



## Before-or-after atomicity and Serializability

Xingda Wei, Yubin Xia

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

### Review: what is a strong consistency model

#### It's easy for users to reason about correctness assuming

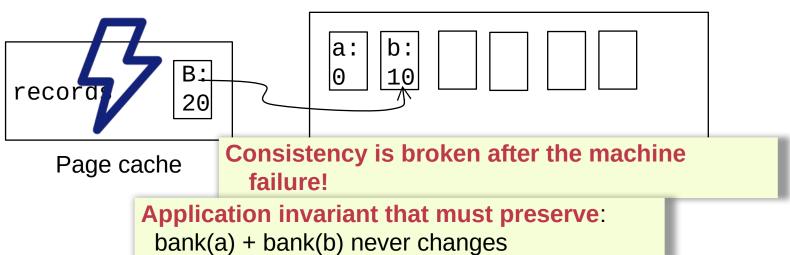
- Everything has only one-copy
- The overall behavior is equivalent to some serial behavior
- The operations that need to be executed in an atomic unit (usually called operations belonging to a transaction) are executed on a machine that happens completely or not at all (all-or-nothing atomicity)

#### All-or-nothing atomicity makes it much easier to reason about failures

 Need to think about the consequences of the action happening or not happening, but not about the action partially happening

### Review: why we need all or nothing atomicity?

```
transfer(bank, a, b, amt): // amt=10
    records = mmap(bank, ...)
    records[a] = records[a] - amt
    records[b] = records[b] + amt
    fsync(bank, ...)
}
```



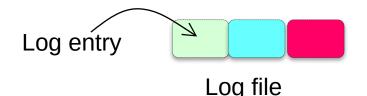
### Review: techniques to support all-or-nothing

#### Systematic methods to support all-or-nothing atomicity

- Shadow copy (Single-file updates)
- Journaling (Filesystem API)

#### Logging

REDO logging (A general-purpose approach)





### **Review: redo-only logging**

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] - amt
    new b = records[b] + amt
    commit log = "log start: a:" + new a + "\b:" +
new b
    log.append(commit_log).sync()
    record[a] = new_a
    record[b] = new b
```

#### log.append(commit\_log).sync()'s return is the commit point

After it returns, the transfer's updates must be all

The whole all-or-nothing unit is usually called transaction (TX)







### **Crash recovery of commit log**

#### After reboot, we need to recover the systems to a consistent state

Based on the log entries stored in the log file

#### Rules

- Travel from start to end
- 2. Re-apply the updates recorded in a complete log entry

### Review: Pros & Cons of redo-only logging so far

#### **Pros**

- The commit is extremely efficient: only one file append operations (w/ updated data)
  - Other methods, e.g., shadow copy copies the entire file

#### Cons

- Wastes of disk I/O: all disk operations must happen at the commit point
- All updates must be buffered in the memory until the transaction commits
  - What if there is insufficient memory?
- The log file is continuously growing while most its updates are already flushed to the disk (unless the machine is rebooted or crashed, and we do the recovery)

### **Pros & Cons of redo-only logging so far**

#### **Pros**

- The commit is extremely efficient: only one file append operations (w/updated data)
  - Other methods, e.g., shadow copy copies the entire file

#### Cons

- Wastes of disk I/O: all disk operations must happen at the commit point
- All updates must be buffered in the memory until the transaction commits
  - What if there is insufficient memory?
- The log file is continuously growing while most its updates are already flushed to the disk

Unlike filesystem journaling, the user can commit a lot of entries in a transaction

#### Basic idea

#### We allow the transaction directly writing uncommitted values to the disk

Before the commit point to free-up memory space & utilize disk I/O

```
transfer(bank, a, b, amt, log): // amt=10
  records = mmap(bank, ...)

records[a] = records[a] - amt The OS will flush the page back if out of the memory
  records[b] = records[b] + amt
```

#### **Problem**

How to prevent a partial updates from uncommitted transactions?

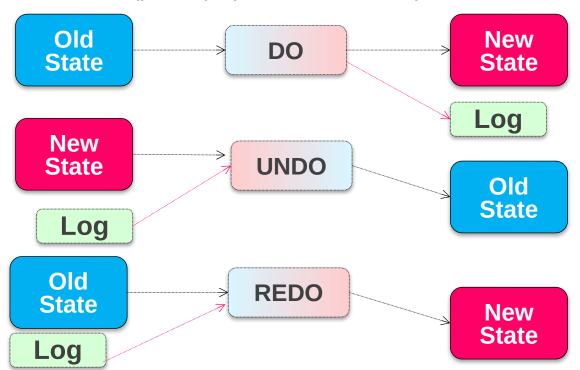
Idea: use log to **undo** updates of uncommitted transactions!

### **Undo logging**

### Keep a log of all update actions

Log

- The log can undo the (partial) updates of a DO operation



### Logging w/ undo

#### Before updates, write an undo log record to the log file

- Should contain sufficient information to undo uncommitted transactions
- E.g., old values

Question: do we need the redo entry?

### Logging w/ undo-redo logging

#### Question: do we need the redo entry?

- Depends on whether we wait for records[a] to be written to the disk (e.g., sync)
- Typically, yes: waiting two disk syncs are slow!
  - Especially for non-logging writes: log is a fast sequential disk write

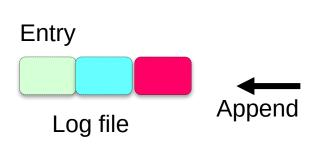
### Log entry vs. log record

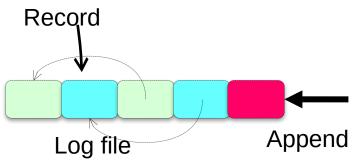
#### Redo-only logging appends log entry to the log file

Containing all the updates of the transaction

#### Undo-redo logging appends log records to the log file

- Containing the updates of a single operation
- Log records from different transaction (TX) may possibly interleave
  - E.g., the OS schedules the transaction out
  - Therefore, we further need pointer to trace operations from the same TX

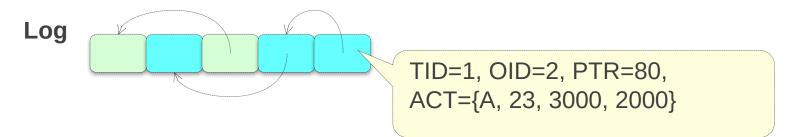




### Put it all together: log record in undo-do logging

#### Each log record consists of

- 1. Transaction ID
- 2. Operation ID
- 3. Pointer to previous record in this transaction
- 4. Value (file name, offset, old & new value)
- 5. ...

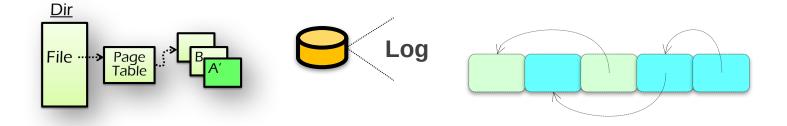


### Put it all together: logging rules

# Write log record to disk before modifying persistent state

- (e.g., replace A's value)
- Write Ahead Log (WAL) protocol

```
transfer(bank, a, b, amt, log): //
amt=10
  records = mmap(bank, ...)
  log.append(...).sync()
  records[a] = records[a] - amt
  log.append(...).sync()
  records[b] = records[b] + amt
  log.append("TX {id} commit").sync()
```



### Put it all together: logging rules

### Write log record to disk before modifying persistent state

- (e.g., replace A's value)
- Write Ahead Log (WAL) protocol

### At commit point, append a commit record to the log last

E.g., when user calls commit

