

Serializability, OCC & Transaction

IPADS, Shanghai Jiao Tong University

<https://www.sjtu.edu.cn>

Review: 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	Bank[Alice]
		0
Read acct		← 0
	Read acct	← 0
Increase		0
	Increase	0
Write acct		→ 10
	Write acct	→ 10

Thread 0	Thread 1	Bank[Alice]
		0
Read acct		← 0
Increase		0
Write acct		→ 10
	Read acct	← 10
	Increase	10
	Write acct	→ 20

Review: **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**

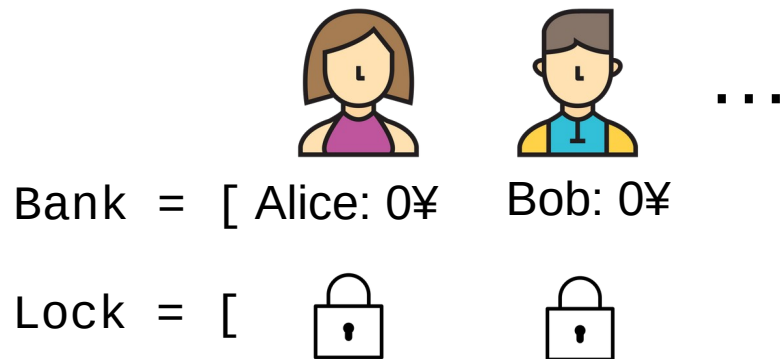
Review: Two-phase locking (2PL)

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



Review: Use 2PL to achieve before-or-after

2PL can guarantee before or after atomicity with serializability for TXs

- 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**

Review: Serializability

2PL can guarantee before or after atomicity with serializability for TXs

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



TX marks the atomicity scope

What does this mean?

- A set of reads and writes that need be executed as an atomic unit

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

T1

begin

read(x)

tmp = read(y)

write(y, tmp+10)

commit

T2

begin

write(x, 20)

write(y, 30)

commit

Possible sequential
schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

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 (remind of the race condition ◀◀):

1. they operate on the same data object, and
2. at least one of them is write, and
3. they belong to different transactions (a single TX is assumed to be executed sequentially)

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

begin

T1.1 read(x)

T1.2 tmp = read(y)

T1.3 write(y, tmp+10)

commit

T2

begin

T2.1 write(x, 20)

T2.2 write(y, 30)

commit

Init: x=0, y=0

Possible sequential
schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

Conflicts

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

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

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

Use Conflict Graph to determine the sequential schedule

Conflict Graph

- Nodes are transactions, edges are **directed**
- Edge between T_i and T_j if and only if:
 1. T_i and T_j have a conflict between them, and
 2. the first step in the conflict occurs in T_i

A schedule is conflict serializable if and only if:

- It has an **acyclic** conflict graph

T1

begin

read(x)

tmp = read(y)

write(y, tmp+10)

commit

T2

begin

write(x, 20)

write(y, 30)

commit

**Possible sequential
schedules**

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

Conflicts

T1: read(x)

T1: tmp = read(y)

T2: write(x, 20)

T1: write(y, tmp+10)

T2: write(y, 30)

At end: x=20, y=30

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

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

T1.3 write(y, tmp+10) -> T2.2 write(y, 30)

(T1 -> T2)

T1

begin

read(x)

tmp = read(y)

write(y, tmp+10)

commit

T2

begin

write(x, 20)

write(y, 30)

commit

**Possible sequential
schedules**

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

T2: write(x, 20)

T1: read(x)

T2: write(y, 30)

T1: tmp = read(y)

T1: write(y, tmp+10)

At end: x=20, y=40

Conflicts

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

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

T2.2 write(y,30) -> **T1.3** write(y,

(T2 -> T1)

Conflicts

T2.1, T1.1

T2.2, T1.2

T2.2, T1.3

Conflict order for sequential schedules

T1.1 -> T2.1

T1.2 -> T2.2 or

T1.3 -> T2.2

(T1 -> T2)

T2.1 -> T1.1

T2.2 -> T1.2

T2.2 -> T1.3

(T2 -> T1)

T2: write(x, 20)

T1: read(x)

T2: write(y, 30)

T1: tmp = read(y)

T1: write(y, tmp+10)

~~T1: read(x)~~

~~T2: write(x, 20)~~

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

~~T1: tmp = read(y)~~

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

T2.1 -> T1.1

T2.2 -> T1.2

T2.2 -> T1.3

T1.1 -> T2.1

T2.2 -> T1.2

T2.2 -> T1.3

Conflicts

T2.1, T1.1

T2.2, T1.2

T2.2, T1.3

Conflict order for sequential schedules

T1.1 -> T2.1

T1.2 -> T2.2

T1.3 -> T2.2

(T1 -> T2)

T2.1 -> T1.1

or T2.2 -> T1.2

T2.2 -> T1.3

(T2 -> T1)

T2: write(x, 20)

T1: read(x)

T2: write(y, 30)

T1: tmp = read(y)

T1: write(y,
tmp+10)

T2 -> T1

Both at end: x=20, y=40

T1: read(x)

T2: write(x, 20)

T2: write(y, 30)

T1: tmp = read(y)

T1: write(y,
tmp+10)

-->
T2
T1 <--

It's final-state serializable,
not conflict serializable

Conflict Equivalence

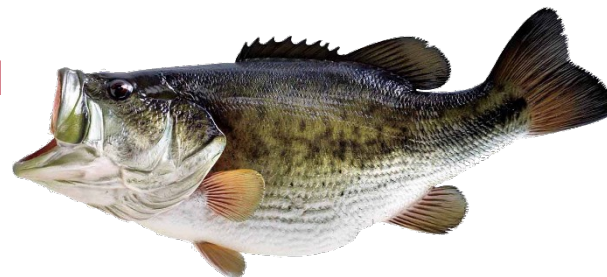


Schedule A

conflict-equal



if



Schedule B

T1 ☑ T2 T4 ☑ T3



Conflicts order A

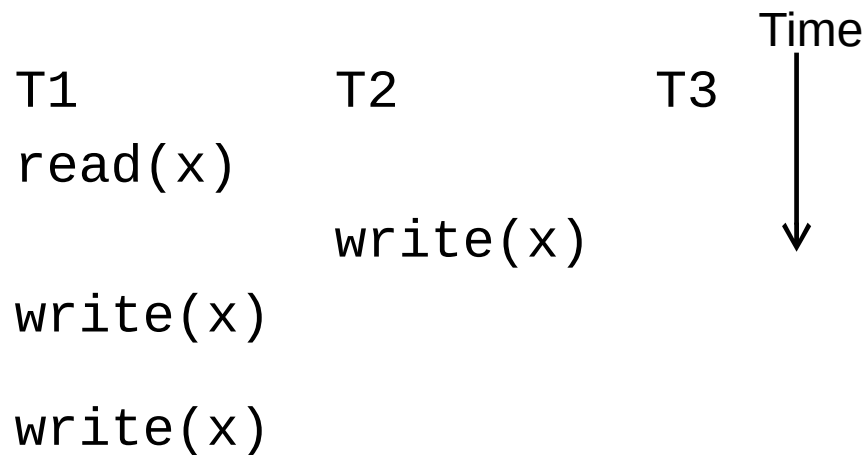
equal



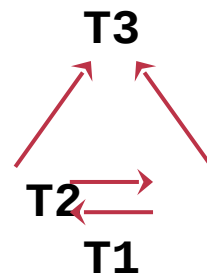
Conflicts order B

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

View Serializability



Conflict graph



Cyclic -> Not **conflict serializable**

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

- Final-state is fine (T3)
- Intermediate reads are fine (T1 reads the initial value of x)

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_i in **S** reads an initial value for X , so does T_i in **S'**
 - If T_i in **S** reads the value written by T_j in **S** for some X , so does T_i in **S'**
 - If T_i in **S** does the final write to X , so does T_i in **S'**

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

Question

Why conflict serializability, given that it seems **too strict**?

