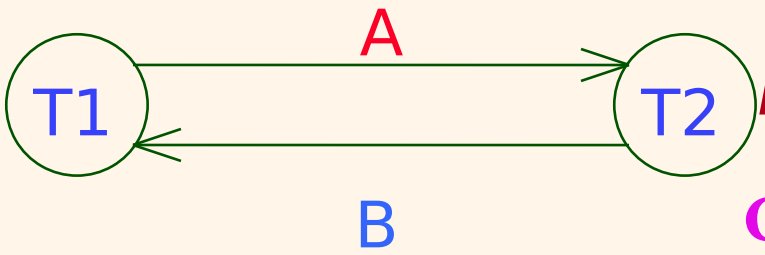
Example



Dependency Graph: One node per **Xact**; edge from ***Ti*** to ***Tj*** if ***Tj*** **reads/writes** an object last **written** by ***Ti***.

- A **Schedule** that is **not Conflict Serializable**:





Dependency Grap

CYCLIC – i.e., A CYCLE

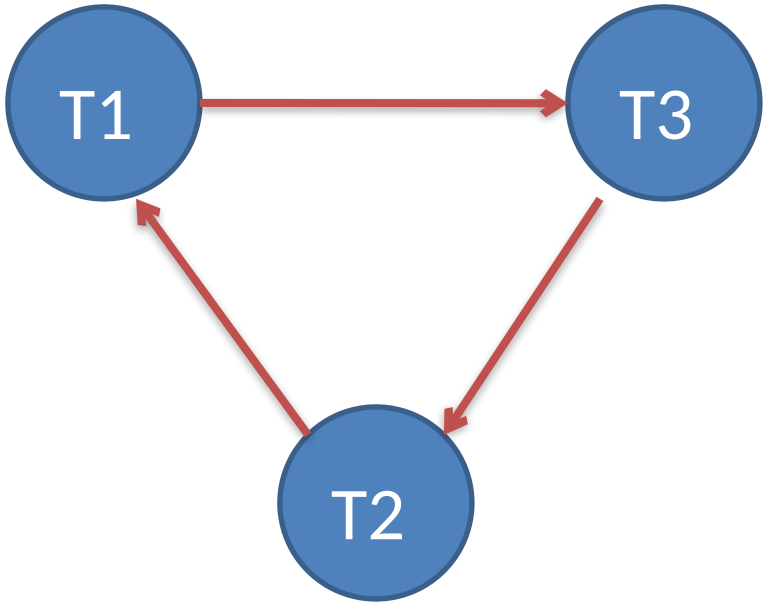
NOT Conflict Serializable

- The **cycle** in the **Graph** reveals the problem.
- The output of T1 depends on T2, and vice-

Draw the Precedence/Dependency graph for this schedule.

Node for each transaction T_i
Edge from T_i to T_j if there is an
action of T_i that precedes and "conflicts" with an action of T_j

	T1	T2	T3
0	start		
1	read X		
2		start	
3		read Y	
4	write X		
5			start
6			read X
7			write X
8			commit
9		read X	
10		write Y	
11		write X	
12		commit	
13	read Y		
14	write Y		
15	commit		



CYCLIC – i.e., A CYCLE

NOT Conflict Serializable

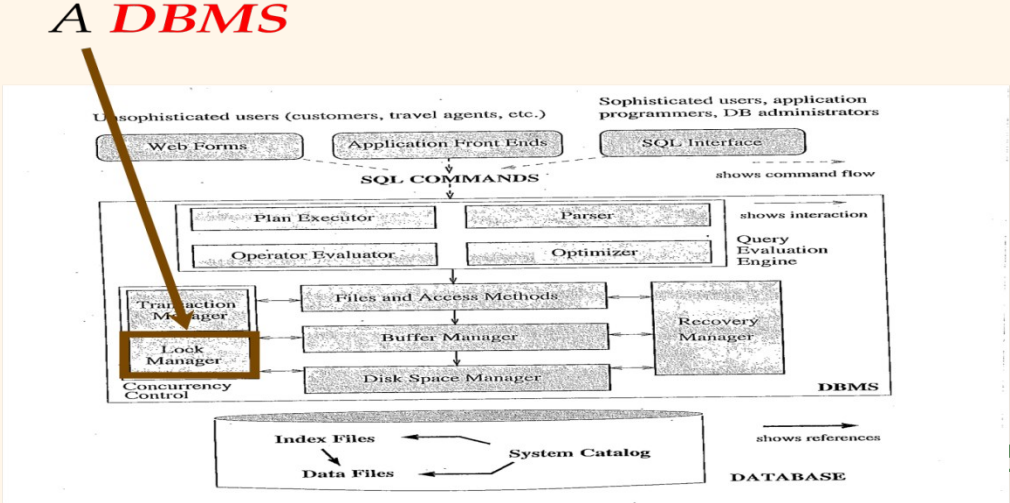
*Concurrency Control with **Locking** Methods*

Lock

- Guarantees exclusive use of a **data** item to a current Transaction
- Required to prevent another Transaction from **reading** inconsistent **data**