```
Commit
Dir
                                               Log
File .....
         Page
Table
```

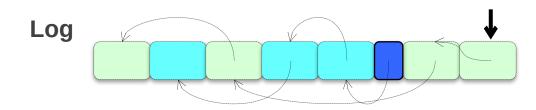
```
transfer(bank, a, b, amt, log): //
amt=10
  records = mmap(bank, ...)
  log.append(...).sync()
  records[a] = records[a] - amt
  log.append(...).sync()
  records[b] = records[b] + amt
  log.append("TX {id} commit").sync()
```

### Recovery rules How to recovery from crash?

#### Read the log and recover states according to its content

#### **Rules:**

1. Travel from end to start



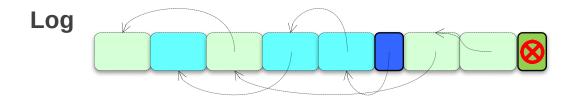
### Recovery rules

### How to **recovery from crash**?

#### Read the log and recover states according to its content

#### **Rules:**

- 1. Travel from end to start
- 2. Mark all TX's log record w/o CMT ABORT log



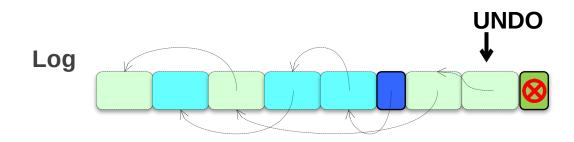
### Recovery rules

### How to **recovery from crash**?

#### Read the log and recover states according to its content

#### **Rules:**

- 1. Travel from end to start
- 2. Mark all TX's log record w/o CMT ABORT log
- 3. UNDO ABORT logs from end to start



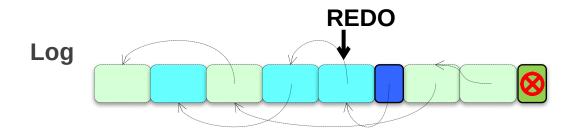
### **Recovery rules**

### How to **recovery from crash**?

#### Read the log and recover states according to its content

#### **Rules:**

- 1. Travel from end to start
- 2. Mark all TX's log record w/o CMT ABORT log
- 3. UNDO ABORT logs from end to start
- 4. REDO CMT logs from start to end



### Problem: continuously growing of the log file

#### Both redo-only logging & undo-redo logging append to the log file

- The log file is continuously growing while most its updates are already flushed to the disk
- The log is only deleted if there is a single machine failure

#### Typically, a machine fails less frequent

E.g., one per day

#### We need checkpoint the log file to reduce the log file size!

 Checkpoint: Determining which parts of the log can be discarded, then discarded them

### **Checkpoint the log**

#### Naïve solution

- Run the recovery process. If it is done, then we can discard all the log file
- Problem: too slow

#### **Observation**

- For redo logging, we only need to flush the page caches so we can discard all the logs of committed TXs
- For undo logging, we only need to wait for TXs to finish to discard all its log entries

### **How to checkpoint? (For both undo + redo logging)**

#### **Basic approach**

- 1. Wait till no TXs are in progress
- 2. Flush the page cache
- 3. Discard all the logs

#### Question

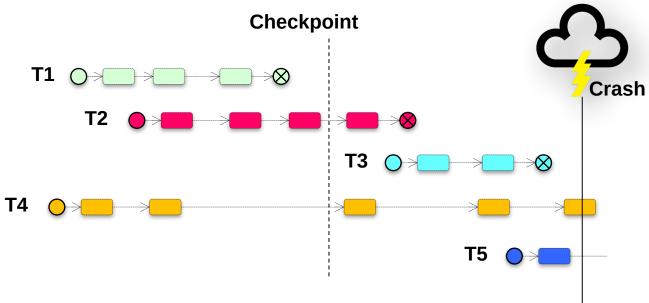
- What if a TX is doing a long time? Can we allow ongoing TXs?
- We need to reserve the log for ongoing TXs!

### **How to checkpoint?**

### How to checkpoint?

- actions
- 1. Wait till no transactions are in progress
- 2. Write a **CKPT** record to log
  - Contains a list of all transaction in process and their logs
- 3. Flush the page cache
- 4. Discard all the log records except the CKPT record

### **Recovery with checkpoint**



- 1. T1: do nothing
- 2. T2: redo its update based on the log in its checkpoint
- 3. T3: redo its updates based on the log entries
- 4. T4: undo its updates based on the log in both CKP & log entries
- 5. T5: undo its updates based on the log entries

### Undo-redo logging vs. redo-only logging

#### **Question:**

- Which one is faster during execution?
- Which one is faster during recovery?

#### **Redo-only logging**

- Less disk operations compared with undo-redo logging
- Only need one scan of the entire log file

Redo-only logging is typically preferred except for TXs with large inmemory states

### **UNDO-only Logging**

#### **Logging rules**

- Append UNDO log record before flushing state modification
- State modification must be flushed before transaction committed
  - w/o REDO

#### Rarely used

- Much slower than UNDO-REDO logging during execution
- Though the recovery speed is faster

### **Summary & durability**

#### Systematic methods to support all-or-nothing atomicity

- Shadow copy (Single-file updates)
- Journaling (Filesystem API)

#### **Logging (General-purpose approaches)**

- REDO-only logging (commit logging)
- UNDO-REDO logging (write-ahead logging)
- UNDO-only logging

#### **Durability:** Committed Action's data on durable storage

Logging is also a method to ensure durability

## **Before-or-after atomicity**

We will evolve from single-threaded to multithreaded setting

### What is a strong consistency model

#### It's easy for users to reason about correctness assuming

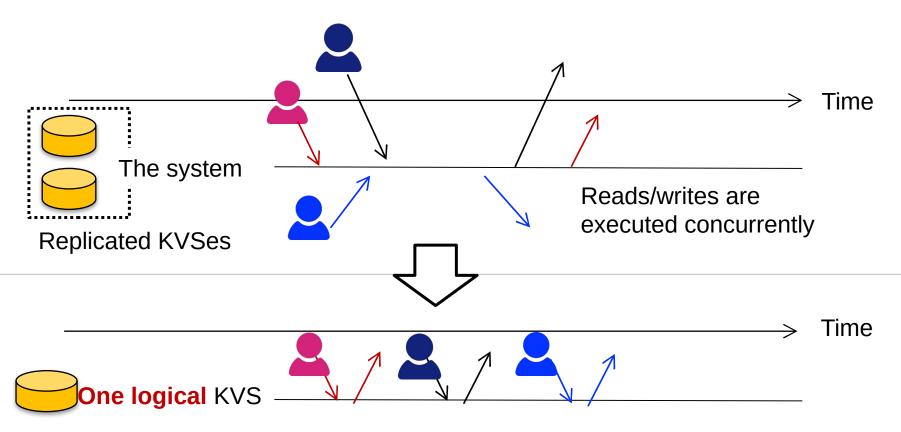
Everything has only one-copy



### The overall behavior is equivalent to some serial behavior

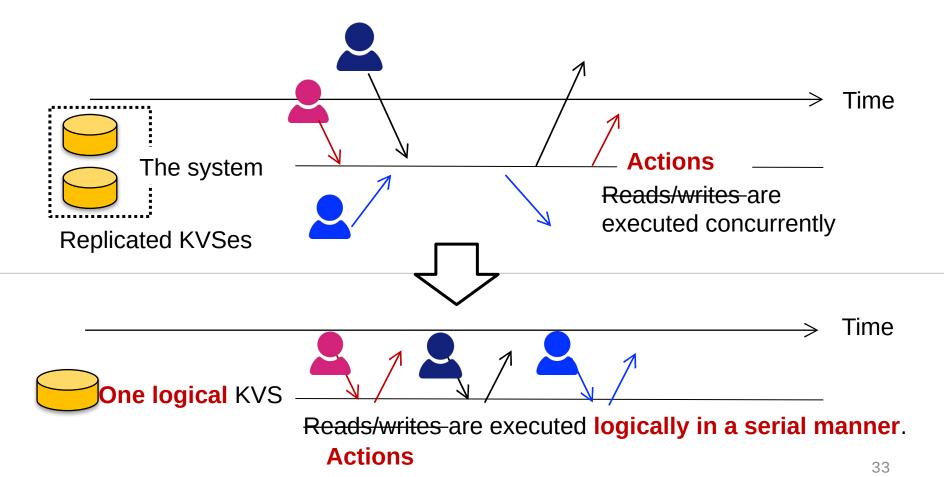
 The operations that need to be executed in an atomic unit (usually called operations belonging to a transaction) are executed on a machine that happens completely or not at all (all-or-nothing atomicity)

### Review: serial execution of reads/writes



Reads/writes are executed logically in a serial manner.

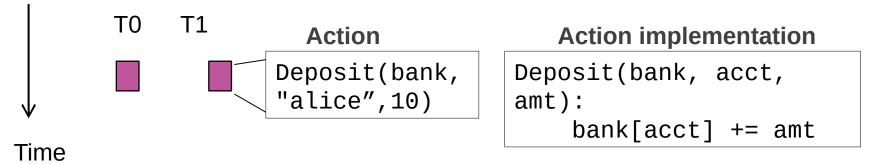
### One step further: serial execution of actions



### **Motivating example**

**Example**: consider **two** threads **access the same account concurrently** 

We have a in-memory array to store all the accounts w/o losing generality

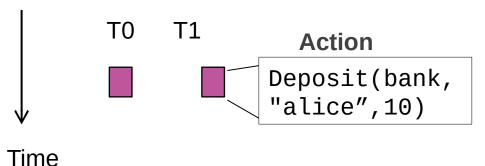


**Question**: What are the expected behavior?

### **Motivating example**

**Example**: consider **two** threads **access the same account concurrently** 

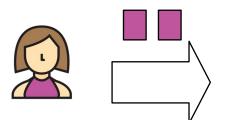
We have a in-memory array to store all the accounts w/o losing generality



#### **Action implementation**

Deposit(bank, acct,
amt):
 bank[acct] += amt

**Question**: What are the expected behavior?



Alice: 0¥

Alice: 10¥? (Expected result: 20¥)

### **Atomicity of the + operator**

#### **Example**: consider **two** threads **access the same account concurrently**

Even a simple + operator is non-atomic

#### Why would this happen?

- The deposit function is not atomic, even assuming each mov is linearizable!
  - CPU can only guarantee the atomicity of one memory op, but it has two

