Concurrent systems

Lecture 6: Further transactions

Dr Robert N. M. Watson

Reminder from last time

- Concurrency without shared data (Active Objects)
- Message passing; the actor model (Linda, occam, Erlang)
- Composite operations; **transactions**, ACID properties; isolation and serialisability

Last time: isolation – serializability

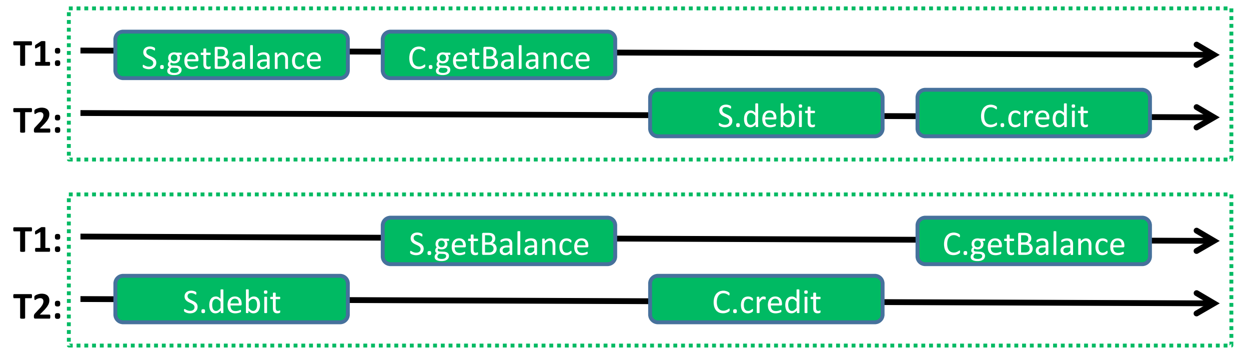
- The idea of executing transactions **serially** (one after the other) is a useful model
  - We want to run transactions concurrently
  - But the result should be **as if** they ran serially
- Consider two transactions, T1 and T2

Isolation and serialisability allow programmers to reason about the interactions between transactions trivially.

But how can the transaction system itself decide whether a given concurrent execution of two transactions is allowable?

From last lecture

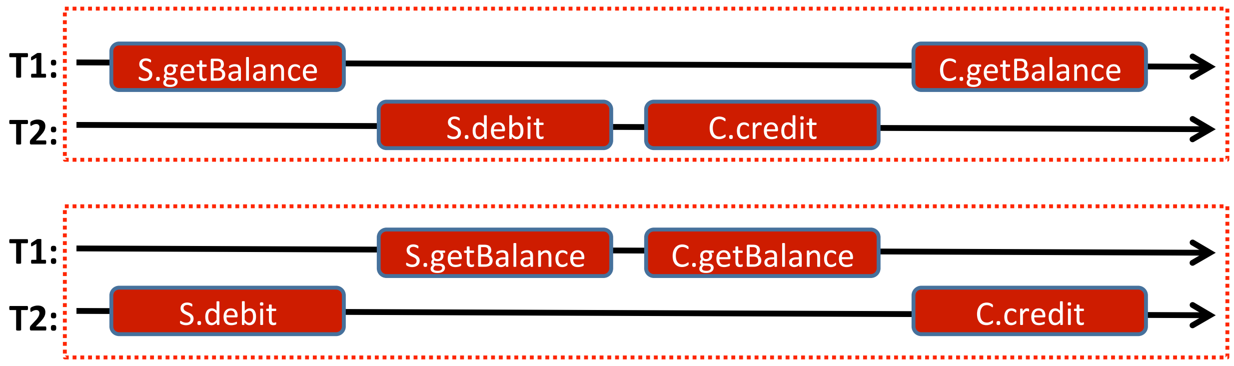
Isolation – serializability



- First case is serial and, as expected, all ok
- Second case is not serial … but result is fine
  - Both of T1’s operations happen after T2’s update
  - This is a **serializable** schedule [as is first case]

From last lecture

Isolation – serializability



- Neither of these two executions is ok
- T1 sees inconsistent values:
  - (top) sees updated version of C, but old version of S
  - (bottom) sees updated S, but original version of C