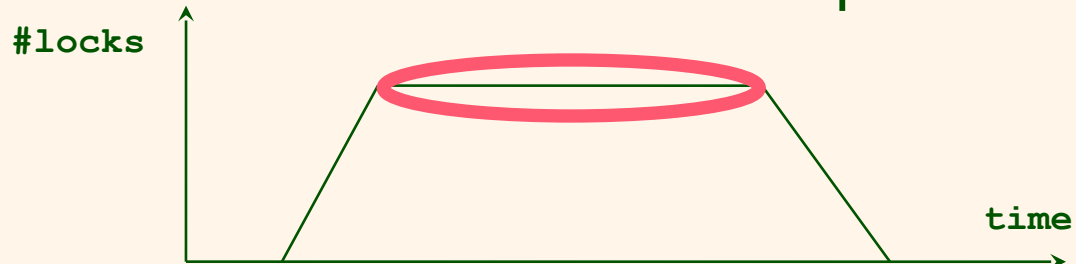
Lock Manager

- Responsible for assigning and policing the **Locks** used b Transaction**s**

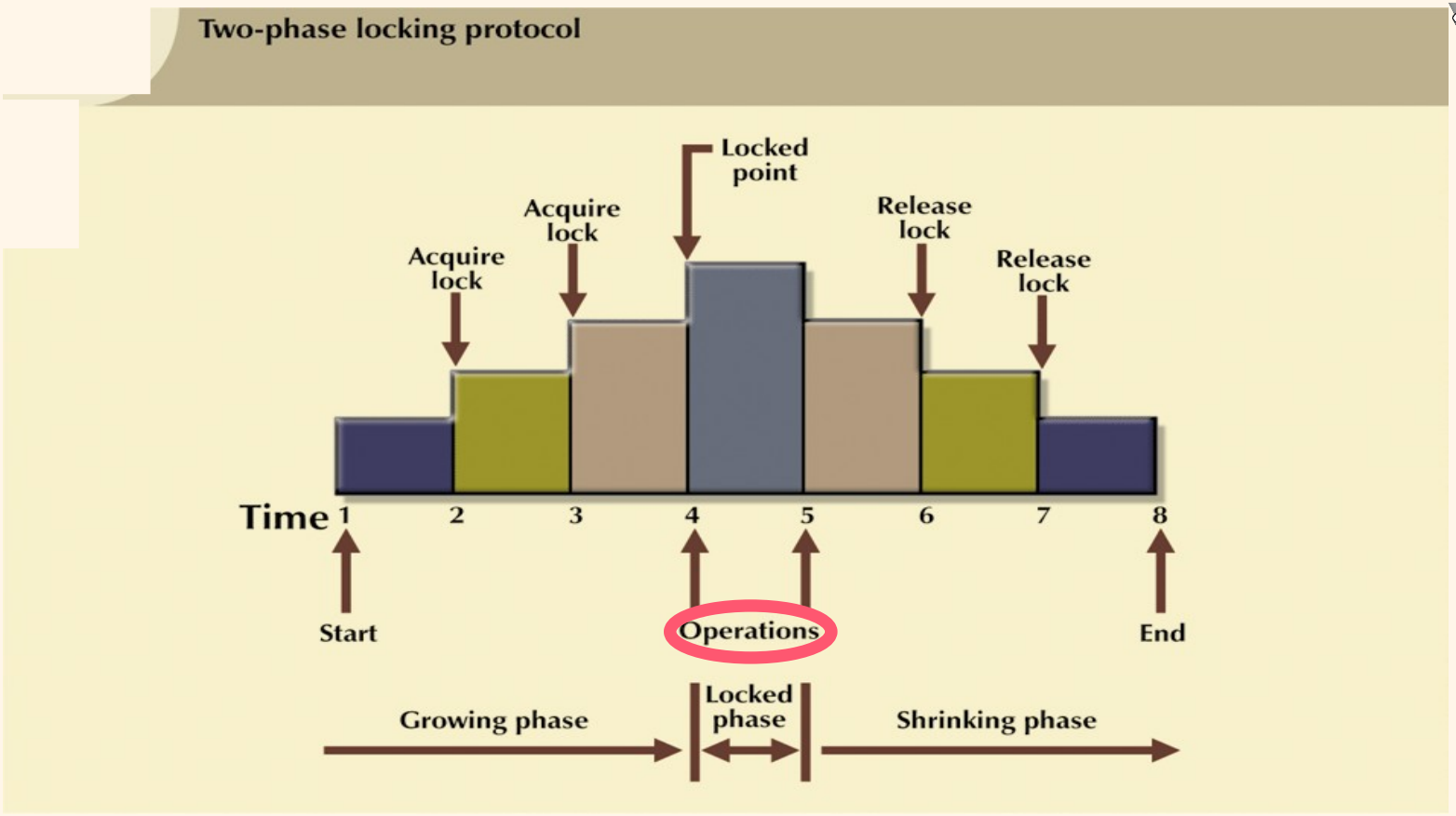


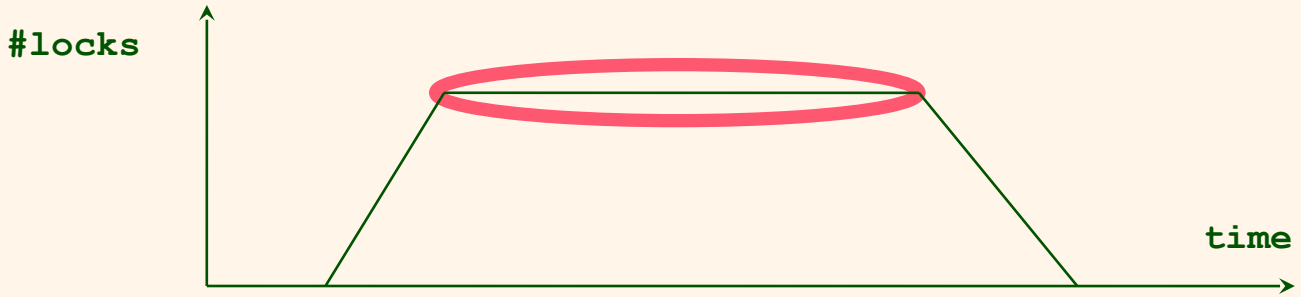
Two-Phase Locking to Ensure Serializability