```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

```
mov acct, %eax add $amt, %eax mov %eax, acct
```

Compiler's generated code

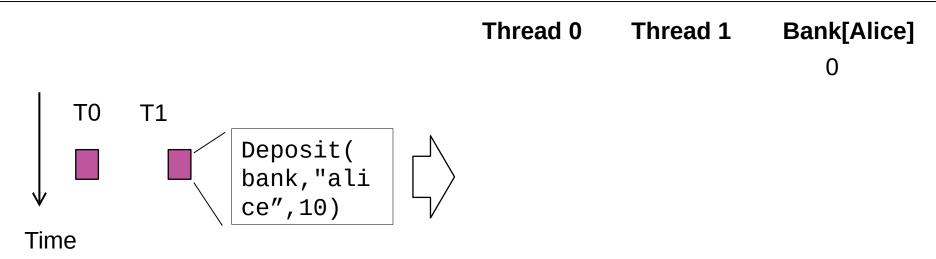
### **Atomicity of the + operator**

```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

```
mov acct, %eax add $amt, %eax mov %eax, acct
```

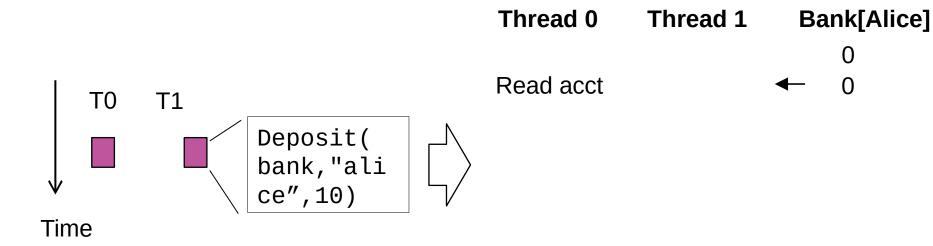
Compiler's generated code



```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

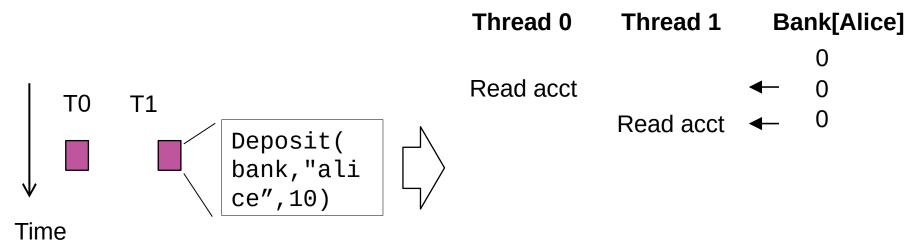
```
mov acct, %eax add $amt, %eax mov %eax, acct
```



```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

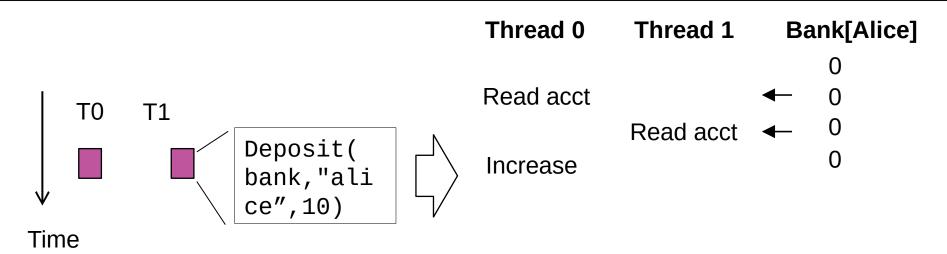
```
mov acct, %eax add $amt, %eax mov %eax, acct
```



```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

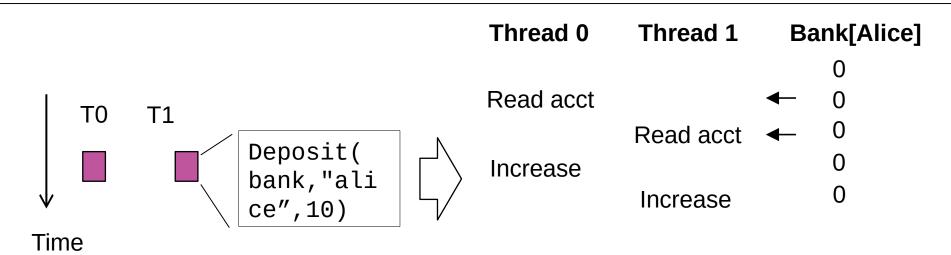
```
mov acct, %eax add $amt, %eax mov %eax, acct
```



```
Deposit(bank, acct, amt):

bank[acct]. += amt
Developer's written code
```

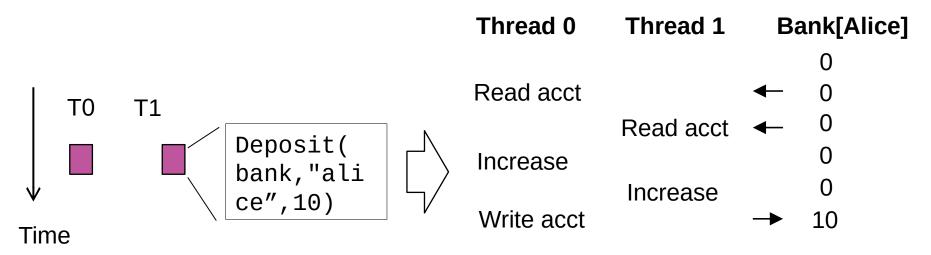
```
mov acct, %eax add $amt, %eax mov %eax, acct
```



```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

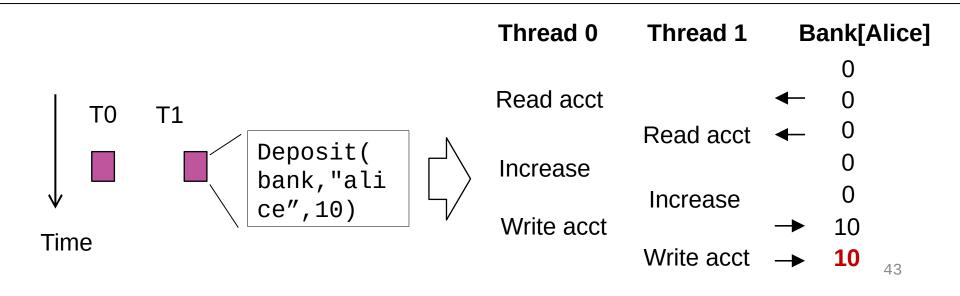
```
mov acct, %eax add $amt, %eax mov %eax, acct
```