Why not focus on view serializability? (Allow more interleaving, & is still correct)

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

In most computer science domains, serializability \sim conflict serializability

Use 2PL to achieve before-or-after

2PL can guarantee before or after atomicity

- Run actions T1, T2, .., TN concurrently, and have it "appears" as if they ran sequentially, i.e., **conflict serializability**

2PL locking rule:

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

Question

- Why 2PL can generate 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:

T1 --> T2 --> ... --> Tk --> T1

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

Question: Is 2PL sufficient to guarantee serializability?

Depends on whether the lock is properly held.

Case study: the software department's salary system

A system to manage the salary of the employers of the software department

- Data to store: the employers

Application's operations to the system

- Update the salaries of **a group of people**
- Print the status of a group of employers satisfying a condition
- Add new employers
- Etc.

Case study: the software department's salary system

Operations

- Update the salaries of **a group of people**
- Print the status of a group of employers satisfying a condition
- Add new employers

Detailed system implementations

- For simplicity, we use a simple linked list to store all the employers
- The list is stored in-memory
 - Fault tolerance can be realized with logging

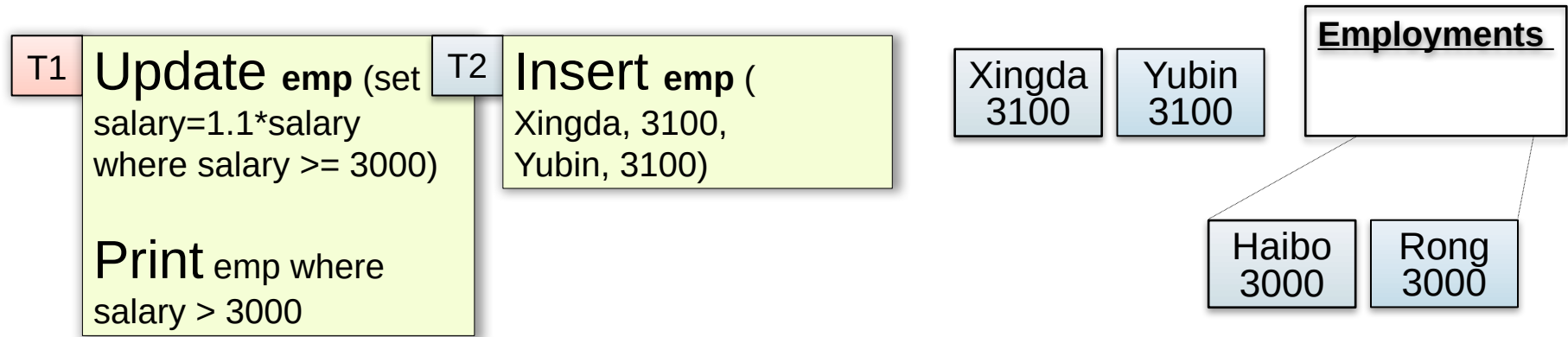
Implementing the software department's salary system

Data layout

- In-memory vector to storing all the employers
- Each employer has a fine-grained lock to realize 2PL

```
database = std::List<Employer  
*>;
```

```
Struct Employer {  
    Lock lock;  
    string name;  
    double salary;  
}
```



Problem

- If T1 sees Xingda & Yubin, it must have updated it
- But T1 misses the insertions on the first updates

Phantom Problem

Issue

- Only lock the things you touch is insufficient!

We must guarantee non-existence of any new ones!

Two operations conflict ☒ they access the same **data item** and at least one is a write

➤ e.g., object, row, list, entire table, ...

Solutions

- Predicate locking
- Range locks in a B-tree index (e.g., assumes an index on **salary**), or lock entire table
- Sometimes being ignored in practice

Remaining issue: deadlock

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?

Fine-grained locking + 2PL

```
Transfer(bank, locks, a, b,  
amt):
```

```
    Acquire(lock[b])  
    bank[b] += amt
```

```
    Acquire(lock[a])  
    bank[a] -= amt  
    Release(lock[a])  
    Release(lock[b])
```

```
Audit(bank):
```

```
    sum = 0  
    for acct in bank:
```

```
        Acquire(locks[acct])  
        sum += bank[acct]  
    print(sum)  
    ReleaseAllLocks(...)
```

Deadlock: what if Thread 1 first acquires lock[a]?

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

Deadlock: what if Thread 1 first acquires lock[a]?

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

Deadlock: what if Thread 1 first acquires lock[a]?

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

Deadlock: what if Thread 1 first acquires lock[a]?

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])			

Deadlock: what if Thread 1 first acquires lock[a]?

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?

Methods for resolving the deadlock

1. Acquire locks in a pre-defined order

Prevention

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

2. Detect deadlock by calculating the conflict graph

Detection

- 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

Retry

- May have false positive, or live locks

Issues of 2PL

Deadlock

- Typically, hard to prevent and detect in practice
- Locking overhead (overhead to acquire the lock + wait for the lock release)

What are the root cause of the above issues of 2PL?

- It pessimistically avoids race conditions

**Can we run transaction
optimistically?**

Optimistic concurrency control

Executing TXs optimistically **w/o acquiring the lock**

Checks the results of TX before it commits

- If violate serializability, then **aborts & retries**

First proposed in 1981, widely used today



OCC Executes a Transaction in 3 Phases

Phase 1: **Concurrent local processing**

- Reads data into a read set
- Buffers writes into a write set

Phase 2: **Validation serializability in critical section**

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: **Commit the results in critical section or abort**

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

OCC Executes a Transaction in 3 Phases

Before phase one, TX needs to allocate the execution context

- Read-set & write set

```
...  
tx.begin();  
...  
tx.read(A)  
...  
tx.commit();  
...
```

Init read_set
Init write_set

OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

```
...  
tx.begin();  
...  
tx.read(A)  
...  
tx.commit();  
...
```

`val_a = read(A)`
`read_set.add(val_a)`

This step should be atomic!

OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

What about a second read?

- Read from the read-set!
- Why? Need to provide repeated read!

```
...  
tx.begin();  
...
```

```
...
```

```
tx.read(A)
```

```
tx.read(A)
```

```
tx.commit();  
...
```

```
if A in read_set:  
    return read_set[A]
```

OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

```
...  
tx.begin();  
...  
tx.read(A)  
tx.write(A)  
tx.commit();  
...
```

} Write_set[A] = ..
if A in read_set:
 read_set[A] = ..