- Governed by the following rules:
  - Each Xact must obtain a **S** (***shared***) **Lock** on **object** before **Reading**, and an X (***exclusive***) **Lock** on **object** before **Writing**.
  - Two Transactions cannot have conflicting **Locks**
  - No **unlock** operation can precede a lock operation in the same Transaction
  - No **data** are affected until all **Locks** are obtained—that is, until Transaction is in its locked point

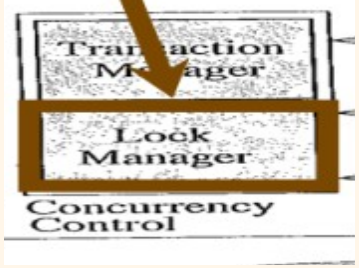


Serializability



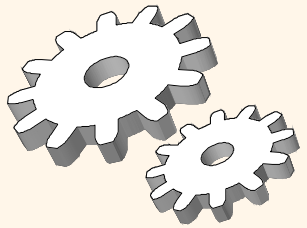


Deadlocks



- Condition that occurs when two Transactions wait for each other to **unlock data**
- Possible only if one of the Transactions wants to obtain an **exclusive Lock (X)** on **a data item**
  - No **Deadlock** condition can exist among **shared Lock**s(**S**)

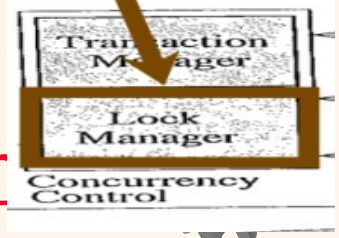
Deadlocks



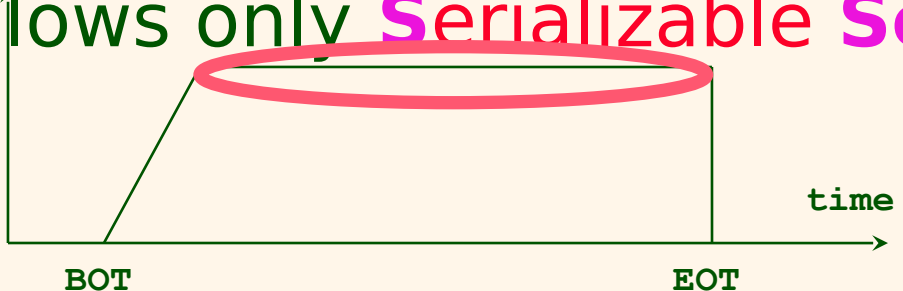
How a Deadlock Condition Is Created

TIME	TRANSACTION	REPLY	LOCK STATUS	
0			Data X	Data Y
1	T1:LOCK(X)	OK	Unlocked	Unlocked
2	T2: LOCK(Y)	OK	Locked	Unlocked
3	T1:LOCK(Y)	WAIT	Locked	Locked
4	T2:LOCK(X)	WAIT	Locked	Locked
5	T1:LOCK(Y)	WAIT	Locked	Locked
6	T2:LOCK(X)	WAIT	Locked	Locked
7	T1:LOCK(Y)	WAIT	Locked	Locked
8	T2:LOCK(X)	WAIT	Locked	Locked
9	T1:LOCK(Y)	WAIT	Locked	Locked
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....

Lock-Based Concurrency Control

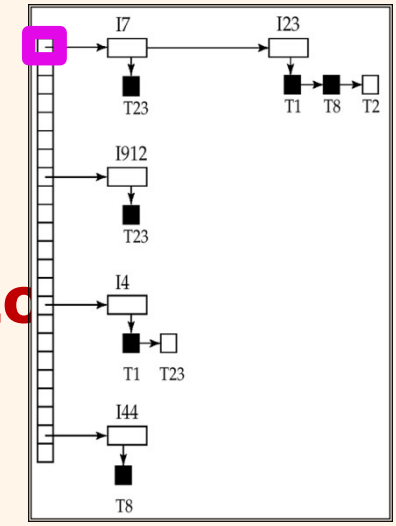


- **Strict Two-phase Locking (Strict 2PL) Pr**
  - Each Xact must obtain a **S** (***shared***) **Lock** on **object** before **Reading**, and an X (***exclusive***) **Lock** on **object** before **Writing**.
  - All **Locks** held by a Transaction are released when the Transaction completes
  - If an Xact holds an **X Lock** on an object, no other Xact can get a **Lock** (**S** or **X**) on that object.
- Strict 2PL allows only Serializable **Schedules**.