```
Deposit(bank, acct, amt):

bank[acct] += amt
Developer's written code
```

```
mov acct, %eax add $amt, %eax mov %eax, acct
```



#### The race condition problem

#### When two or more threads access shared data and at least one is write

#### Timing dependent error involving shared state

- i.e., whether the scheduling will cause reading a non-atomic update state

Thread 0	Thread 1	В	ank[Alice]	Thread 0	Thread 1	Ba	nk[Alice]
Read acct		4	0	Dood cost		4	0
Redu deci	Read acct	<b>←</b>	0 0	Read acct Increase			0 0
Increase			0	Write acct		<b>→</b>	10
Write acct	Increase	<b>→</b>	0 10		Read acct Increase	<b>←</b>	10 10
vviite acct	Write acct	<b>→</b>	10		Write acct	<b>→</b>	<b>20</b>
							44

#### Race condition is hard to control

#### Must make sure all possible schedules are safe

- Number of possible schedules permutations is huge
- Bad schedules that will and will not work sometimes

#### They are intermittent

- Small timing changes between invocations might result in different behavior which can hide bug (e.g., Therac-25)
- Also known as Heisenbugs (Heisenberg uncertainty)
  - Possible system solution-1: DMT (Deterministic Multi-Threading)
  - Possible system solution-2: Record and Replay
  - Both are hard in either correctness or cost, or both

#### Race condition is common (benign case)

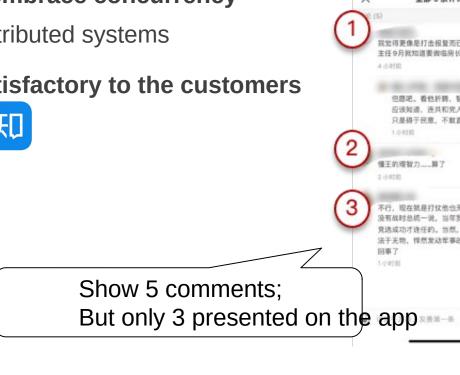
#### Real-world computing systems embrace concurrency

E.g., embrace the design of distributed systems

#### Bad: race condition brings unsatisfactory to the customers

E.g., the comment system in





■ 知学

全部 5 条评论

ad 40 €

0 0

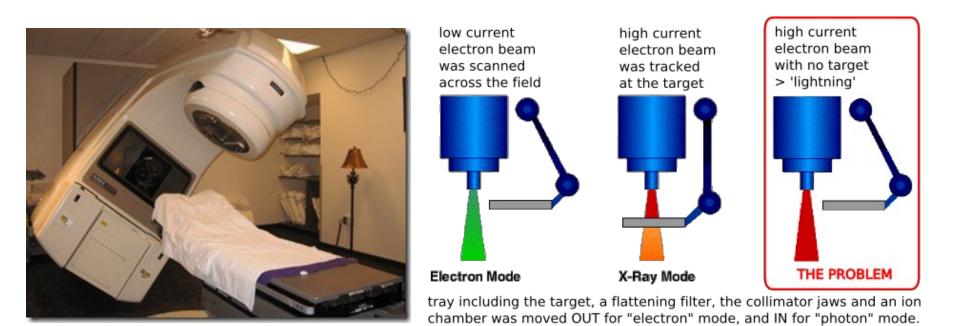
0 0

0 0

发布

=1

#### Race condition may be dangerous (e.g., in Therac-25)



The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly. In three cases, the injured patients later died as a result of the overdose.

47

# **Before-or-After Atomicity**

#### Goal: before-or-after atomicity

To prevent hazard produced by race condition, we need a group of reads/writes to be atomic

E.g., cannot see/overwrites the intermediate states of a concurrent action

Thread 1 should not see the intermediate states of thread 0

Thread 0	Thread 1	В	ank[Alice]	
			0	
Read acct		<b>←</b>	0	
	Read acct	<b>←</b>	0	
Increase			0	
	Increase		0	
Write acct		<b>→</b>	10	
	Write acct	<b>→</b>	10	

#### Goal: before-or-after atomicity (a.k.a, Isolation)

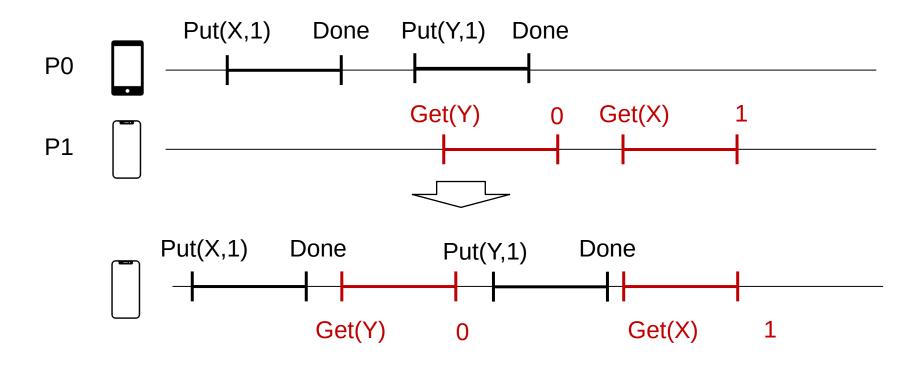
To prevent hazard produced by race condition, we need a group of reads/writes to be atomic

- E.g., cannot see/overwrites the intermediate states of a concurrent action
- A concurrent action may have multiple linearizable reads/writes

#### Before or after atomicity

Concurrent actions have the before-or-after property if their effect from the point of view of their invokers is as if the actions occurred either completely before or completely after one another

#### Before-or-after vs. Linearizability



The above schedule is allowed in Linearizability, but not before-or-after atomicity

#### Use locking to achieve before-or-after

Locks: a data structure that allows only one CPU acquire at a given time

- Programs can acquire and release the lock
- If the acquisition fails, then the CPU will wait until it is succeed
- Example: pthread\_mutex

#### Implementing the locks efficiently is a broad topic

And is non-trivial: out of the scope of this lecture

**Locks:** a data structure that allows only one CPU acquire at a given time

- Programs can acquire and release the lock
- If the acquisition fails, then the CPU will wait until it is succeed
- Example: pthread\_mutex

#### **Global lock:**

 The action must acquire the global lock before executing, and release it after it commits

```
Deposit(bank, acct, amt):
bank[acct] += amt
```



```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Deposit(bank, acct,
amt):
 bank[acct] += amt



```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Thread 1

Bank[Alice]

Deposit(bank, acct,
amt):
 bank[acct] += amt



Thread 0	Thread 1	В	ank[Al	ic
			0	
Read acct		<b>←</b>	0	
	Read acct	<b>←</b>	0	
Increase			0	
	Increase		0	
Write acct		<b>→</b>	10	
	Write acct	<b>→</b>	10	

```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Thread 0 Thread 1 Bank[Alice]

Acquire(lock)

0

Deposit(bank, acct,
amt):
 bank[acct] += amt



Thread 0	Thread 1	Bank[Alio		
			0	
Read acct		<b>←</b>	0	
	Read acct	<b>←</b>	0	
Increase			0	
	Increase		0	
Write acct		<b>→</b>	10	
	Write acct	<b>→</b>	10	

```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Thread 0	Thread 1	В	ank[A
A oguiro (lo ola)			0
Acquire(lock)			U
Read acct		<b>←</b>	0

Deposit(bank, acct,
amt):
 bank[acct] += amt



Thread 0	Thread 1	В	Bank[Alice]			
			0			
Read acct		<b>←</b>	0			
	Read acct	<b>←</b>	0			
Increase			0			
	Increase		0			
Write acct		<b>→</b>	10			
	Write acct	<b>→</b>	10			

```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Thread 0 Thread 1 Bank[Alice]

0

Acquire(lock) 0

Read acct ← 0

Acquire(lock) 0

Deposit(bank, acct,
amt):
 bank[acct] += amt



Thread 0	Thread 1	Bank[Alice]		
			0	
Read acct		<b>←</b>	0	
	Read acct	<b>←</b>	0	
Increase			0	
	Increase		0	
Write acct		<b>-</b>	10	
	Write acct	<b>→</b>	10	

Deposit(bank, lock, acct, amt): acquire(lock) bank[acct] += amt release(lock) Thread 0 Thread 1 Bank[Alice] Acquire(lock) Read acct Acquire(lock)

Can this happen?

Read acct

Write acct →

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                      amt):
  amt):
                                          acquire(lock)
       bank[acct] += amt
                                          bank[acct] += amt
                                           release(lock)
                        Bank[Alice] Thread 0
Thread 0
            Thread 1
                                                 Thread 1
                                                              Bank[Alice]
                                    Acquire(lock)
Read acct
                                      Read acct
           Read acct
                                                Acquire(lock)
Increase
            Increase
Write acct
```

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                      amt):
  amt):
                                          acquire(lock)
       bank[acct] += amt
                                           bank[acct] += amt
                                           release(lock)
                        Bank[Alice] Thread 0
Thread 0
            Thread 1
                                               Thread 1
                                                              Bank[Alice]
Read acct
                                     Acquire(lock)
                                      Read acct
           Read acct
                                                Acquire(lock)
Increase
                                                Acquire(lock)
                                      Increase
            Increase
Write acct
                                           Thread is stuck at the
           Write acct →
```

acquiring the lock

Write acct →

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                       amt):
  amt):
                                           acquire(lock)
       bank[acct] += amt
                                           bank[acct] += amt
                                            release(lock)
                        Bank[Alice] Thread 0
Thread 0
            Thread 1
                                                  Thread 1
                                                               Bank[Alice]
Read acct
                                     Acquire(lock)
                                      Read acct
            Read acct
                                                 Acquire(lock)
Increase
                                                 Acquire(lock)
                                      Increase
            Increase
                                      Write back Acquire(lock)
                                                                    10
Write acct
                           10
```

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                       amt):
  amt):
                                            acquire(lock)
       bank[acct] += amt
                                            bank[acct] += amt
                                            release(lock)
                         Bank[Alice] Thread 0
Thread 0
            Thread 1
                                                   Thread 1
                                                                Bank[Alice]
Read acct
                                      Acquire(lock)
                                       Read acct
            Read acct
                                                  Acquire(lock)
Increase
                                                  Acquire(lock)
                                       Increase
            Increase
                                       Write back Acquire(lock)
                                                                    10
Write acct
                            10
                                     Release(lock) Acquire(lock)
                                                                    10
            Write acct →
```