OCC Executes a Transaction in 3 Phases

Phase 2:

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

```
...  
tx.begin();  
...  
tx.read(A)  
...  
tx.commit();  
...
```

```
for d in read_set:  
    if d has changed:  
        abort()
```

OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

```
...  
tx.begin();  
...  
tx.read(A)  
...  
tx.commit();  
...
```

```
for d in read-set:  
    if d has changed:  
        abort()  
for d in write_set:  
    write(d)
```

OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

```
...  
tx.begin();  
...  
tx.read(A)  
...  
tx.commit();  
...
```

Phase 2 & 3 should execute in a critical section

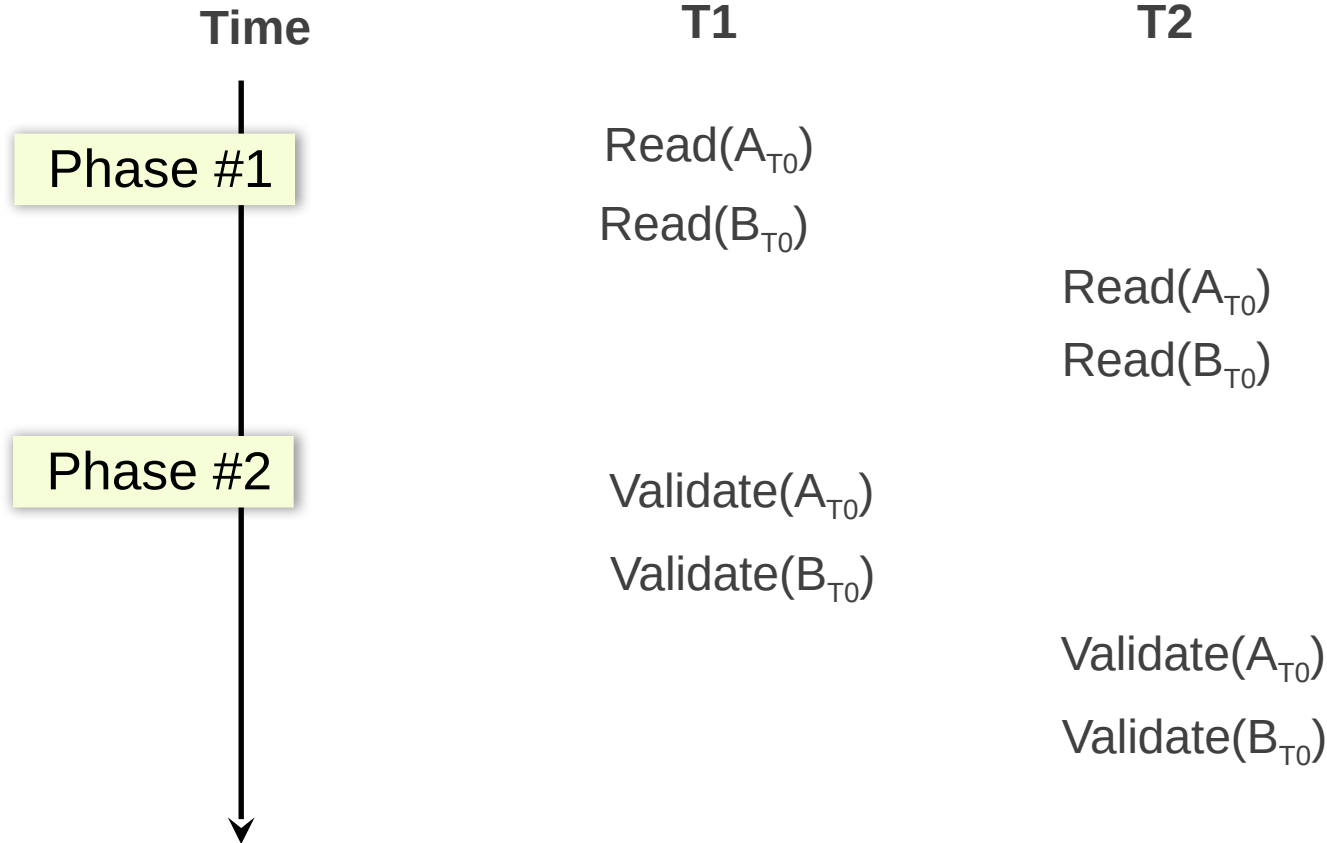
- Otherwise, what if a value has changed during validation?

Critical section

```
for d in read_set:  
    if d has changed:  
        abort()  
for d in write_set:  
    write(d)
```

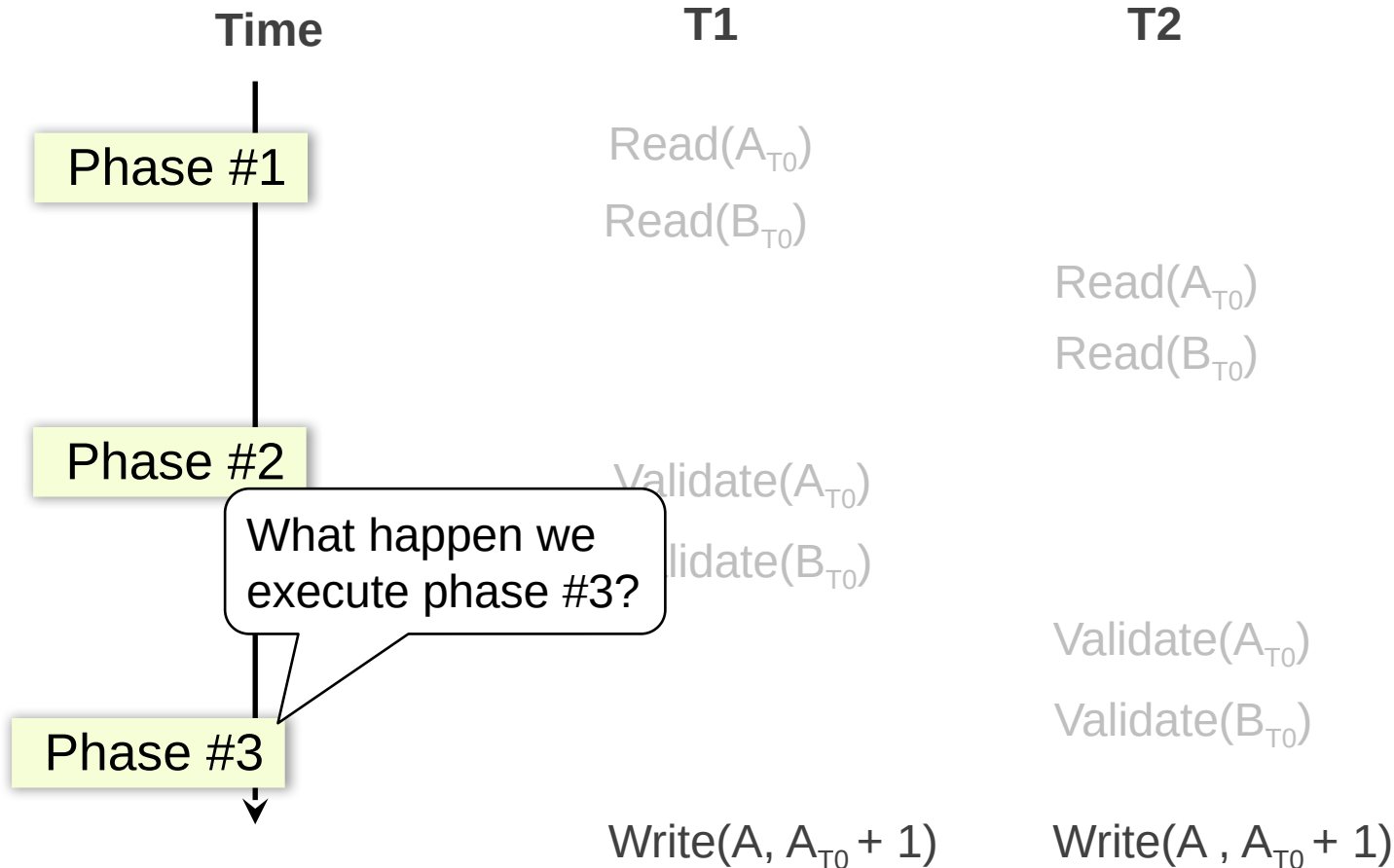
Why phase 2 & phase 3 must be in critical section?

Both TX:
A += 1
B += 1



Why phase 2 & phase 3 must be in critical section?

Both TX:
A += 1
B += 1



How to implement the critical section for phase 2 & 3?