This time

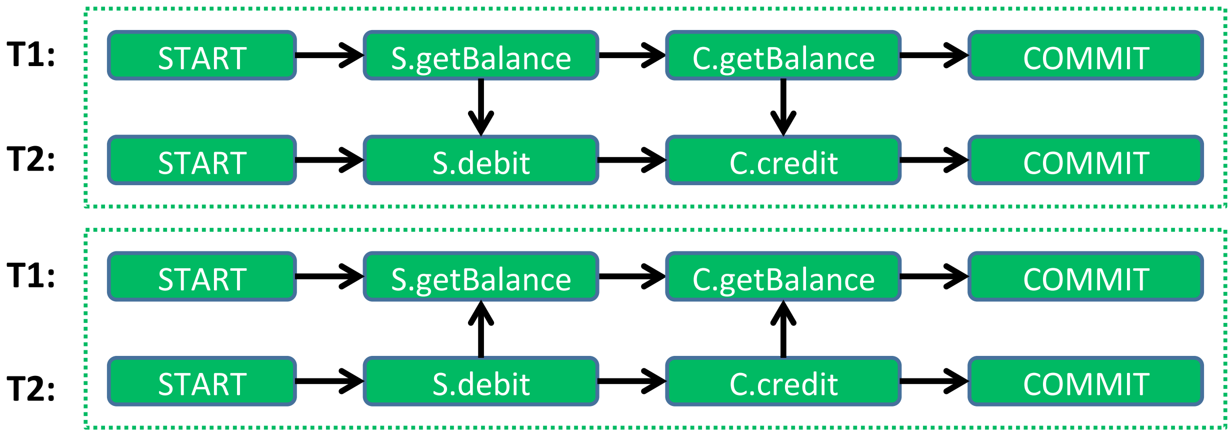
- History graphs; good (and bad) schedules
- Isolation vs. strict isolation; enforcing isolation
- Two-phase locking; rollback
- Timestamp ordering (TSO)
- Optimistic concurrency control (OCC)
- Isolation and concurrency summary

This lecture considers how the transaction implementation itself can provide transactional (ACID) guarantees

History graphs

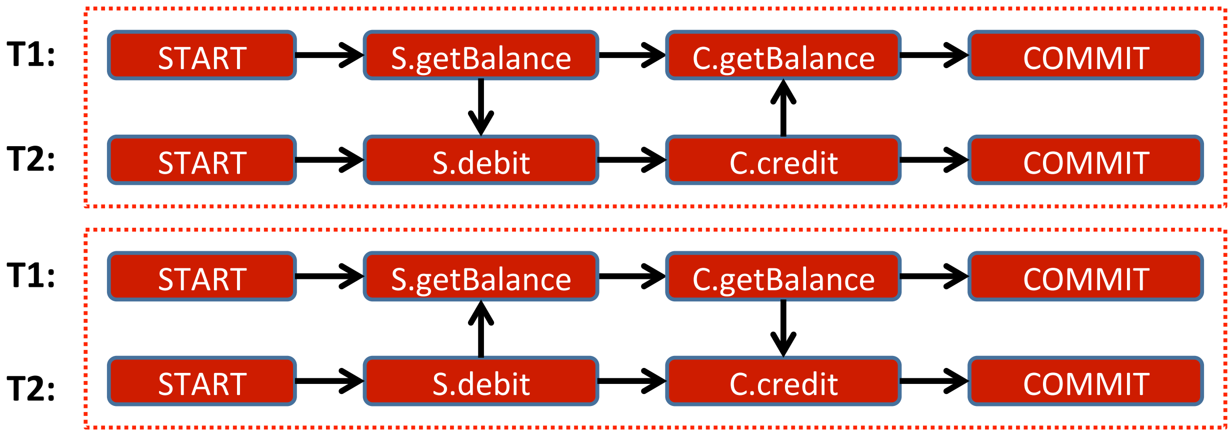
- Can construct a graph for any execution:
    - Nodes represent individual operations, and
    - Arrows represent "happens-before" relations
- Operations within a given transaction must happen in program order (i.e. as written)
- **Conflicting** operations are ordered by the implementation of the underlying object
    - conflicting operations = non-commutative
    - e.g. A.credit(), A.debit() commute [don't conflict], while A.credit() and A.addInterest() **do** conflict
- The next few graphs represent **schedules** rather than **possible schedules**

