Concurrent systems

Lecture 6: Isolation vs. Strict Isolation, 2-Phase Locking (2PL), Time Stamp Ordering (TSO), and Optimistic Concurrency Control (OCC)

Dr Anil Madhavapeddy

Reminder from last time

- Concurrency without shared data
  - Active objects
- Message passing; the actor model
  - Occam, Erlang
- Composite operations
  - Transactions, ACID properties
  - Isolation and serialisability
- History graphs; good (and bad) schedules

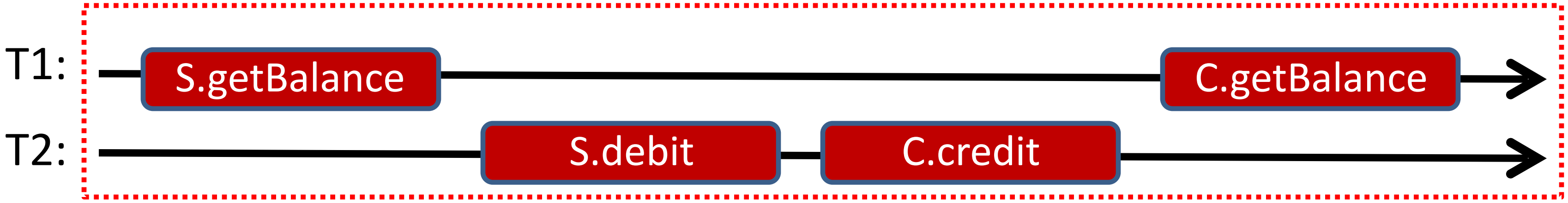
Last time: isolation – serialisability

- 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

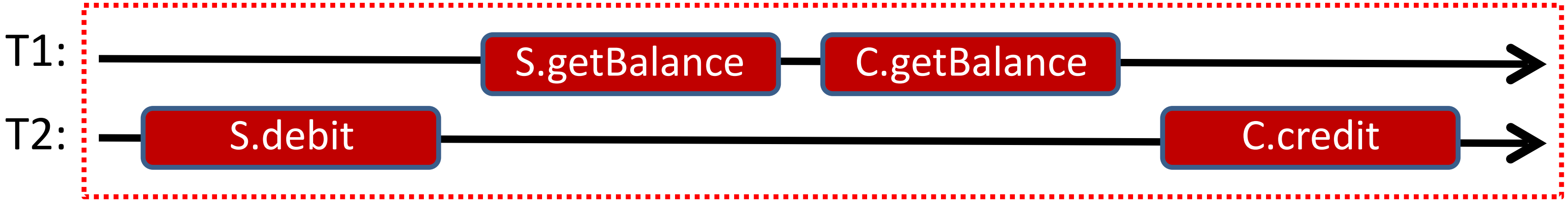
Isolation allow transaction programmers to reason about the interactions between transactions trivially: they appear to execute in serial.

Transaction systems execute transactions concurrently for performance and rely on the definition of serialisability to decide if an actual execution schedule is allowable.

Isolation – serialisability



- This execution is neither serial nor serialisable
  - T1 sees inconsistent values: old S and new C



The transaction system must ensure that, regardless of any actual concurrent execution used to improve performance, only results consistent with serialisable orderings are visible to the transaction programmer.

This time

- Effects of 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

Effects 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)
- Dirty Reads
  - T1 reads an object which has been updated an uncommitted transaction T2
  - (also called a read-after-write conflict)
- 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 atomicity: operation results "lost"

Lack of isolation: partial result seen

Lack of isolation: read value unstable

Atomicity: all or none of operations performed – abort must be "clean"
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 (i.e., more optimistic)
- Non-strict isolation usually allows more concurrency but can lead to complications
  - E.g. if T2 reads something written by T1 (a "dirty read") then T2 cannot commit until T1 commits
  - And T2 must abort if T1 aborts: cascading aborts
- Both approaches ensure that only serialisable schedules are visible to the transaction programmer

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)
- More complete descriptions and examples of these approaches can be found in:

  Operating Systems, Concurrent and Distributed Software Design, Jean Bacon and Tim Harris, Addison-Wesley 2003.

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);
}
```

Expanding Phase

Shrinking Phase

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
  - Easy if a transaction statically declares its affected objects
  - But some transactions look up objects dynamically
- Risk of deadlock
  - Can attempt to impose a partial order
  - Or can detect deadlock and abort, releasing locks
  - (this is safe for transactions due to rollback, 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

Unlock All 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);
}
```

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 transaction aborts, need to ensure no visible effects
- Rollback is the process of returning the world to the state it in was before the transaction started
  - I.e., to implement atomicity: all happened, or none.

Why might a transaction abort?

- Some failures are internal to transaction systems:
  - 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 must be programmer visible
- Others may 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
- Time Stamp Ordering (TSO) is an alternative approach:
  - As a transaction begins, it is assigned a timestamp – the proposed eventual (total) commit order / serialisation
  - Timestamps are comparable, and unique (can think of as e.g. current time – or a logical incrementing number)
  - Every object O records the timestamp of the last transaction to successfully access (read? write?) 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")
  - If T is non-serialisable with timestamp, abort and roll back

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 that objects 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.

Succeeded as all conflicting operations executed in timestamp order

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 >= 10); 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 >= 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 2PL
- 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 if can abort)
- TSO decides a priori on one serialisation
  - 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

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 "later" 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 (1)

- NB: This is a simplified presentation of the algorithm – please refer to the book for the full description!
- 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
- Serialisability validation:
  - Must ensure that there are no conflicts with any committed transactions which have an later start time

Implementing OCC (2)

- 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 (1)

- 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 (2)

- 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!
  - Starvation possible if conflicting set continually retries
  - Will the transaction system always make progress?

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
- Differing tradeoffs between optimism, concurrency, but also potential starvation, livelock, and deadlock
- Ideas like TSO/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