Global lock may satisfy

- The phase 2 & 3 are typically short
 - No TX logic is executed, only the validation

```
def validate_and_commit() // phase 2 & 3
    global_lock.lock()
    for d in read-set:
        if d has changed:
            abort()
    for d in write-set:
        write(d)
    global_lock.unlock()
```

How to implement the critical section for phase 2 & 3?

Using **two-phase locking**

- Allow more concurrency
- What about the problem of deadlock?

```
def validate_and_commit() // phase 2 & 3
    for d in read-set:
        d.lock()
        if d has changed:
            abort() // abort will release all the held
lock
    for d in write-set:
        d.lock()
        write(d)
    // release the locks
    ...
```

How to implement the critical section for phase 2 & 3?

Using two-phase locking

- Allow more concurrency
- Use sort to avoid deadlock; b

Question: do we need
to lock the read-set?

locking overhead as 2PL

```
def validate_and_commit(): // phase 2 & 3
    for d in sorted(read-set + write-set):
        d.lock()
    for d in read-set:
        if d has changed:
            abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

How to implement the critical section for phase 2 & 3?

Observation (for the read-set):

- If the validation passes, then it "appears" that a lock is held during

```
def validate_and_commit() // phase 2 & 3
    for d in sorted(read-set + write-set):
        d.lock()
    for d in read-set:
        if d has changed:
            abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

How to implement the critical section for phase 2 & 3?

Observation (for the read-set):

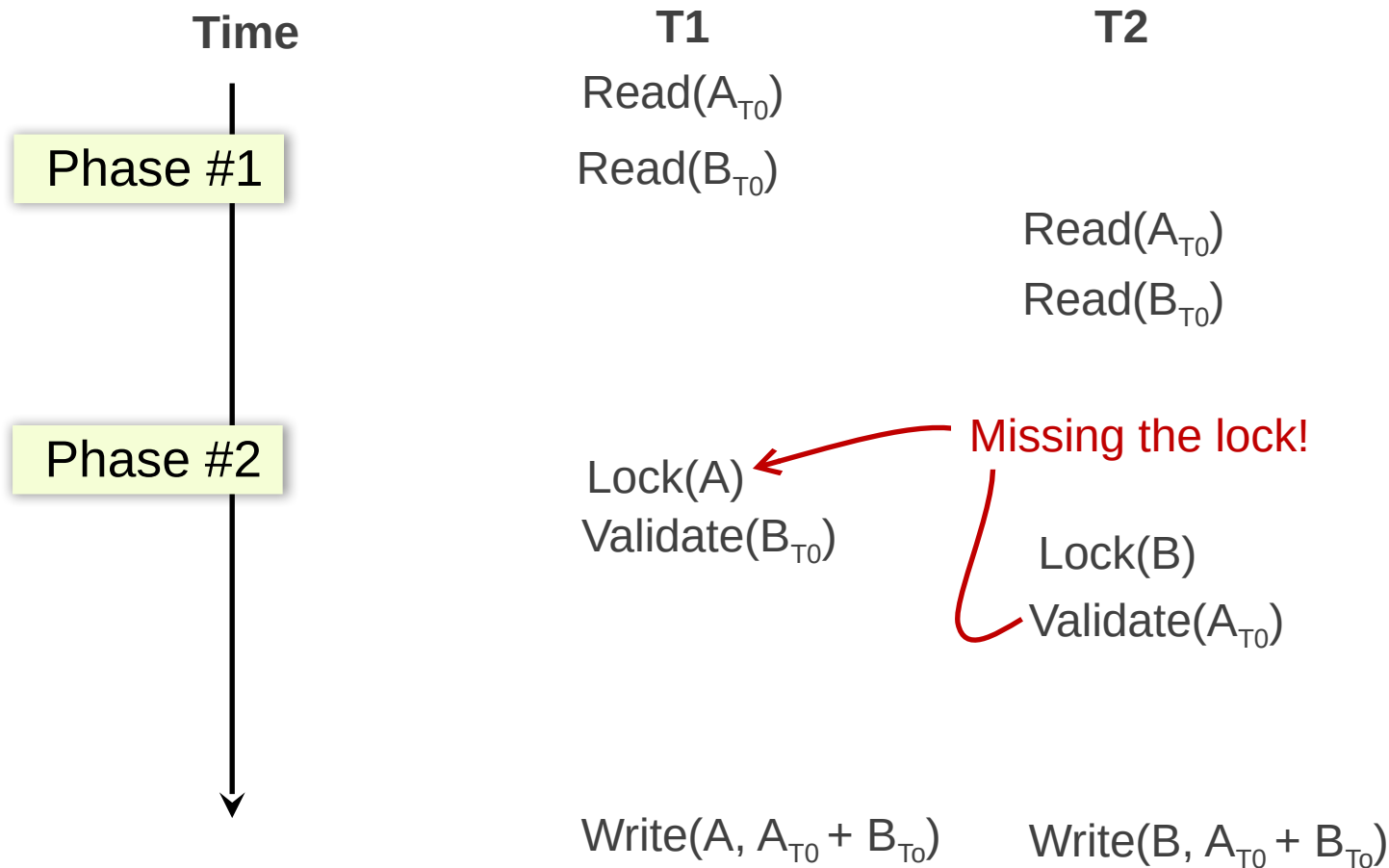
- If the validation passes, then it "appears" that a lock is held during
- Question: is the following implementation correct?

```
def validate_and_commit() // phase 2 & 3
    for d in sorted(write-set):
        d.lock()
    for d in read-set:
        if d has changed:
            abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

Problem: read-write conflict

T1:
A += B

T2:
B += A



How to implement the critical section for phase 2 & 3?

Use two-phase locking + read validation

- Only lock the write set, after that,
- Check the read set has not changed & locked

```
def validate_and_commit() // phase 2 & 3 with before-or-  
after  
    for d in sorted(write-set):  
        d.lock()  
    for d in read-set:  
        if d has changed or d has been locked:  
            abort()  
    for d in write-set:  
        write(d)  
    // release the locks  
    ...
```


OCC: An Example

Tuple **A** in T1's read set has been changed by T2, thus T1 must be **aborted**.

T1: $B = A + B;$

Thread 1

BEGIN

R(A);

R(B);

W(B);

Abort

Thread 2

T2: $A = A * 1.1;$

BEGIN

R(A);

W(A);

Commit

T2's read set remains unchanged, thus T2 is allowed to commit

T1's read set:

A (100)

B (100)

T1's write set:

B (200)

Thread-local memory

T2's read set:

A (100)

T2's write set:

A (110)

Thread-local memory

Time

Database storage

ID	Value
A	110
B	100

OCC Advantages

Phase 1: **Concurrent local processing**

- Reads data into a read set
- Buffers writes into a write set

Operates in private workspace;
rare inter-thread
synchronization (**optimistic**)

Phase 2: Validation in critical section

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: Commit the results in critical section or abort

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

OCC Advantages

Phase 1: Concurrent local processing

- Reads data into a read set
- Buffers writes into a write set

Needs synchronization (e.g., with lock), but usually very short at low contention

Phase 2: **Validation in critical section**

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: **Commit the results in critical section or abort**

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

OCC Advantages

Use reads more than writes (for each read-only data access)

- OCC (in the optimal case, i.e., no abort)
 - 1 read to read the data value
 - 1 read to validate whether the value has been changed or not (as well as locked)
- 2PL
 - 1 operation to acquire the lock (typically an atomic CAS)
 - 1 read to read the data value
 - 1 write to release the lock
 - A single CPU write is atomic, no need to do the atomic CAS

Locking is costly especially compared to reads

Locking preliminary

Many implementation exists to achieve the best performance & scalability

- Still an active topic in system research

We will focus on the basic design

- The details will be left to the OS class

Basic impl: using a flag to indicate whether the log is hold

- Acquire: read the current flag, if not locked, then set it as locked

Interestingly, acquiring the lock itself also requires before-of-after

- Both threads will read the current lock status
- If the status is not locked, update it to locked
 - What if two threads concurrently read an unlocked state?

Locking preliminary: atomic compare and swap

Hardware primitive to guarantee before-or-after atomicity

- Compare-and-swap (on SPARC)
- Compare-and-exchange (on x86)
- **Lock prefix** to ensure an instruction is atomically executed on a memory address