History graphs: good schedules



- Same schedules as before (both ok)
- Can easily see that everything in T1 either happens before everything in T2, or vice versa
    - Hence schedule can be serialized

History graphs: bad schedules



- Both schedules are bad :-(
  - Arrows from T1 to T2 mean “T1 must happen before T2”
  - But arrows from T2 to T1 => “T2 must happen before T1”
- Can’t both be true => schedules are not serializable.

Causes of bad schedules

- **Lost Updates**
  - T1 updates (writes) an object, but this is then overwritten by concurrently executing T2
  - (also called a write-write conflict)

Lack of **atomicity**: operation results “lost”

- **Dirty Reads**
  - T1 reads an object which has been updated an uncommitted transaction T2
  - (also called a read-after-write conflict)

Lack of **isolation**: partial result seen

- **Unrepeatable Reads**
  - T1 reads an object which is then updated by T2
  - Not possible for T1 to read the same value again
  - (also called a write-after-read conflict)

Lack of **isolation**: read value unstable

Atomicity: all or none of operations performed
Isolation: transactions execute as if isolated from concurrent effects

Isolation and strict isolation

- Ideally want to avoid all three problems
- Two ways: Strict Isolation and Non-Strict Isolation
  - **Strict Isolation**: guarantee we never experience lost updates, dirty reads, or unrepeatable reads
  - **Non-Strict Isolation**: let transaction continue to execute despite potential problems
- Non-strict isolation usually allows more concurrency but can lead to complications
  - e.g. if T1 reads something written by T2 (a “dirty read”) then T1 cannot commit until T2 commits
  - and T1 must abort if T2 aborts: **cascading aborts**