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                       amt):
  amt):
                                            acquire(lock)
       bank[acct] += amt
                                            bank[acct] += amt
                                            release(lock)
                         Bank[Alice] Thread 0
Thread 0
            Thread 1
                                                   Thread 1
                                                                Bank[Alice]
Read acct
                                      Acquire(lock)
                                       Read acct
            Read acct
                                                  Acquire(lock)
Increase
                                                  Acquire(lock)
                                       Increase
            Increase
                                       Write back Acquire(lock)
                                                                     10
Write acct
                            10
                                     Release(lock) Acquire(lock)
                                                                     10
            Write acct →
                                                    Increase
                                                                     10
```

```
Deposit(bank, lock, acct,
  Deposit(bank, acct,
                                       amt):
  amt):
                                            acquire(lock)
       bank[acct] += amt
                                            bank[acct] += amt
                                             release(lock)
                         Bank[Alice] Thread 0
Thread 0
            Thread 1
                                                   Thread 1
                                                                 Bank[Alice]
Read acct
                                      Acquire(lock)
                                       Read acct
            Read acct
                                                  Acquire(lock)
Increase
                                                  Acquire(lock)
                                       Increase
            Increase
                                       Write back
                                                  Acquire(lock)
                                                                     10
Write acct
                            10
                                      Release(lock) Acquire(lock)
                                                                     10
            Write acct
                                                    Increase
                                                                     10
                                                    Write back
                                                                     20
```

amt):	(bank, acc	,		amt):  acqui bank[	oank, <mark>lock</mark> , tre(lock) [acct] += a ase(lock)		,
Thread 0	Thread 1	Ba	ank[Alice][	Thread 0	Thread 1	Ban	k[Alice]
			0				0
Read acct		<b>←</b>	0	Acquire(lock)			0
	Read acct	<b>←</b>	0	Read acct		<b>←</b>	0
Increase			0		Acquire(lock)		0
	Increase		0	Increase	Acquire(lock)		0
Write acct		<b>→</b>	10	Write back	Acquire(lock)	<b>→</b>	10
	Write acct	<b>→</b>	10	Release(lock)	Acquire(lock)		10
					Increase		10
					Write back	<b>→</b>	20
					Release(lock)		20

**Locks:** a data structure that allows only one CPU acquire at a given time

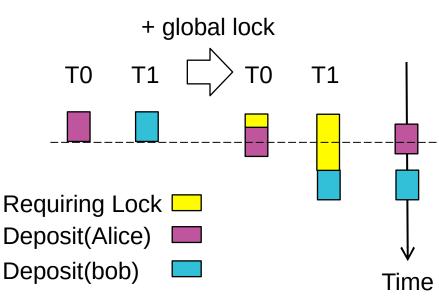
Programs can acquire and release the lock

#### **Global lock:**

 The action must acquire the global lock before executing, and release it after it commits

#### **Drawback: too coarse-grained**

- Only allow one action to run at a time
- Even they don't read/write each other



# Use locking to achieve before-or-after: fine-grained locking

#### **Fine-grained locking:**

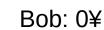
- Each shared data has one lock
- E.g., a lock for Alice & a lock for Bob

#### **Locking rule:**

 The action must acquire the shared data's lock before access it, and releases it after the data access finishes







Lock = [

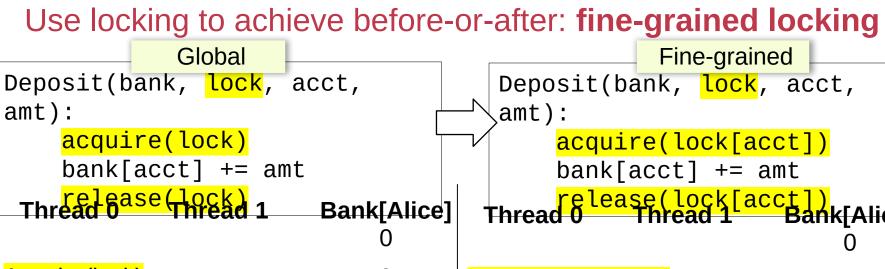
Bank = [ Alice: 0¥





```
Deposit(bank, lock, acct,
amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

```
Deposit(bank, lock, acct,
amt):
    acquire(lock[acct])
    bank[acct] += amt
    release(lock[acct])
```



Thread 0 Thread 1	<b>Bank[A</b>	lice] Thread 0 Thre	ock[acct]) ad 1 Bank  (
Acquire(lock)	0	Acquire(lock[alice])	(
Doad acct	<b>←</b>	Dood cost	

20

Release(lock)

Thread D	Thread 1	Bar	nk[Alice]	Thread 0	Thread 1	<b>Ban</b>	k[Alice]
			0				0
Acquire(lock)			0	Acquire(lock	alice])		0
Read acct		<b>←</b>	0	Read acct	+	_	0
	Acquire(lock)		0		Acquire(lock[ali	ce])	0
Increase	Acquire(lock)		0	Increase	Acquire(lock[alid	ce])	0
Write back	Acquire(lock)	<b>→</b>	10	Write back	Acquire(lock[alid	ce])	10
Release(lock)	Acquire(lock)		10	Release(	Acquire(lock[alid	ce])	10
	Increase		10	lock[alice])	Increase		10
	Write back	<b>-</b>	20	<del></del>	Write back -	<b>→</b>	20

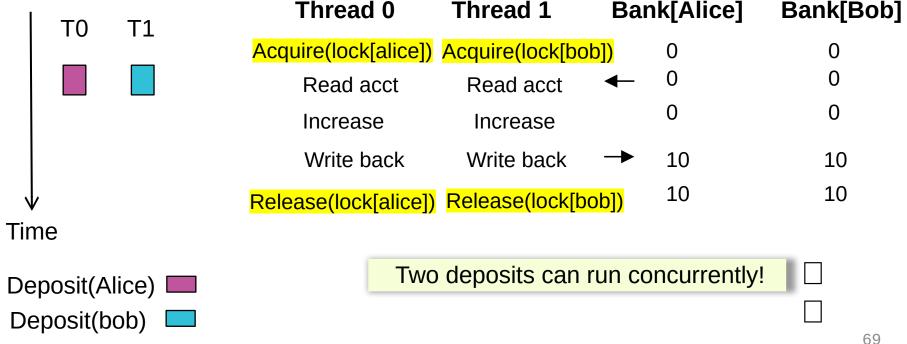
Release(lock[alice])

Thread 0	se (Thread 1	Ва	nk[Alice]	Thread 0	ease(lock[acct]) Thread 1 Bar	ı <mark>k[Alic</mark> ı	е
			0			0	
.cquire(lock)	)		0	Acquire(lock	[ <mark>alice])</mark>	0	
Read acct		<b>←</b>	0	Read acct	←	0	
	Acquire(lock)		0		Acquire(lock[alice])	0	
Increase	Acquire(lock)		0	Increase	Acquire(lock[alice])	0	
Write back	Acquire(lock)	<b>→</b>	10	Write back	Acquire(lock[alice])	10	

## Use locking to achieve before-or-after: **fine-grained locking**

#### Fine-grained locking allows more concurrency

If the deposits operate on different accounts, they can run concurrently



# Use locking to achieve before-or-after: fine-grained locking

#### Fine-grained locking:

- Each data record has one lock
- E.g., a lock for Alice & a lock for Bob

#### Lock acquire rule:

 The action must acquire the account lock before access it, and releases it after the data access finishes





. . .

Bank = [ Alice: 0¥

Bob: 0¥

Lock = [





Question: can fine-grained locking avoids **any** the race conditions?