```
1 int CompareAndSwap(int *ptr, int expected, int new) {  
2     int actual = *ptr;  
3     if (actual == expected)  
4         *ptr = new;  
5     return actual;  
6 } // hardware ensures atomicity
```

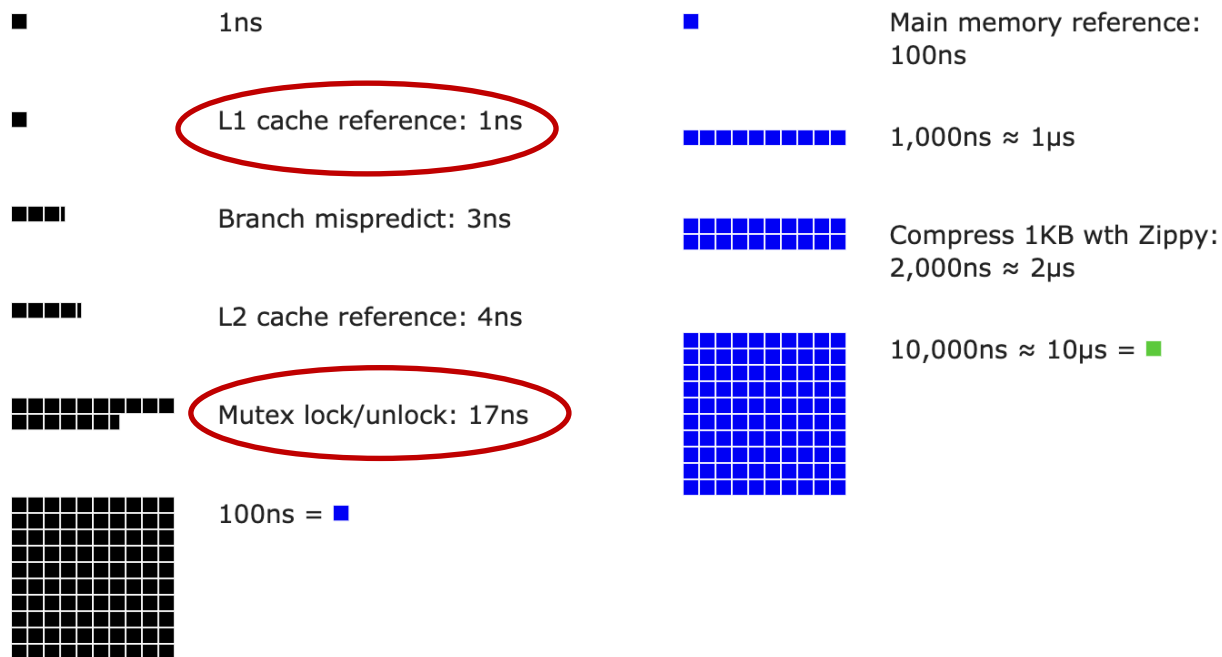
Locking (Spin Lock) using Compare-and-swap

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Atomic operations are costly for spin lock

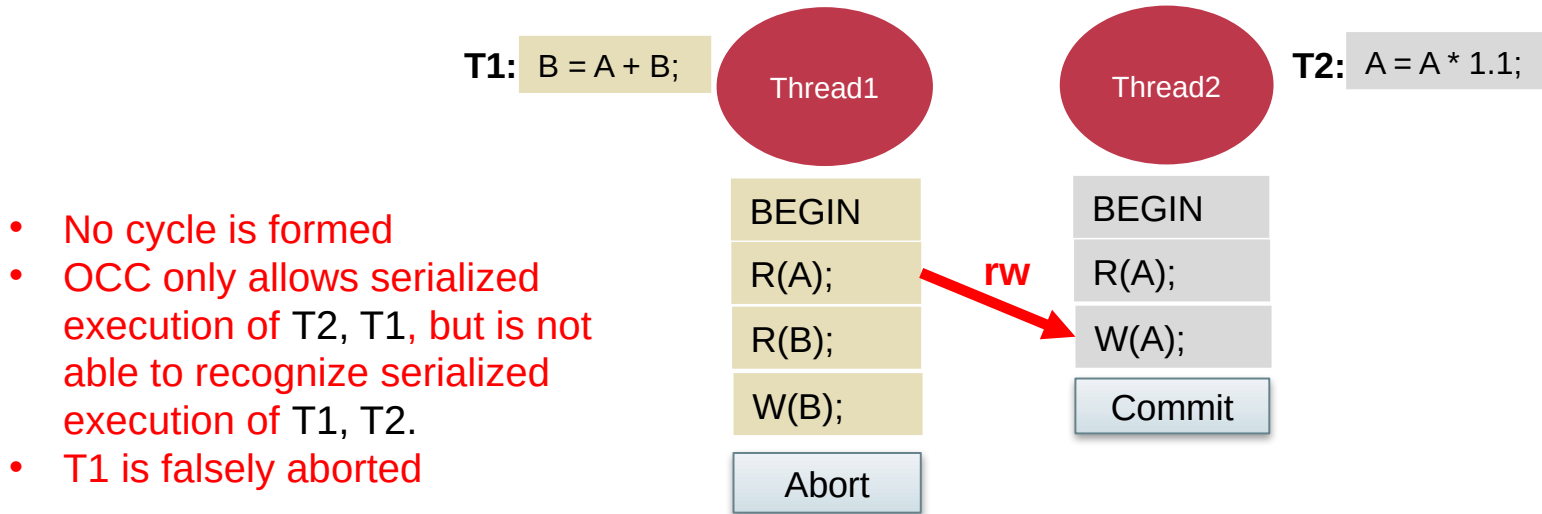
Latency Numbers Every Programmer Should Know

2020



OCC's Problem: False Aborts

Some transactions aborted by OCC could have been allowed to commit without causing an unserializable schedule



OCC's Problem: False Aborts

Some transactions aborted by OCC could have been allowed to commit without causing an unserializable schedule

Especially for TXs with a lot of reads

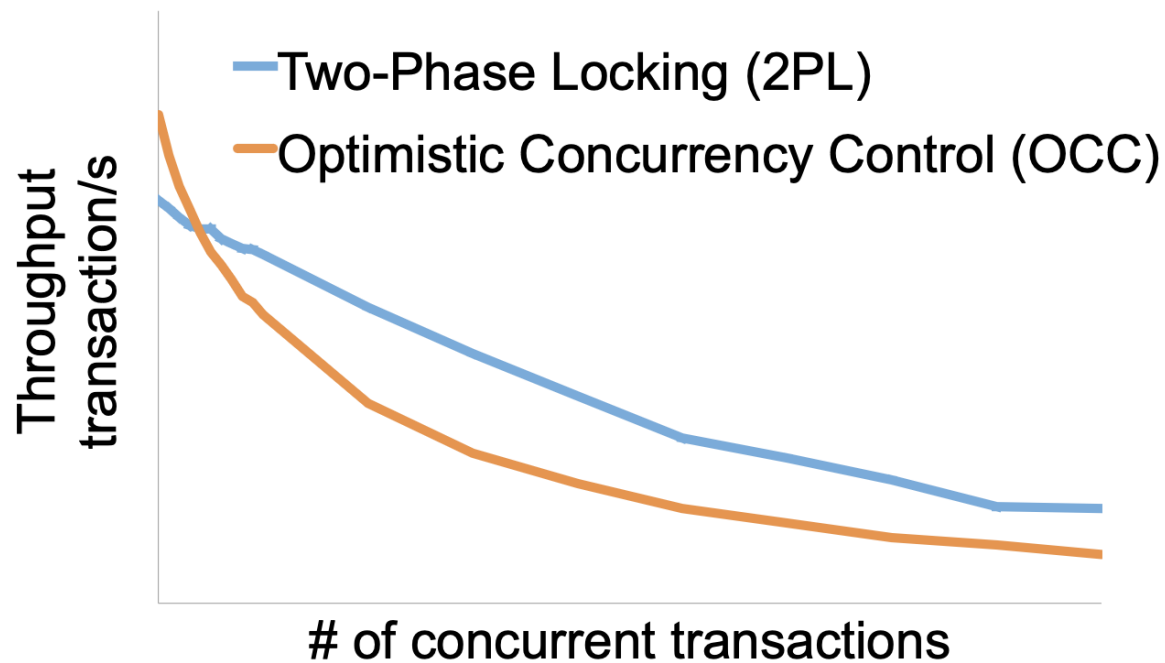
- E.g., read a lot of items → large read set → easier to abort
- May abort *even under low-contention*

OCC's Problem: livelock

Under high contention, OCC may continuously abort

- Especially with large read/write sets, or long execution time
- Transaction is executing, but no progress!

2PL vs. OCC: in a nutshell



5

OCC and hardware transactional memory

Hardware Transactional Memory

A new CPU feature for writing concurrent programs

CPU guarantees the before-or-after atomicity of memory reads/writes

- i.e., no race conditions & no need for 2PL & (Software-implemented) OCC!

Intel supports HTM named as RTM

- Restricted Transactional Memory
- First released in Haswell processor

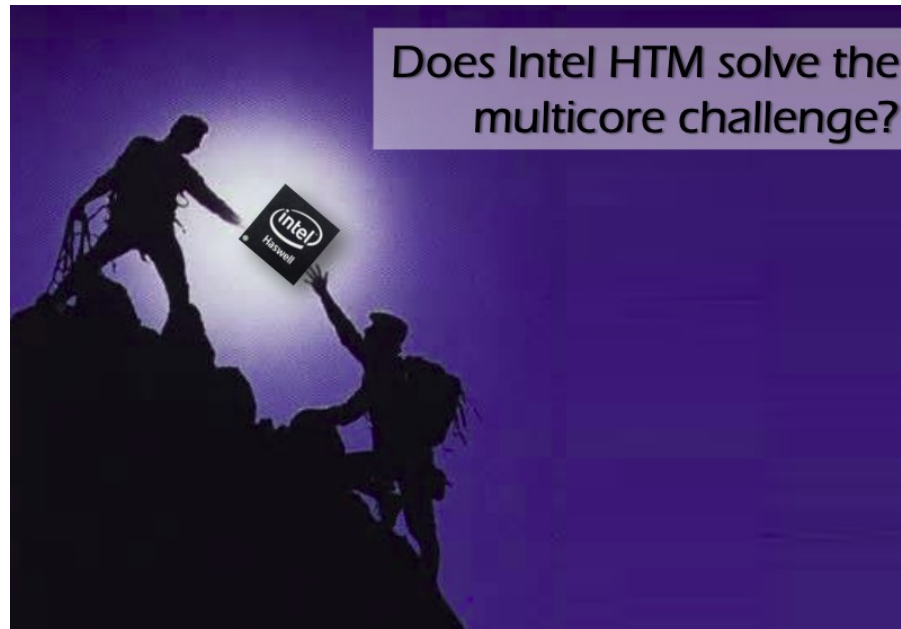
Other implementations still exist

- E.g., ARM Transactional Memory Extension (TME)



Does RTM^[*] solve all the problem?

Restricted **transactional memory**



* Since we focus on RTM in this lecture, we will use RTM and HTM interchangeably.

A little introduction to RTM's ISA

ISA support for transactional memory

Recall: Writing with TX is straightforward

- Use `TX.begin` to mark when a TX starts
- Use `TX.commit` to commit the TX

In RTM, the concept is similar (using new instructions)

- Use `xbegin` to mark an RTM execution start
- Use `xend` to mark an RTM end

Programming with RTM

If transaction starts successfully

- Do work protected by RTM, and then try to commit

Fallback routine to handle abort event

- If abort, system rollback to `xbegin`, return an abort code

Manually abort inside a transaction