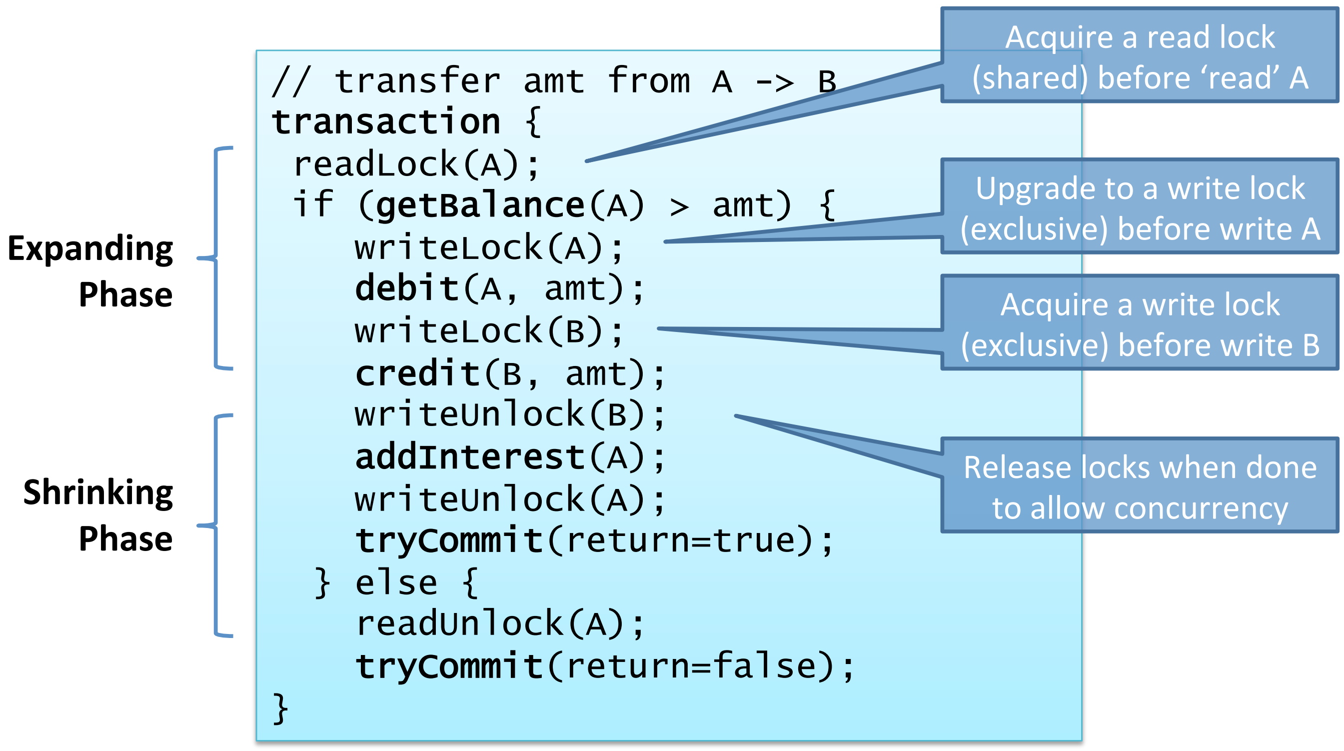
Enforcing isolation

- In practice there are a number of techniques we can use to enforce isolation (of either kind)
- We will look at:
  - Two-Phase Locking (2PL);
  - Timestamp Ordering (TSO); and
  - Optimistic Concurrency Control (OCC)

Two-phase locking (2PL)

- Associate a lock with every object
  - Could be mutual exclusion, or MRSW
- Transactions proceed in two phases:
  - Expanding Phase: during which locks are acquired but none are released
  - Shrinking Phase: during which locks are released, and no more are acquired
- Operations on objects occur in either phase, providing appropriate locks are held
  - Should ensure serializable execution

2PL example



```
// transfer amt from A -> B
transaction {
 readLock(A);
 if (getBalance(A) > amt) {
    writeLock(A);
    debit(A, amt);
    writeLock(B);
    credit(B, amt);
    writeUnlock(B);
    addInterest(A);
    writeUnlock(A);
    tryCommit(return=true);
  } else {
    readUnlock(A);
    tryCommit(return=false);
}
```