Lock Management

Lock and **Unlock** requests handled by the **Lock** Manager



Lock Table entry:

- Number of transactions currently holding a **Lo**
- Type of **Lock** held (**S**hared or e**X**clusive)
- Pointer to queue of **Lock** requests

Locking and **Unlocking** have to be atomic operations

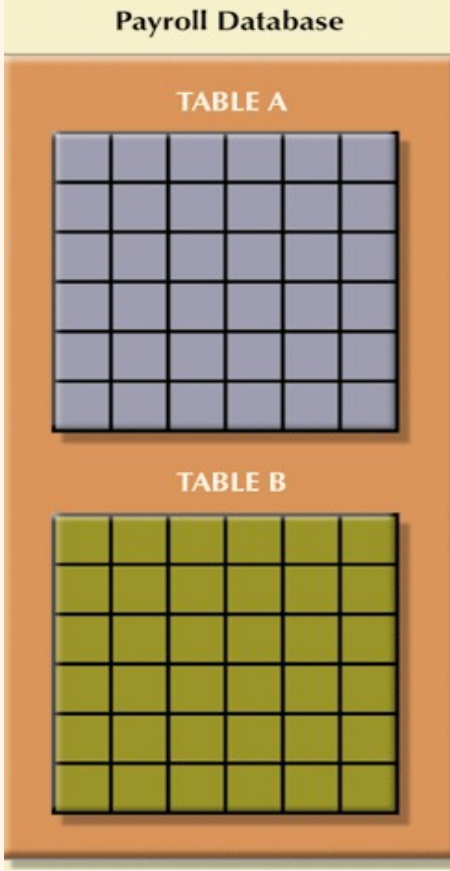
Lock upgrade: transaction that holds shared **Lock**

The concurrency system monitors active transactions, determines when locks are needed, and issues requests to grant and release locks. Requests are sent to the ***lock manager***, a component of the concurrency system that tracks, grants, and releases locks.

By requesting shared and exclusive locks as needed, the concurrency system implements the isolation level specified for each transaction.

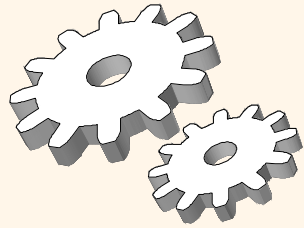
By minimizing lock scope and duration, the concurrency system reduces the duration of transactions that are waiting for locked data.

Lock Granularity



- **DATABASE-level Lock**
  - Entire **DATABASE** is locked
- **TABLE-level Lock**
  - Entire **TABLE** is locked
- **PAGE-level Lock**
  - Entire disk **PAGE** is locked
- **Row-level Lock**
  - Allows concurrent Transactions to access different **Row**s of same **TABLE**, even if Rows are located on same **PAGE**
- **Field-level Lock**
  - Allows concurrent Transaction**s** to access same **Row**, as long as they require use of different **Field**s

Deadlock Detection



Create a **Waits-For Graph** (**WFG**):

- Nodes are Transactions
- **There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock**

Periodically check for **CYCLES** in the **Waits-For Gra***Deadlocks*

How a Deadlock Condition Is Created

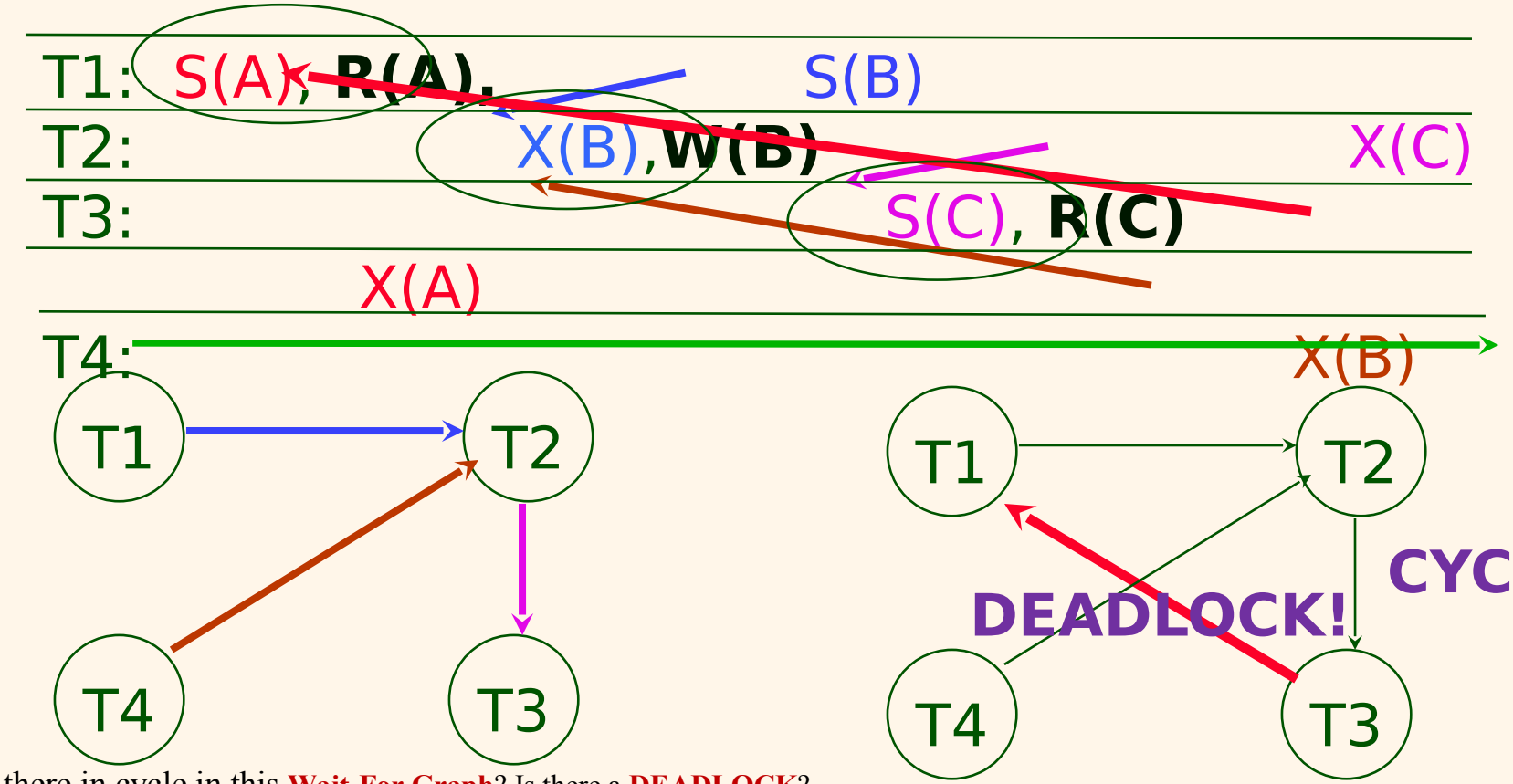
TIME	TRANSACTION	REPLY	LOCK STATUS	
0			Data X	Data Y
1	T1:LOCK(X)	OK	Unlocked	Unlocked
2	T2: LOCK(Y)	OK	Locked	Unlocked
3	T1:LOCK(Y)	WAIT	Locked	Locked
4	T2:LOCK(X)	WAIT	Locked	Locked
5	T1:LOCK(Y)	WAIT	Locked	Locked
6	T2:LOCK(X)	WAIT	Locked	Locked
7	T1:LOCK(Y)	WAIT	Locked	Locked
8	T2:LOCK(X)	WAIT	Locked	Locked
9	T1:LOCK(Y)	WAIT	Locked	Locked
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....