```
if _xbegin() ==  
_XBEGIN_STARTED:  
    if conditions:  
        _xabort()  
        critical code  
        _xend()  
    else  
        fallback routine
```

Programming with RTM

If transaction starts successfully

- Do work protected by RTM, and then try to commit

Fallback routine to handle abort event

- If abort, CPU rollbacks to line `xbegin`, return an abort code, and goto fallback

Manually abort inside a transaction

```
if _xbegin() ==  
_XBEGIN_STARTED:  
    if conditions:  
        _xabort()  
        critical code (access x)  
        _xend()  
else  
    fallback routine
```

Another process

```
x = 1
```

Programming with RTM

If transaction starts successfully

- Do work protected by RTM, and then try to commit

Fallback routine to handle abort event

- If abort, CPU rollbacks to line `xbegin`, return an abort code, and goto fallback

Manually abort inside a transaction

```
if _xbegin() ==  
_XBEGIN_STARTED:  
    if conditions:  
        _xabort()  
        critical code  
        _xend()  
else  
    fallback routine
```

How to handle aborts? Can we use simple retry?

RTM provides new instruction set

- `xbegin()`, `xend()` & `xabort()`

Problem: RTM cannot guarantee success

Beginning:

```
if _xbegin() == _XBEGIN_STARTED
    /* do some critical work */
    _xend()
else
    goto beginning
```

Simply Retry

Can we use simple retry? No

Problem: RTM cannot guarantee success

- Similar to OCC, code in RTM can be frequently aborted due to conflicts

Must switch to **a fallback path** after *some* retries (e.g., using a counter)

- E.g., degrade to use lock in the fallback path (since it's not in a transaction)