Acquire a read lock (shared) before 'read' A

Upgrade to a write lock (exclusive) before write A

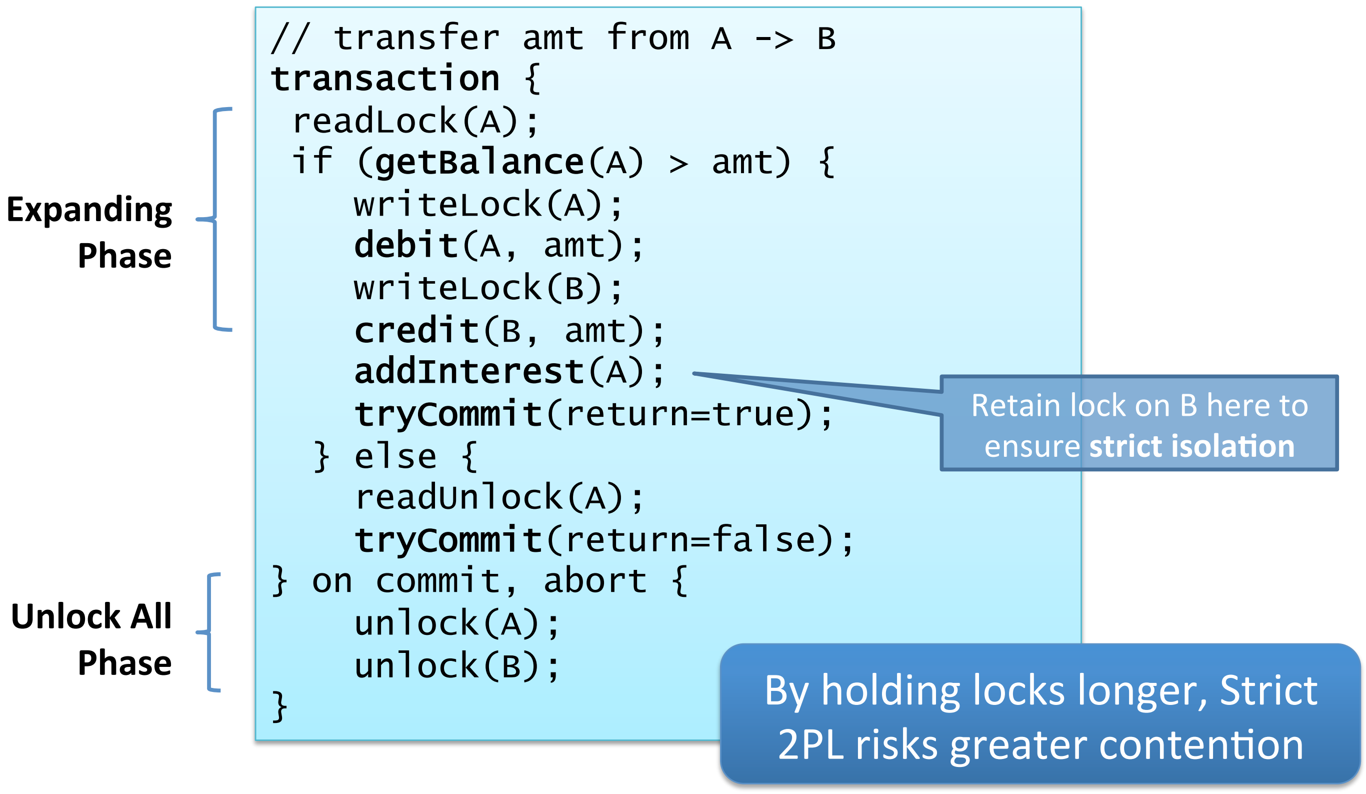
Acquire a write lock (exclusive) before write B

Release locks when done to allow concurrency

Problems with 2PL

- Requires knowledge of which locks required
  - Can be automated in many systems
- Risk of deadlock
  - Can attempt to impose a partial order
  - Or can detect deadlock and abort, releasing locks
  - (this is safe for transactions, which is nice)
- Non-strict Isolation: releasing locks during execution means others can access those objects
  - e.g. T1 updates A, then releases write lock; now T2 can read or overwrite the uncommitted value
  - Hence T2's fate is tied to T1 (whether commit or abort)
  - Can fix with **strict 2PL**: hold all locks until transaction end

Strict 2PL example



Expanding Phase

```
// transfer amt from A -> B
transaction {
 readLock(A);
 if (getBalance(A) > amt) {
    writeLock(A);
    debit(A, amt);
    writeLock(B);
    credit(B, amt);
    addInterest(A);
    tryCommit(return=true);
  } else {
    readUnlock(A);
    tryCommit(return=false);
} on commit, abort {
    unlock(A);
    unlock(B);
}
```

Unlock All Phase

Retain lock on B here to ensure **strict isolation**

By holding locks longer, Strict 2PL risks greater contention

2PL: rollback

- Recall that transactions can **abort**
  - Could be due to run-time conflicts (non-strict 2PL), or could be programmed (e.g. on an exception)
- Using locking for isolation works, but means that updates are made 'in place'
  - i.e. once acquire write lock, can directly update
  - If txaction aborts, need to make sure no effects visible
- **Rollback** is the process of returning the world to the state it in was before the start of the txaction

Why might a transaction abort?