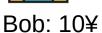
#### More example: multiple records

Transfer + Audit for the bank application





Alice: 10¥







```
Transfer(bank, locks, a, b,
amt):
    Acquire(lock[b])
    bank[b] += amt
    Release(lock[b])
    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
```

```
Audit(bank):
    sum = 0
    for acct in bank:

Acquire(locks[acct])
        sum += bank[acct]

Release(locks[acct])
```

Question: what is the ideal output of Audit?

# Accessing multiple records

Transfer
(alice, bob, 10) Audit()

i	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
$\downarrow$					
Time					

# Accessing multiple records

Transfer
(alice, bob, 10) Audit()

I	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
			10	10	0	
	Acquire(lock[b])	Acquire(lock[b])	10	10	0	

Time

Transfer
(alice, bob, 10) Audit()

I	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = 10	Acquire(lock[b])	10	10	0

Transfer
(alice, bob, 10) Audit()

1	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = $10$	Acquire(lock[b])	10	10	0
	Write(b) = $20$	Acquire(lock[b])	10	20	0

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = $10$	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0



Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
		10	10	0	
Acquire(lock[b])	Acquire(lock[b])	10	10	0	
Read(b) = $10$	Acquire(lock[b])	10	10	0	
Write(b) = 20	Acquire(lock[b])	10	20	0	
Release(lock[b])	Acquire(lock[b])	10	20	0	
	Read(b) = $20$	10	20	20	



Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20



Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = $10$	Acquire(lock[b])	10	10	0
Write(b) = $20$	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a])	10	20	20



Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 30



Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 30
	Release(lock[a])	10	20	30

Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = $10$	Acquire(lock[b])	10	10	0
Write(b) = $20$	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 30
	Release(lock[a])	10	20	30
Acquire(lock[a])		10	20	30

Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = $10$	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = $20$	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 30
	Release(lock[a])	10	20	30
Acquire(lock[a]) Read(a) = 10		10 10	20 20	30 30

Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = 20	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 30
	Release(lock[a])	10	20	30
Acquire(lock[a]) Read(a) = 10		10 10	20 20	30 30
$\bigvee Write(a) = 0$		0	20	30

I	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = $10$	Acquire(lock[b])	10	10	0
	Write(b) = $20$	Acquire(lock[b])	10	20	0
	Release(lock[b])	Acquire(lock[b])	10	20	0
		Read(b) = $20$	10	20	20
		Release(lock[b])	10	20	20
		Acquire(lock[a]) Read(a) = 10	10 10	20 20	20 20
		Release(lock[a])	10	20	30
	<mark>Acquire(lock[a])</mark> Read(a) = 10		10 10	20 20	30 30
	Write(a) = $0$		0	20	30
ne	Release(lock[a])		0	20	30

### **Problem:** accessing multiple records

#### **Question #1:**

- Is the final state of the program correct (i.e., bank accounts) ?

#### Question #2:

– Is the sum of the audit correct?

		Release(lock[a])	10	20	30
	Acquire(lock[a]) Read(a) = 10		10 10	20 20	30 30
$\downarrow$	Write(a) = $0$		0	20	30
Time	Release(lock[a])		0	20	30

### **Problem:** accessing multiple records

#### **Core problem**

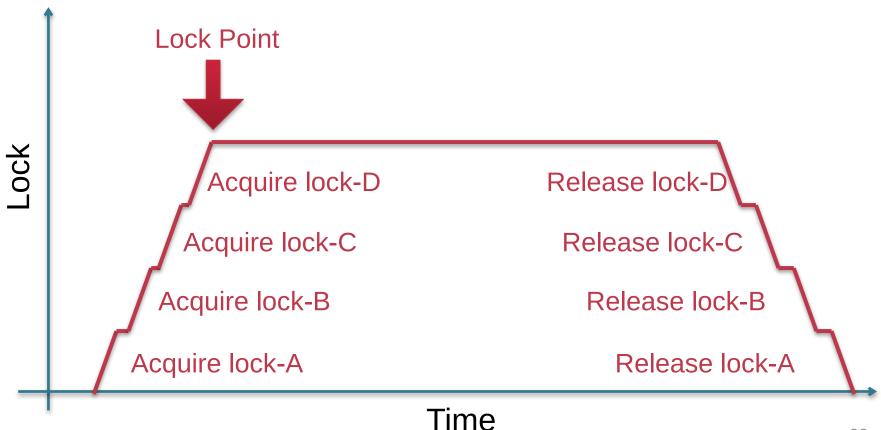
Simple fine-grained locking does not prevent the exposure of intermediate result

#### Insight

- we need to delay the lock release time for each shared data
  - Until all the transaction's updates finish

Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	10
	Read(b) = 20	10	20	10
	Release(lock[b])	10	20	10
	Acquire(lock[a]) Read(a) = 10	10 10	20 20	10 30

### Solution: acquire all locks first, and release them at last



88

#### Fine-grained locking:

- Each shared data has one lock
- E.g., a lock for Alice & a lock for Bob

#### **2PL locking rule:**

The action must acquire the shared data's lock before access it, and releases it after the data access finishes and release it until the action finishes







Bank = [ Alice: 0¥

Lock = [





Bob: 0¥

#### **2PL Lock acquire rule:**

 The action must acquire the shared data's lock before access it, and release it until all the action finishes

```
Fine-grained locking + 2PL
       Fine-grained locking
Transfer(bank, locks, a, b,
                                    Transfer(bank, locks, a, b,
amt):
                                    amt):
    Acquire(lock[b]
                                         Acquire(lock[b]
    bank[b] += amt
                                         bank[b] += amt
    Release(lock[b])
    Acquire(lock[a])
                                         Acquire(lock[a])
    bank[a] -= amt
                                         bank[a] -= amt
   Release(lock[a])
                                         Release(lock[a])
                                         Release(lock[b])
```

Ī	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
			10	10	0	
	Acquire(lock[b])	Acquire(lock[b])	10	10	0	
	Read(b) = $10$	Acquire(lock[b])	10	10	0	
	Write(b) = 20	Acquire(lock[b])	10	20	0	
	Release(lock[b])	Acquire(lock[b])	10	20	0	



l .	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
			10	10	0	
	Acquire(lock[b])	Acquire(lock[b])	10	10	0	
	Read(b) = $10$	Acquire(lock[b])	10	10	0	
	Write(b) = 20	Acquire(lock[b])	10	20	0	
	Acquire(lock[a])	Acquire(lock[b])	10	20	0	



I	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
			10	10	0	
	Acquire(lock[b])	Acquire(lock[b])	10	10	0	
	Read(b) = $10$	Acquire(lock[b])	10	10	0	
	Write(b) = $20$	Acquire(lock[b])	10	20	0	
	Acquire(lock[a])	Acquire(lock[b])	10	20	0	
	Read(a) = $10$	Acquire(lock[b])	10	10	0	



l .	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
			10	10	0	
	Acquire(lock[b])	Acquire(lock[b])	10	10	0	
	Read(b) = $10$	Acquire(lock[b])	10	10	0	
	Write(b) = 20	Acquire(lock[b])	10	20	0	
	Acquire(lock[a])	Acquire(lock[b])	10	20	0	
	Read(a) = $10$	Acquire(lock[b])	10	10	0	
	Write(a) = 0	Acquire(lock[b])	0	20	0	



Transfer
(alice, bob, 10) Audit()

l _	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = $10$	Acquire(lock[b])	10	10	0
	Write(b) = $20$	Acquire(lock[b])	10	20	0
	Acquire(lock[a])	Acquire(lock[b])	10	20	0
	Read(a) = $10$	Acquire(lock[b])	10	10	0
	Write(a) = $0$	Acquire(lock[b])	0	20	0
	Release(lock[b])	Acquire(lock[b])	0	20	0



Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Acquire(lock[a])	Acquire(lock[b])	10	20	0
Read(a) = 10	Acquire(lock[b])	10	10	0
Write(a) = 0	Acquire(lock[b])	0	20	0
Release(lock[b]) Release(lock[a])	Acquire(lock[b])	0	20	0
	Read(b) = 20	0	20	20



Transfer
(alice, bob, 10) Audit()

ı	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = $10$	Acquire(lock[b])	10	10	0
	Write(b) = 20	Acquire(lock[b])	10	20	0
	Acquire(lock[a])	Acquire(lock[b])	10	20	0
	Read(a) = $10$	Acquire(lock[b])	10	10	0
	Write(a) = $0$	Acquire(lock[b])	0	20	0
	Release(lock[b]) Release(lock[a])	Acquire(lock[b])	0	20	0
		Read(b) = 20 Acquire(lock[a])	0 0	20 20	20 20



Transfer
(alice, bob, 10) Audit()