```
if _xbegin() == _XBEGIN_STARTED
    if lock.held()                check lock status to avoid conflict with
        xabort()                 fallback handlers on other CPU
        /* do some critical work */
    _xend()
else
    lock.acquire()
```

Switch to Pessimistic Sync.

Fun facts about the implementation of RTM

The benefits of RTM

- Memory operations between `xbegin()` & `xend()` satisfy ACI properties
- HTM itself is inspired from transactions

Drawbacks

- Cannot guarantee success

Sounds familiar?

Fun facts about the implementation of RTM

The benefits of RTM

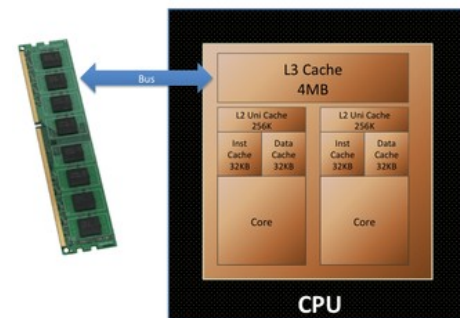
- Memory operations between `xbegin()` & `xend()` satisfy ACI properties
- HTM itself is inspired from transactions

Drawbacks

- Cannot guarantee success

Intel implements RTM using OCC

- Use CPU cache to track the **read/write sets**
- How to detect conflicts? Based on cache coherence protocols
- **Questions:** what are the other limits of using CPU cache to tack read/write sets?

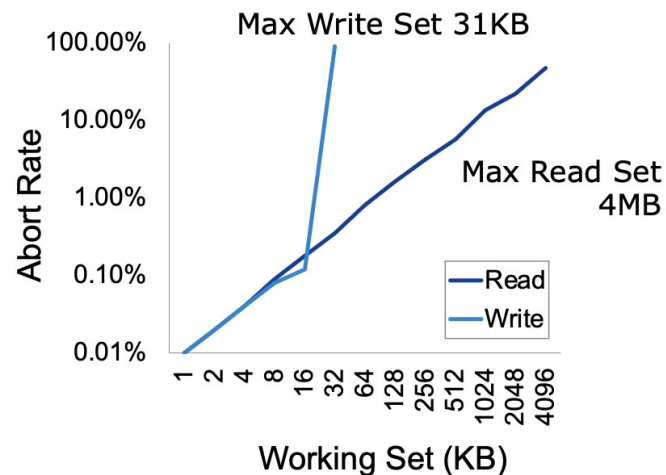


Another drawback of RTM: limited working set

Reason: CPU has limited cache size

How big is the RTM **read/write sets**?

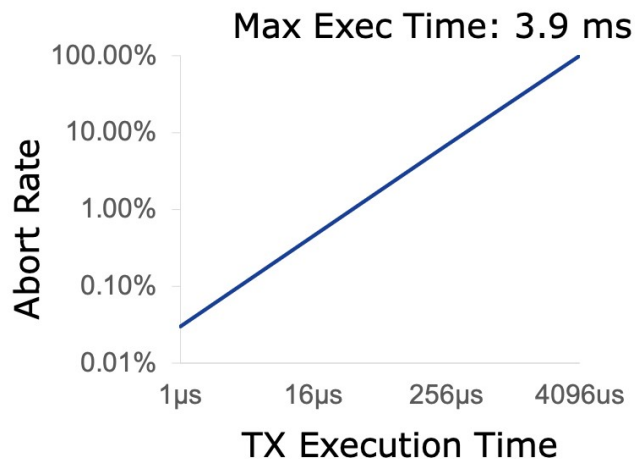
- Depends on various factors
 - Hardware setup (e.g., CPU cache size)
 - Access pattern: read or write
 - L1 cache tracks all the writes
 - L2, or L3 to tracks all the reads
 - Why read set is bigger?



How long can an RTM program execute?

Not very long

- The longer the execution, the higher the probability a TX aborts
- Affected by other facts, e.g., CPU interrupts



HTM on transactions

Naïve: using HTM to execute the in-memory transaction

- Smallbank: transfer & Audit
- TPC-C: much complex, insert 10+ orders and update 10+ stocks

Silo@SOSP'13: the fastest in-memory OCC implementation

Short summary of RTM

Hardware support for transactional memory

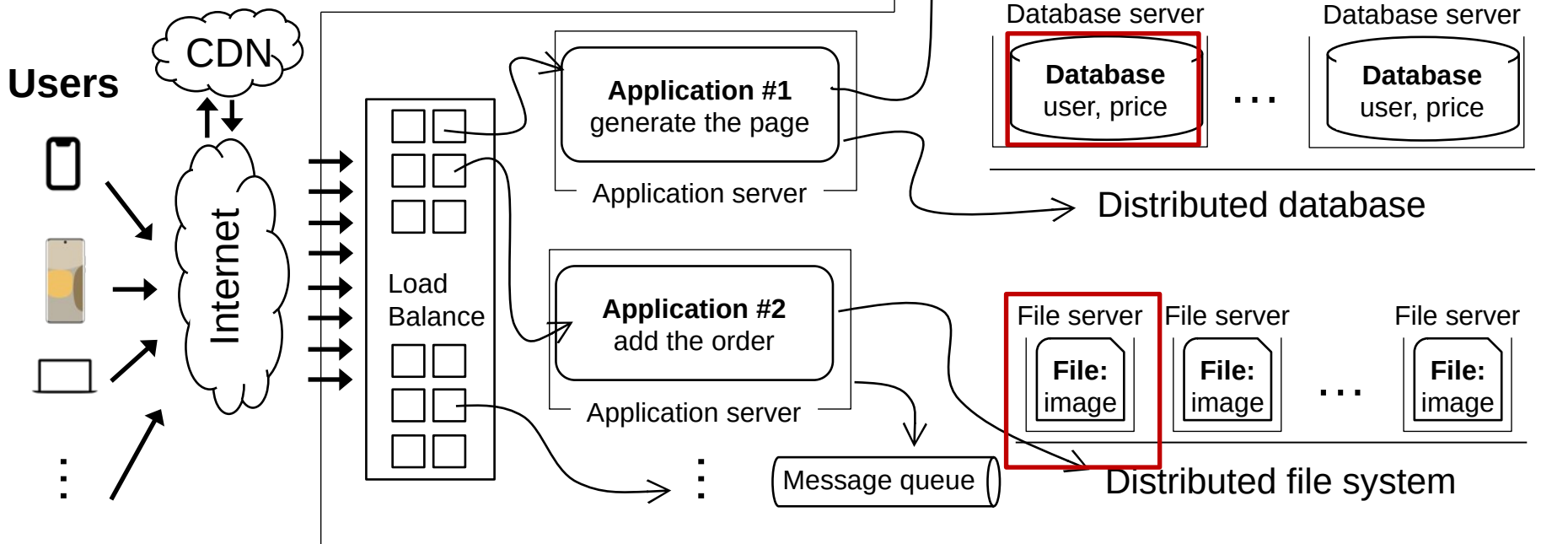
- Easy programming model for the programmer (no need to wrap transaction in read/write API of the transaction)
 - As long as the programmer do the in-memory computations, e.g., in-memory database
- Good performance if using properly
 - No need for locking & atomic operations
- However, the programmer should handle its pitfalls

Transactions


Review: widely used transaction (TX)s



All can use TX to
manager data!
e.g., FS meta data



Review: widely used transaction (TX)s

店铺宝贝	商品属性	单价	数量	优惠方式	小计
店铺: 天猫国际进口超市					
 【直营正品88会员9.5折】Appl... 颜色分类: 深空灰色 网络类型: WIFI 发货时间: 店铺预售, 付款后3天内 存储容量: 64GB 套餐类型: 官方标配		4788.00	1	无优惠	4788.00
给卖家留言: 选错, 请先和商家协商一致 0/200		店铺优惠: 省300元; 满减活动			-300.00
		运送方式: 普通配送 快速 免邮			0.00
		运费险: <input checked="" type="checkbox"/> 运费险 卖家赠送, 退换货可赔			0.00
店铺合计(含运费) ¥4488.00					
<input type="checkbox"/> 朋友代付 <input checked="" type="checkbox"/> 匿名购买 <input type="checkbox"/> 花呗分期					

实付款: **¥4488.00**

寄送至: 上海...

[返回购物车](#) [提交订单](#)

if item.count > 0:
 item.count -= 1
 Cart.add(item)

Review: What is a transaction (TX)?

An abstraction to manage the data

Data is also an abstract concept, can be arbitrary computing data.

Concrete examples including:

- Key-value store entries
- File system metadata (e.g., directory,, inode, etc)
- Processor's metadata (e.g., child processors)

Look like similar program, with data managed by the TX system, and extra mark to denote the start/end of a TX

Review: What a TX typically provide? ACID

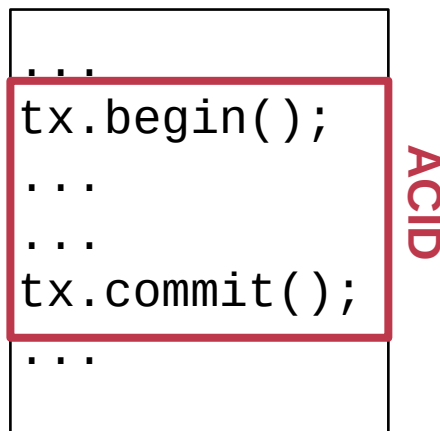
The program btw { `tx.begin()` & `tx.commit()` } has the following properties:

A : Atomicity

C : Consistency

I : Isolation

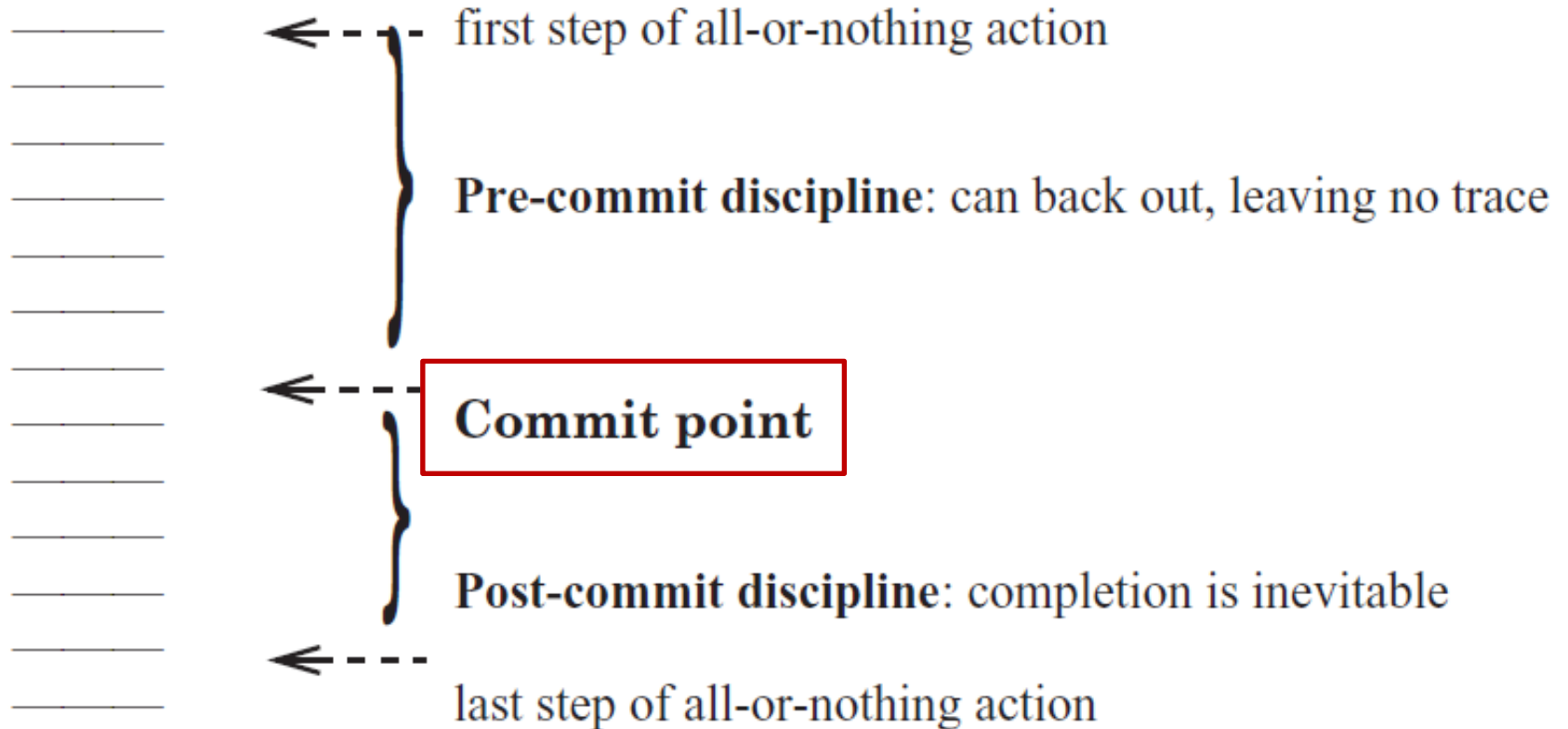
D : Durability



Application program



Review: Atomicity: All-or-nothing



Review: Consistency

Transaction must change the data from a consistent state to another

- What is consistent is **defined by the applications**
- E.g., transfer should leave the $\text{sum}(\text{bank}[a] + \text{bank}[b])$ **unchanged**

Transaction alone is not sufficient for application consistency

- E.g., If the programmer writes the incorrect program.
 - $\text{bank}[a] = \text{bank}[a] - \text{amt} * 2$