- Some failures are internal to the transaction system:
  - Transaction T2 depends on T1, and T1 aborts
  - Deadlock is detected between two transactions
  - Memory is exhausted or a system error occurs
- Some are programmer-triggered:
  - Transaction self-aborted – e.g., `debit()` failed due to inadequate balance
- Some failures will be programmer visible
- Others will simply trigger retry of the transaction

Implementing rollback: undo

- One strategy is to **undo** operations, e.g.
  - Keep a log of all operations, in order: O_1, O_2, .. O_n
  - On abort, undo changes of O_n, $O_{(n-1)}$, .. O_1
- Must know how to undo an operation:
  - Assume we log both operations and parameters
  - Programmer can provide an explicit counter action
    - UNDO(credit(A, x)) ⇔ debit(A, x);
- May not be sufficient (e.g. setBalance(A, x))
  - Would need to record previous balance, which we may not have explicitly read within transaction…

Implementing rollback: copy

- A more brute-force approach is to take a copy of an object before [first] modification
  - On abort, just revert to original copy
- Has some advantages:
  - Doesn't require programmer effort
  - Undo is simple, and can be efficient (e.g. if there are many operations, and/or they are complex)
- However can lead to high overhead if objects are large … and may not be needed if don't abort!
  - Can reduce overhead with partial copying

Timestamp ordering (TSO)

- 2PL and Strict 2PL are widely used in practice
  - But can limit concurrency (certainly the latter)
  - And must be able to deal with deadlock
- **TSO** is an alternative approach:
  - As a transaction begins, it is assigned a timestamp
  - Timestamps are comparable, and unique (can think of as e.g. current time – ~~or as a ticket from a sequencer~~)
  - Every object O records the timestamp of the last transaction to successfully access it: **V**(O)
  - T can access object O iff **V**(T) >= **V**(O), where **V**(T) is the timestamp of T (otherwise rejected as "***too late***")

Timestamps allow us to explicitly track new "happens-before" edges, detecting (and preventing) violations

TSO example 1

```
T1 transaction {
  s = getBalance(S);
  c = getBalance(C);
  return = s + c;
}
```

```
T2 transaction {
  debit(S, 100);
  credit(C, 100);
  return true;
}
```

Imagine S and C start off with version 10

1. T1 and T2 both start concurrently:
   - T1 gets timestamp **27**, T2 gets timestamp **29**
2. T1 reads S => ok! (27 >= 10); S gets timestamp 27
3. T2 does debit S, 100 => ok! (29 >= 27); S gets timestamp 29
4. T1 reads C => ok! (27 => 10); C gets timestamp 27
5. T2 does credit C, 100 => ok! (29 >= 27); C gets timestamp 29
6. Both transactions commit.

TSO example 2

```
T1 transaction {
   s = getBalance(S);
   c = getBalance(C);
   return = s + c;
}
```

```
T2 transaction {
   debit(S, 100);
   credit(C, 100);
   return true;
}
```

As before, S and C start off with version 10

1. T1 and T2 both start concurrently:
   - T1 gets timestamp **27**, T2 gets timestamp **29**
2. T1 reads S => **ok**! (27 >= 0); S gets timestamp 27
3. T2 does debit S, 100 => **ok**! (29 >= 27); S gets timestamp 29
4. T2 does credit C, 100 => **ok**! (29 >= ~~0~~10); C gets timestamp 29
5. T1 reads C => **FAIL**! (27 < 29); T1 aborts
6. T2 commits; T1 restarts, gets timestamp **30**…

Advantages of TSO

- Deadlock free
- Can allow more concurrency than 2P~~C~~L
- Can be implemented in a decentralized fashion
- Can be augmented to distinguish reads & writes
  - objects have read timestamp **R** & write timestamp **W**

```
READ(O, T) {
   if(V(T) < W(O)) abort;
   // do actual read
   R(O): = MAX(V(T), R(O));
}
```

Only safe to read if no-one wrote "after" us

R(O) holds timestamp of *latest* transaction to read

```
WRITE(O, T) {
   if(V(T) < R(O)) abort;
   if(V(T) < W(O)) return;
   // do actual write
   W(O) := V(T);
}
```

Unsafe to write if later txaction has read value

But if later txaction *wrote* it, just skip write (he won!). Or?

However…

- TSO needs a rollback mechanism (like 2PL)
- TSO does not provide strict isolation:
  - hence subject to cascading aborts
  - (can provide strict TSO by locking objects when access is granted – still remains deadlock free)
- TSO decides *a priori* on one serialization
  - even if others might have been possible
- And TSO does not perform well under contention
  - will repeatedly have transactions aborting & retrying & …
- In general TSO is a good choice for *distributed* systems [decentralized management] where conflicts are rare

Something to think about: can TSO livelock?