Deadlock Detection

T1 is waiting on T2 for B
T2 is waiting on T3 for C
T4 is waiting on T2 for B
T3 is waiting on T1 for A

There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock

Example:



7. Is there in cycle in this **Wait-For Graph**? Is there a **DEADLOCK**?

In a **deadlock** detection, transactions are allowed to wait and they not aborted until a deadlock has been detected. (Compared to other schemas where aborts transaction may have been aborted prematurely.)

Schedule S1:T1:R(X),T2:W(X),T3:W(Y),T1:W(Y),T1:Commit,T2:Commit,T3:Commit

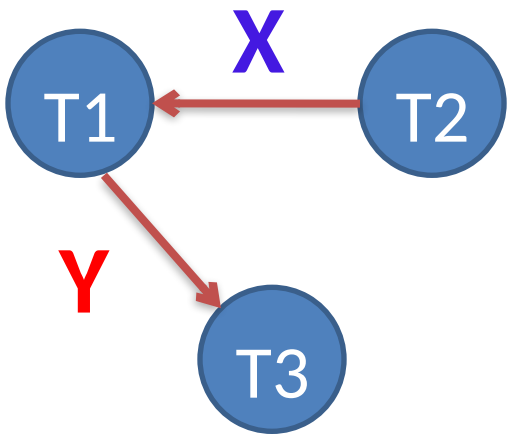
There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock

T1	S(x)Rx			X(y)Wy	commit		
T2		X(x)Wx				commit	
T3			X(y)Wy				commit

Schedule S1:

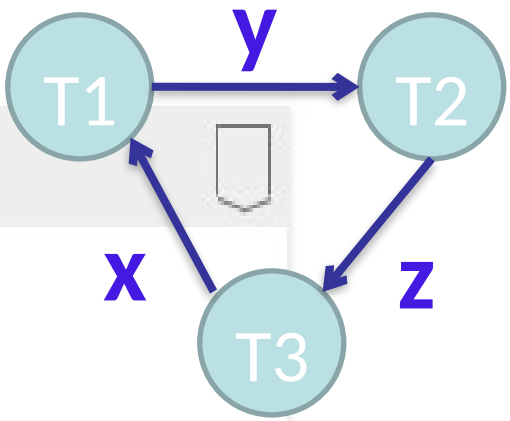
Acyclic

Not a **Deadlock**

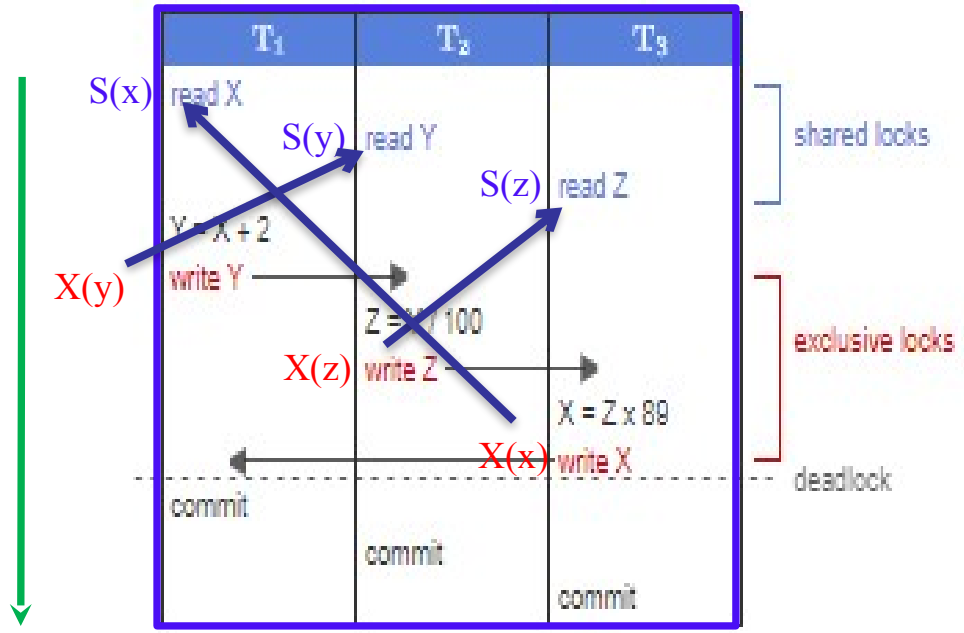


- ***Cycle detection***. The concurrency system periodically checks for cycles of dependent transactions. When a cycle is detected, the concurrency system selects and rolls back the 'cheapest' transaction. The cheapest transaction might, for example, have the fewest rows locked or most recent start time. The rollback breaks the deadlock.

- There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock



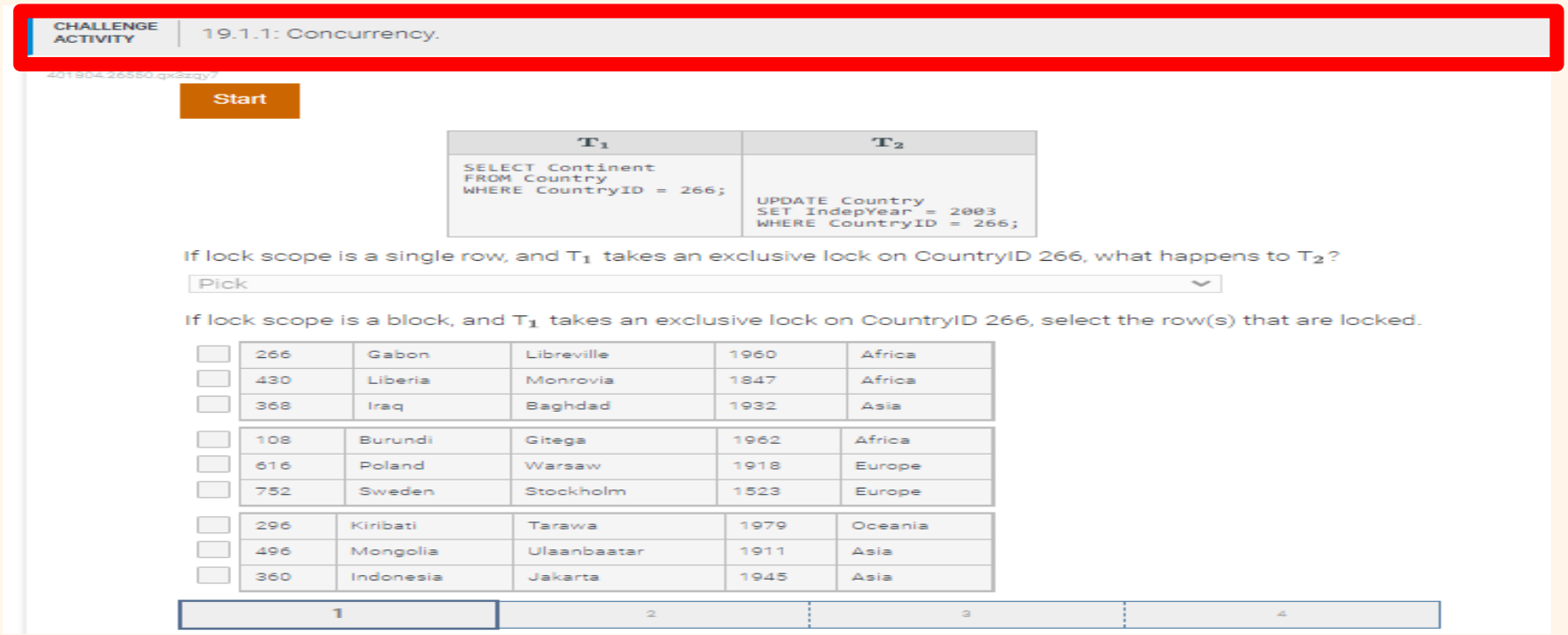
PARTICIPATION ACTIVITY | 19.1.5: Cycle of dependent transactions.



Schedule:

Cycle

Deadlock



TA time (Alvaro)
(CA 19.1.1 Step 1 – Concurrency)

TA time (Alvaro)
(CA 19.1.1 Step 2 – Concurrency)

☑	499	Montenegro	Podgorica	2006	Europe
☐	266	Gabon	Libreville	1960	Africa
☐	44	Bahamas	Nassau	1973	NAmerica
☐	484	Mexico	Mexico City	1810	NAmerica
☐	4	Afghanistan	Kabul	1919	Asia
☐	308	Grenada	St. George's	1974	NAmerica
☐	798	Tuvalu	Funafuti	1978	Oceania

1 2 3 4

Check Next

✔ Expected:
(a) The lock allows T2 to read the row with CountryID 499.
690, 860, 499

Since T_1 and T_2 access different rows, T_2 is not affected by T_1's exclusive lock on the row with CountryID 690. Thus, the lock allows T_2 to read the row with CountryID 499.