İ	Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
			10	10	0
	Acquire(lock[b])	Acquire(lock[b])	10	10	0
	Read(b) = $10$	Acquire(lock[b])	10	10	0
	Write(b) = $20$	Acquire(lock[b])	10	20	0
	Acquire(lock[a])	Acquire(lock[b])	10	20	0
	Read(a) = $10$	Acquire(lock[b])	10	10	0
	Write(a) = $0$	Acquire(lock[b])	0	20	0
	Release(lock[b]) Release(lock[a])	Acquire(lock[b])	0	20	0
		Read(b) = 20 Acquire(lock[a]) Read(a) = 0	0 0 0	20 20 20	20 20 <b>20</b>

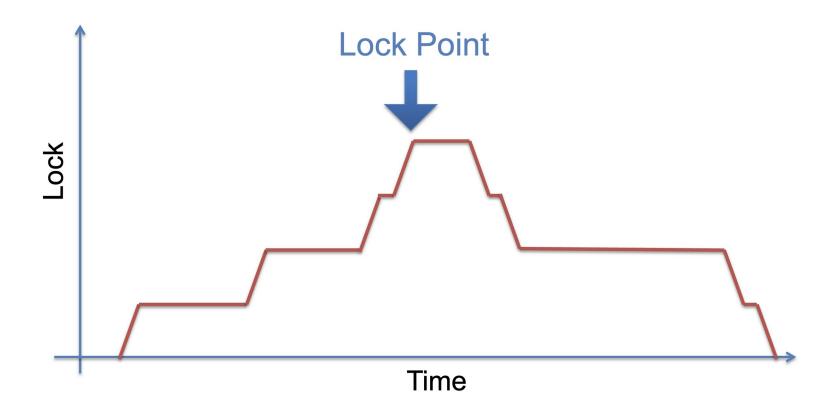


Time

Transfer
(alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = $10$	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Acquire(lock[a])	Acquire(lock[b])	10	20	0
Read(a) = $10$	Acquire(lock[b])	10	10	0
Write(a) = $0$	Acquire(lock[b])	0	20	0
Release(lock[b]) Release(lock[a])	Acquire(lock[b])	0	20	0
	Read(b) = 20 Acquire(lock[a]) Read(a) = 0 Release(lock[a])	0 0 0 0	20 20 20 20	20 20 <b>20</b> <b>20</b>

Audit appears to run after the transfer



### A note on fine-grained locking

#### The number of locks used is made explicit in our previous examples

- Global lock: only 1 lock
- 2PL (ideal): one per record

#### One lock per record record may waste many memory

Especially when there are many records (e.g., records in files)

#### Configurable number of locks

- We can set a constant number of locks for the system
- Only need to ensure the access to the same record will acquire the same lock

#### Question: what are the trade-offs?

### Use locking to achieve before-or-after: **2PL**

#### 2PL can guarantee before or after atomicity with serializability

 Run actions T1, T2, ..., TN concurrently, and have it "appears" as if they ran sequentially (we will prove it later)

#### Before or after atomicity also is usually termed as serializability

### Before or after atomicity

Concurrent actions have the before-or-after property if their effect from the point of view of their invokers is as if the actions occurred either completely before or completely after one another

### Serializability

#### 2PL can guarantee before or after atomicity with serializability

Run actions T1, T2, .., TN concurrently, and have it "appears" as if they ran sequentially (we will prove it later)

What does this mean?

### Serializability

#### 2PL can guarantee before or after atomicity with serializability

Run actions T1, T2, ..., TN concurrently, and have it "appears" as if they ran sequentially (we will prove it later)

#### What does this mean?

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
T2
Action
start
write(x, 20)
write(y, 30)
commit
Action end
```

```
Init: x=0, y=0
T1
                     T2
                                         Possible sequential schedules
begin
                     begin
                                         T1 \rightarrow T2: x=20, y=30
read(x)
                     write(x, 20)
                                         T2 \rightarrow T1: x=20, y=40
tmp = read(y)
                     write(y, 30)
write(y, tmp+10)
                     commit
commit
                                         read(x)
   T2: write(x, 20)
                                    T2: write(x, 20)
   T1: read(x)
   T2: write(y, 30)
                                    T1: tmp = reac(y)
                                    T2: write
                                                    30)
   T1: tmp = read(y)
                                    T1: write(y,
                                                    mp+10)
   T1: write(y, tmp+10)
```

At end: x=20, y=40 At end: x=20, y=10

•	e(y, tmp+10)	commit		, ,
Т2	2: write(x, 2	20)	T1: read(x)	x=0
T1	L: read(x)		<b>T2</b> : write(x, 20)	
Т2	: write(y, 3	30)	<b>T2</b> : write(y, 30)	
T1	L: tmp = read	d(y)	<b>T1:</b> $tmp = read(y)$	) y=30
T1	L: write(y, d	tmp+10)	T1: write(y, tmp	+10)
At	end: x=20,	y=40	At end: x=20, y=	40
In the se	econd schedule	e, the results are c	orrect. But T1 reads	x=0 and

y=30; those reads are not possible in a sequential schedule. Is that ok?

**T2** 

tmp = read(v) write(v, 30)

begin

write(x, 20)

**T1** 

begin

read(x)

Init: x=0, y=0

Possible sequential schedules

 $T1 \rightarrow T2: x=20, y=30$ 

 $T2 \rightarrow T1: x=20, y=40$ 

#### Whether OK depends:

- There are many ways for multiple transactions to "appear" to have been run in sequence; there are different notions of serializability
- Chose a type of serializability depends on applications

```
T2: write(x, 20)
T1: read(x)
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)
T1: tmp = read(y)
T1: write(y, tmp+10)
T1: write(y, tmp+10)
At end: x=20, y=40

T1: read(x)
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)
T1: tmp = read(y)
T1: write(y, tmp+10)
At end: x=20, y=40
```

In the second schedule, the results are correct. But T1 reads x=0 and y=30; those reads are not possible in a sequential schedule. Is that ok?

### Serializability has many types

#### Final-state serializability

 A schedule is final-state serializable if its final written state is equivalent to that of some serial schedule

Conflict serializability Most widely used

View serializability

#### Final-state serializability

T1: read(x) x=0

T2: write(x, 20)

**T2**: write(y, 30)

T1: tmp = read(y) y=30

**T1**: write(y, tmp+10)

At end: x=20, y=40

# **Conflict Serializability**

## **Conflict Serializability**

#### Two operations conflict if:

- 1. they operate on the same data object, and
- 2. at least one of them is write, and
- 3. they belong to different transactions

#### **Conflict serializability**

 A schedule is conflict serializable if the order of its conflicts (the order in which the conflicting operations occur) is the same as the order of conflicts in some sequential schedule

```
T1 T2 Possible sequential schedules T1.1read(x) T2.1write(x, 20) T1 -> T2: x=20, y=30 T1.2tmp = read(y) T2.2write(y, 30) T2 -> T1: x=20, y=40 T1.3write(y, tmp+10) commit commit
```

```
on x

T1.1 read(x) and T2.1 write(x,20)

T1.2 tmp = read(y) and T2.2 write(y,30)

T1.3 write(y, tmp+10) and T2.2 write(y,30)
```

## **Conflict Graph**

#### **Conflict Graph**

- Nodes are transactions, edges are directed
- Edge between T<sub>i</sub> and T<sub>i</sub> if and only if:
  - 1. T<sub>i</sub> and T<sub>i</sub> have a conflict between them, and
  - 2. the first step in the conflict occurs in T<sub>i</sub>

#### A schedule is conflict serializable if and only if:

It has an acyclic conflict graph

```
T1T2beginbeginSchedulesread(x)write(x, 20)T1 -> T2: x=20, y=30tmp = read(y)write(y, 30)T2 -> T1: x=20, y=40write(y, tmp+10)commit
```

```
T1T2beginbeginPossible sequential<br/>schedulesread(x)write(x, 20)T1 -> T2: x=20, y=30tmp = read(y)write(y, 30)T2 -> T1: x=20, y=40write(y, tmp+10)commit
```

```
T2.1 write(x,20) -> T1.1 read(x)