Optimistic concurrency control

- **OCC** is an alternative to 2PL or TSO
- Optimistic since assume conflicts are rare
  - Execute transaction on a **shadow** [copy] of the data
  - On commit, check if all “OK”; if so, apply updates; otherwise discard shadows & retry
- “OK” means:
  - All shadows read were mutually consistent, and
  - No-one else has committed changes to any object that we are hoping to update
- Advantages: no deadlock, no cascading aborts
  - And “rollback” comes pretty much for free!

Key idea: when ready to commit, search for a serialisable order that accepts the transaction

Implementing OCC

- Various efficient schemes for shadowing
  - e.g. write buffering, page-based copy-on-write.
- Complexity arises in performing **validation** when a transaction T finishes & tries to commit
- Read Validation:
  - Must ensure that all versions of data read by T (all shadows) were valid at some particular time *t*
  - This becomes the tentative **start time** for T
- Serializability Validation:
  - Must ensure that there are no conflicts with any transactions which have an earlier start time

OCC example (1)

- All objects are tagged with a version
  - Validation timestamp of the transaction which most recently wrote its updates to that object
- Many threads execute transactions
  - When wish to read an object, take a shadow copy, and take note of the version number
  - If wish to write: first take copy, then update that
- When a thread finishes a transaction, it submits the versions to a single threaded validator

OCC example (2)

- Validator keeps track of last *k* validated transactions, their timestamps, and the objects they updated

Transaction	Validation Timestamp	Objects Updated	Writeback Done?
T5	10	A, B, C	Yes
T6	11	D	Yes
T7	12	A, E	No

- The versions of the objects are as follows:
  - T7 has started, but not finished, writeback
  - (A has been updated, but not E)

Object	Version
A	12
B	10
C	10
D	11
E	9

What will happen if we now start a new transaction **T8** on {B, E} before **T7** writes back E?

OCC example (3)

- Consider T8: { write(B), write(E) };
- T8 executes and makes shadows of B & E
  - Records timestamps: B@10, E@9
  - When done, T8 submits for validation
- Phase 1: read validation
  - Check shadows are part of a consistent snapshot
  - Latest committed start time is 11 = ok (10, 9 < 11)
- Phase 2: serializability validation
  - Check T8 against all later transactions (here, T7)
  - Conflict detected! (T7 updates E, but T8 read old E)

Looking at log: have other transactions interfered with **T8**'s inputs?

Looking at log: would committing **T8** invalidate other now-committed transactions?