```
transfer(bank, a, b, amt):
```

```
    bank[a] = bank[a] - amt
```

```
    bank[b] = bank[b] + amt
```

Review: Isolation & Durability

Isolation: Two concurrently transactions are isolated

- E.g., not viewing the intermediate results of another TX
- Avoid the race conditions

Durability

- Once a transaction is committed, its changes (e.g., writes) must durably stored to a persistent storage

Review: Enforcing ACID properties

Enforcing A & D:

- Logging and recovery (see the last last lecture)

-Enforcing C:

- Database constraint system (not covered in this class)

Enforcing I:

- Concurrency control methods (last & **this lecture**)

Review: Serialization

Goal: run transactions T1, T2, .., TN concurrently, and have it "**appears**" as if they ran sequentially

T1

begin

read(x)

tmp = read(y)

write(y, tmp+10)

commit

T2

begin

write(x, 20)

write(y, 30)

commit

Review: Why serializability is ideal?

Because it simplifies the programmer to enforce consistency

- Recall: consistency depends on how the programmer writes the program

Consistency

Transaction must change the data from a consistent state to another

- What is consistent is **defined by the applications**
- E.g., transfer should leave the $\text{sum}(\text{bank}[a] + \text{bank}[b])$ **unchanged**

Review: Why serializability is ideal?

Assumption: programmers are pro at writing single-thread programs

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

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

- i.e., the concurrent execution can reduce to a serial execution. If C_0 is consistent, then C_n must be consistent

Why we need to use a TX?

Programmers can manually add logging & locking to their program

- Can guarantee all-or-nothing and before-of-after

However, this is a typically bad idea

- Question: what if the programmer falsely releases the lock (like the fine-grained lock example in our previous lecture?)
- What if the programmer write the wrong logging & recovery mechanism?

Review of TX: an abstraction to ease data management

Handle **failure atomicity** (all-or-nothing) & **race conditions** (before-or-after)

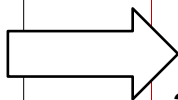
Writing with TX is straightforward, the atomicity is left to the system

- Use `TX.begin` to mark when a TX starts
- Use `TX.commit` to commit the TX
- Rewrite the data read/write with TX's interface

- E.g., `bank[a] = bank[a] - amt`
`tx.write(a, tx.read(a) - amt);`

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

Program



```
transfer(bank, a, b, amt, tx):  
    tx.begin()  
    tx.write(b, tx.read(b) +  
amt);  
    tx.write(a,  
amt);
```

Transaction

Summary

Transactions provide ACID properties

OCC & 2PL are basic protocols to provide serializability

- Problem of 2PL: locking overhead & deadlock
- Problem of OCC: False abort & livelock

Hardware transactional memory (HTM)

- Advanced CPU features inspired by ACID properties of TXs
- Commercial implementation of HTM uses OCC
 - Leads to several drawbacks

The cost of concurrency control can be reduced w/ relaxed isolation level

- Snapshot Isolation (require global counter, not serializable)