T_1 takes an exclusive lock on the block containing the row with CountryID 690. Thus, all rows in the entire block are locked: 690, 860, and 499.

View solution (Instructors only)

Feedback?

Urtaza, Alvaro A

Participants

Invite someone or dial a number

Share invite

In this meeting (110) Mute all

Hilford, Victoria Organizer
Adhikari, Rohit
Ahmed, Mohamed A
Akram, Ali
Alvarez, Stephanie
Anayor-Achu, Ogochukwu E
Avci, Hatice Kubra
Aysola, Riya
Bahl, Anish
Banza, Sean Paolo B
Bui, Hieu
Burger, Jake
Carrillo-Zepeda, Victor E
Carter, Deric

COSC 3380 23578 M & W 4 to 5:30 PM LECTURE (in TEAMS from 3:50 to 5:15 PM)

01:09:53

Take control | Pop out | Chat | People 110 | Raise | View | Rooms | Apps | More | Camera | Mic | Share | Leave

AU Urtaza, Alvaro A

TN Nguyen, Tri...

View all

zyBooks My library > COSC 3380: Database Systems home > 19.1: Concurrency

zyBooks catalog | Help/FAQ | Alvaro Urtaza

Consider the following transaction schedule:

T_1	T_2
read W	
	read X
X = 79 - W write X	
	W = X + 82 write W
commit	
	commit

Does a deadlock occur when using:

aggressive locking? No

cycle detection? Yes

1 2 3 4

Check Next

✔ Expected: No, Yes

When the database uses aggressive locking:
Each transaction requests all locks when the transaction starts, so the transactions cannot participate in a deadlock.

When the database uses cycle detection:
Transactions first take shared locks when reading, then request exclusive locks when writing.
T_1 reads and creates a shared lock on W. T_2 reads and creates a shared lock on X.
T_2 waits for T_1 to release the shared lock on W. T_1 waits for T_2 to release the shared lock on X. Deadlock occurs.
The concurrency system detects the cycle between T_1 and T_2, rolls back the 'cheapest' transaction, and breaks the deadlock.

View solution (Instructors only)

Feedback?

Urtaza, Alvaro A

Participants

Invite someone or dial a number

Share invite

In this meeting (110) Mute all

Hilford, Victoria Organizer
Adhikari, Rohit
Ahmed, Mohamed A
Akram, Ali
Alvarez, Stephanie
Anayor-Achu, Ogochukwu E
Avci, Hatice Kubra
Aysola, Riya
Bahl, Anish
Banza, Sean Paolo B
Bui, Hieu
Burger, Jake
Carrillo-Zepeda, Victor E
Carter, Deric

Concurrency Control Practice Questions

A database administrator is updating one of the tables in the database. Which of the following techniques can prevent other database administrators from doing concurrent transactions to the same table?

a. Mirroring

b. Scoping

c. Logging

d. Locking

A ____ is a part of the concurrency system that monitors, grants, and releases locks.

a. Lock Administrator

b. Lock System

c. Lock Manager

d. Lock Optimizer

In a ____, all transactions come to a halt and remain at a standstill until one of the transactions is aborted.

a. timeout

b. deadlock

c. two-phase locking

d. locking

Which deadlock management technique automatically rolls back a transaction when a lock is not released in a fixed period of time?

a. Timeout

b. Aggressive locking

c. Data ordering

d. Cycle detection

Refer to the table and schedule below. The UPDATE statement in transaction T1 holds an exclusive lock on the Class table. With strict two-phase locking, when does the SELECT statement in transaction T2 execute?

T_1	T_2
UPDATE Class SET TeacherID = 32412 X(Class) WHERE ClassID = 80;	
	SELECT CourseTitle FROM Class WHERE ClassID = 80;
ROLLBACK;	

Strict Two-phase Locking (Strict 2PL) Protocol:

- Each Xact must obtain a **S** (*shared*) **Lock** on **object** before **Reading**, and an X (*exclusive*) **Lock** on **object** before **Writing**.
- All **Locks** held by a Transaction are released when the Transaction completes
- If an Xact holds an **X Lock** on an object, no other Xact can get a **Lock** (**S** or **X**) on that object.

Strict 2PL allows only Serializable **Schedules**.