T2.2 write(y,30) -> T1.2 tmp = read(y)

tmp+10) -> T1.3 write(y,

(T2 -> T1)
```

## **Conflict order for sequential schedules**

T2.1, T1.1 T1.1 -> T2.1 T2.1 -> T1.1 T2.2, T1.2  $T1.2 \rightarrow T2.2$  or  $T2.2 \rightarrow T1.2$ T2.2, T1.3 T1.3 -> T2.2 T2.2 -> T1.3 (T2 -> T1)(T1 -> T2)

**T2**: write(x, 20) T1: read(x)

**T2**: write(y, 30)

T1: tmp = read(y)

**T1**: write(y, tmp+10)

T2.1 -> T1.1 T2.2 -> T1.2

 $T2.2 \rightarrow T1.3$ 

T1. read(x)**T2:** write(x, 2

T2: write 30) T1: tmp = read(v)

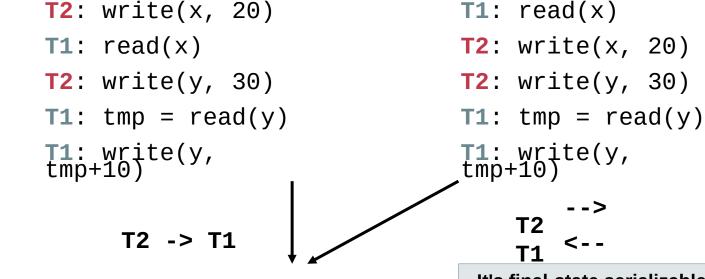
T1. write(y, tmp+1

T1.1 -> T2.1 T2.2 -> T1.2

T2.2 -> T1.3

## **Conflict order for sequential schedules**

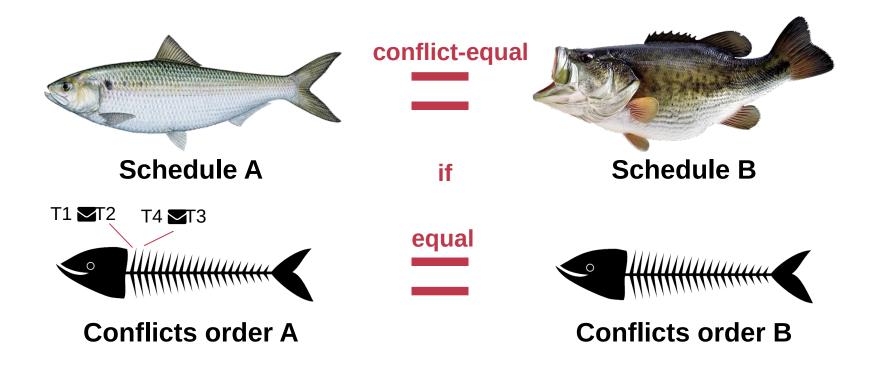
T2.1, T1.1 T1.1 -> T2.1 T2.1 -> T1.1 T2.2, T1.2  $T1.2 \rightarrow T2.2$  or  $T2.2 \rightarrow T1.2$ T2.2, T1.3 T2.2 -> T1.3 T1.3 -> T2.2 (T1 -> T2)(T2 -> T1)



Both at end: x=20, y=40

It's final-state serializable, not conflict serializable

## **Conflict Equivalence**

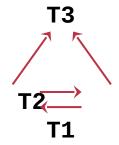


If conflicts-order-A equals to conflicts-order-B, then schedule-A conflict-equals to schedule-B

## **View Serializability**

```
T1 T2 T3
read(x)
write(x)
write(x)
write(x)
```

#### **Conflict** graph



#### **Cyclic -> Not conflict serializable**

But compare it to running T1 then T2 then T3 (serially)

- Final-state is fine
- Intermediate reads are fine

Question: why shouldn't we allow this schedule?

## **View Serializability**

#### Informal definition

 A schedule is view serializable if the final written state as well as intermediate reads are the same as in some serial schedule

#### Formally, for those interested

- Two schedules S and S' are view equivalent if:
  - If T<sub>i</sub> in S reads an initial value for X, so does T<sub>i</sub> in S'
  - If T<sub>i</sub> in S reads the value written by T<sub>j</sub> in S for some X, so does
     T<sub>i</sub> in S'
  - If T<sub>i</sub> in S does the final write to X, so does T<sub>i</sub> in S'

A schedule is view serializable if it is view equivalent to some serial schedule

## Question

Why conflict serializability when it seems too strict?

Why not focus on view serializability?

Final-state Serializability

Care the final state only

View

→ Serializability

Care the final state as well as intermediate read

Conflict

→ Serializability

Care the final state as well as all the data dependency

## Conflict Serializability VS. View Serializability

#### Conflict serializability is easy to test

- i.e., check whether a graph is acyclic
- View serializability is hard to test (likely NP-hard)

#### Conflict serializable schedules are easy to generate

Using concurrency control protocols. E.g., 2PL (two-phase-locking)

#### Conflict serializable schedules are also view serializable

 No easy way to generate view schedules that allows for ones like the previous example

## Use locking to achieve before-or-after: 2PL

#### 2PL can guarantee before or after atomicity

 Run actions T1, T2, ..., TN concurrently, and have it "appears" as if they ran sequentially

#### **2PL locking rule:**

 The action must acquire the shared data's lock before access it and release it until the action finishes

#### Question

- Which serializability does 2PL guarantee?
- Conflict serializability!

#### **Proof of 2PL**

Suppose 2PL does **not** generate conflict serializable schedule

Suppose the conflict graph produced by an execution of 2PL has a cycle, which without loss of generality, is:

Let the shared variable (the one that causes the conflict) between **T\_i** and **T\_{i+1**} be represented by **x\_i**.

#### T1 violates 2PL!

```
T1 acquires x1.lock
T1 releases x1.lock
...
T1 acquires x k.lock
```

```
T1 and T2 conflict on x1
T2 and T3 conflict on x2
...
Tk and T1 conflict on x_k
```



T1 acquires x1.lock
T2 acquires x1.lock and x2.lock
T3 acquires x2.lock and x3.lock
...

Tk acquires x\_{k-1}.lock and x\_k.lock
T1 acquires x\_k.lock



T1 acquires x1.lock

T1 releases x1.lock

T2 acquires x1.lock and x2.lock

...

Tk acquires x\_{k-1}.lock and x\_k.lock

T1 acquires x\_k.lock

# Serializability simplifies enforcing concurrent correctness / data consistency

## **Correctness (app-semantic consistency)**

#### **Correctness is hard for computer system to enforce**

- It is at the application level, with application specific semantic
- E.g., developer can write a buggy program, even no concurrency (single-threaded)

```
Transfer(bank, a, b,
amt):
    bank[b] += amt
    bank[a] -= amt
```

```
Buggy

Transfer(bank, a, b, amt):

bank[b] += amt * 2
bank[a] -= amt
```

#### To ensure concurrent correctness, we need assumptions

i.e., The single-threaded implementation of actions must be correct

## **Correctness and serializability**

#### Assumption: programmers are pro at writing single-thread programs

 Specially, in a single-thread context, can move data from a consistent state () to another consistent state ()

## Then, if the execution of actions guarantee serializability, then the final state of concurrent execution is consistent

i.e., the concurrent execution can reduce to If C0 is consistent, then Cn must be consistent

## The deadlock issues of 2PL

#### **Deadlock**

#### Two-phase lock is pessimistic:

- Before proceed, each TX must wait for conflicting TX to release the lock
- What can happen if two TX are waiting for each other?

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Acquire(lock[a])	Acquire(lock[b])	10	20	0
Read(a) = 10	Acquire(lock[b])	10	10	0
Write(a) = $0$	Acquire(lock[b])	0	20	0
Release(lock[b])	Acquire(lock[b])	0	20	0
	Read(b) = 20	0	20	20
	Acquire(lock[a])	0	20	20
	Read(a) = 0	0	20	20
	Release(lock[a])	0	20	20

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
		10	10	0	
Acquire(lock[b])	Acquire(lock[a])	10	10	0	

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
		10	10	0	
Acquire(lock[b])	Acquire(lock[a])	10	10	0	
Read(b) = $10$	Read(a) = 10	10	10	10	

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum	
		10	10	0	
Acquire(lock[b])	Acquire(lock[a])	10	10	0	
Read(b) = $10$	Read(a) = 10	10	10	10	
Acquire(lock[a])	Acquire(lock[b])				

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[a])	10	10	0
Read(b) = $10$	Read(a) = $10$	10	10	10
Acquire(lock[a])	Acquire(lock[b])			
Acquire(lock[a])	Acquire(lock[b])			

Question: can thread 0 or thread 1 finishes the execution?

## **Resolving deadlock**

#### 1. Acquire locks in a pre-defined order

Not support general TX: TX must know the read/write sets before execution

#### 2. Detect deadlock by calculating the conflict graph

- If there is a cycle, then there must be a deadlock
- Abort one TX to break the cycle
- High cost for detection

#### 3. Using heuristics (e.g., timestamp) to pre-abort the TXs

May have false positive, or live locks