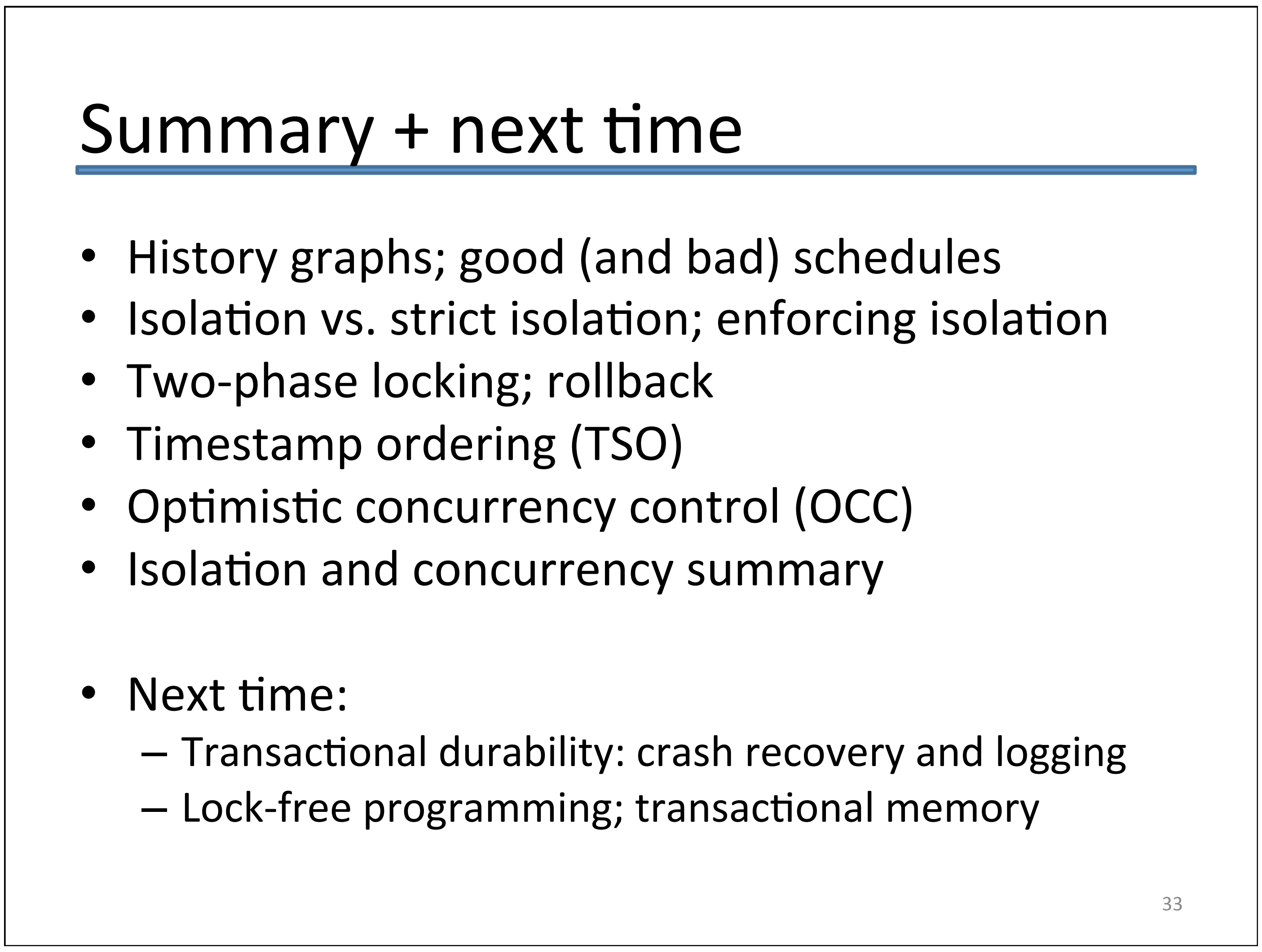
Issues with OCC

- Preceding example uses a simple validator
  - Possible will abort even when don't need to
  - (e.g. can search for a 'better' start time)
- In general OCC can find more serializable schedules than TSO
  - Timestamps assigned after the fact, and taking the actual data read and written into account
- However OCC is not suitable when high conflict
  - Can perform lots of work with 'stale' data => wasteful!
  - ~~Livelock~~ Starvation possible if conflicting set continually retries

Something think about: what happens when *k*-transaction log is exhausted?

Isolation & concurrency: summary

- **2PL** explicitly locks items as required, then releases
  - Guarantees a serializable schedule
  - Strict 2PL avoids cascading aborts
  - Can limit concurrency; & prone to deadlock
- **TSO** assigns timestamps when transactions start
  - Cannot deadlock, but may miss serializable schedules
  - Suitable for distributed/decentralized systems
- **OCC** executes with shadow copies, then validates
  - Validation assigns timestamps when transactions end
  - Lots of concurrency, & admits many serializable schedules
  - No deadlock but potential livelock when contention is high
- Ideas like TSO and OCC will recur in Distributed Systems



Summary + next time

- History graphs; good (and bad) schedules
- Isolation vs. strict isolation; enforcing isolation
- Two-phase locking; rollback
- Timestamp ordering (TSO)
- Optimistic concurrency control (OCC)
- Isolation and concurrency summary

- Next time:
  - Transactional durability: crash recovery and logging
  - Lock-free programming; transactional memory