63

a. Immediately after the UPDATE statement in T1 executes

b. After the ROLLBACK statement in T1 executes

c. At the same time as the UPDATE statement in T1 executes

d. The SELECT statement in T2 will never execute

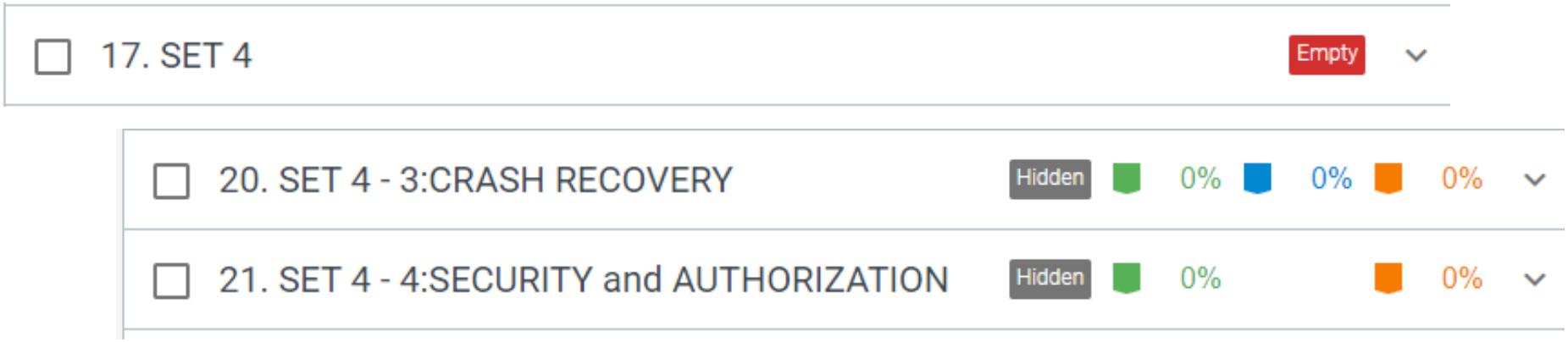
At 5:00 PM .

04.15.2024 (24 – Mo)	ZyBook SET 4 - 2	Set 4 LECTURE 17 CONCURRENCY CONTROL

VH work on
SET 4 – 2: Concurrency

Next

04.17.2024 (25 – We)	**ZyBook SET 4 - 3**	**Set 4** LECTURE 18 CRASH RECOVERY LECTURE 19 SECURITY and AUTHORIZATION



VH, unHIDE

From 5:05 to 5:15 PM – 5 minutes.

04.15.2024 (24 – Mo)	ZyBook SET 4 - 2	Set 4 LECTURE 17 CONCURRENCY CONTROL

CLASS PARTICIPATION 20 points — 20% of Total

CONCURRENCY

Class 24 END PARTICIPATION
Not available until Apr 15 at 5:05pm | Due Apr 15 at 5:15pm

VH, publish

This is a synchronous online class.

Attendance is required.

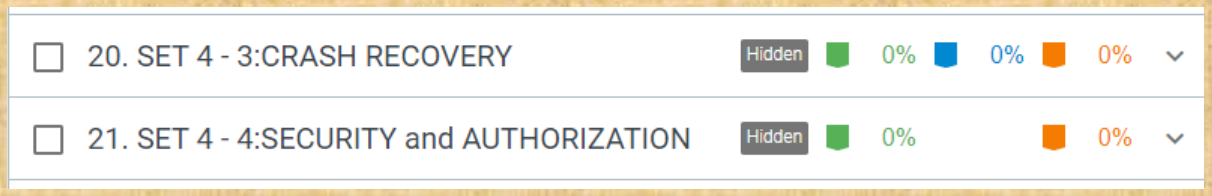
Recording or distribution of class materials is prohibited.

1. At the beginning of selected classes there is an **assessment in the first 10 minutes**. (**beige** BOX in the Detailed Syllabus)
2. At the end of selected classes there is an **assessment in the last 10 minutes**. (**blue** BOX in the Detailed Syllabus)
3. ZyBook sections will be downloaded and used for 30% of Total Score on the dates specified in the Detailed Syllabus.
4. EXAMS are in CANVAS. **No late EXAMS**.
5. I have to be **present in TEAMS in order to take any graded assignment assigned during that class.**

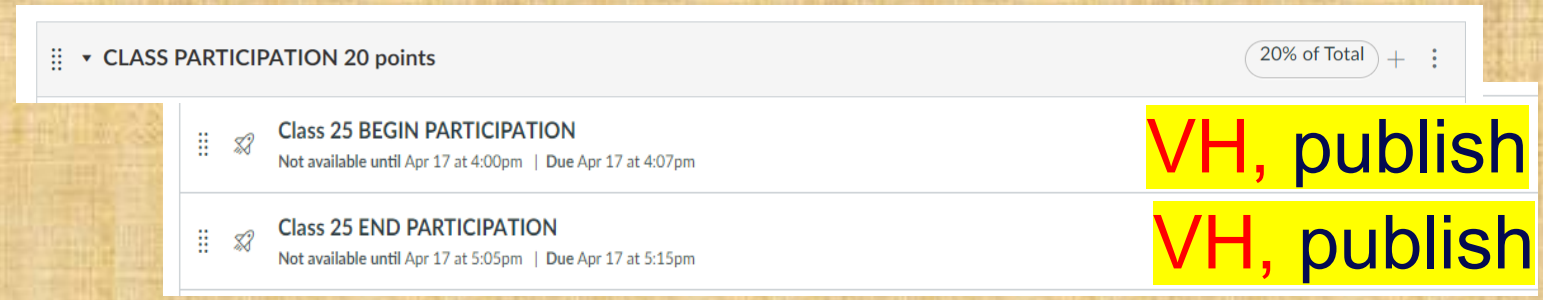
At 5:15 PM.

End Class 24

VH, unHIDE ZyBook SET 4 – 3,4 : RECOVERY & SECURITY.



VH, Download Attendance Report
Rename it:
4.15.2024 Attendance Report FINAL TEAMS



VH, upload Class 24 to CANVAS.