2PL: Rollback

• Recall that transactions can **abort**
  – Could be 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 make sure no effects visible
• **Rollback** is the process of returning the world to the state it in was before the start of the transaction
Implementing Rollback: Undo

• One strategy is to **undo** operations, e.g.
  – Keep a log of all operations, in order: \(O_1, O_2, \ldots, O_n\)
  – On abort, undo changes of \(O_n, O_{(n-1)}, \ldots, O_1\)

• 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 \( V(T) \geq V(O) \), where \( V(T) \) is the timestamp of T (otherwise rejected as “too late”)
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

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); 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 2PC
• Can be implemented in a decentralized fashion
• Can be augmented to distinguish reads & writes
  – objects have read timestamp $R$ & write timestamp $W$

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

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

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

Unsafe to write if later transaction has read value

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

Only safe to read if no-one wrote “after” us
However...

- TSO needs a rollback mechanism (like 2PC)
- 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
Optimistic Concurrency Control

- **OCC** is an alternative to 2PC 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!
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

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Validation Timestamp</th>
<th>Objects Updated</th>
<th>Writeback Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5</td>
<td>10</td>
<td>A, B, C</td>
<td>Yes</td>
</tr>
<tr>
<td>T6</td>
<td>11</td>
<td>D</td>
<td>Yes</td>
</tr>
<tr>
<td>T7</td>
<td>12</td>
<td>A, E</td>
<td>No</td>
</tr>
</tbody>
</table>

- The versions of the objects are as follows:
  - T7 has started, but not finished, writeback
  - (A has been updated, but not 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)
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 possible if conflicting set continually retries
Isolation & Concurrency: Summary

• **2PL** explicitly locks items as required, then releases
  – Guarantees a serializable schedule
  – Strict 2PC 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
Crash Recovery & Logging

• Transactions require ACID properties
  – So far have focused on I (and implicitly C).
• How can we ensure Atomicity & Durability?
  – Need to make sure that if a transaction always done entirely or not at all
  – Need to make sure that a transaction reported as committed remains so, even after a crash
• Consider for now a **fail-stop** model:
  – If system crashes, all in-memory contents are lost
  – Data on disk, however, remains available after reboot
Using Persistent Storage

• Simplest “solution”: write all updated objects to disk on commit, read back on reboot
  – Doesn’t work, since crash could occur during write
  – Can fail to provide Atomicity and/or Consistency

• Instead split update into two stages
  1. Write proposed updates to a write-ahead log
  2. Write actual updates

• Crash during #1 => no actual updates done
• Crash during #2 => use log to redo, or undo
Write-Ahead Logging

• Ordered append-only file on disk
• Contains entries like <txid, obj, op, old, new>
  – ID of transaction, object modified, (optionally) the operation performed, the old value and the new value
  – This means we can both “roll forward” (redo operations) and “rollback” (undo operations)
• When persisting a transaction to disk:
  – First log a special entry <txid, START>
  – Next log a number of entries to describe operations
  – Finally log another special entry <txid, COMMIT>
Using a Write-Ahead Log

- When executing transactions, perform updates to objects in memory with lazy write back
  - i.e. the OS can push changes to disk whenever it wants
- Initially can do the same with the log entries...
- But when wish to commit a transaction, must first **synchronously** flush a commit record to the log
  - Assume there is a ‘fsync’ operation or similar which allows us to force data out to disk
  - Only report transaction as committed when fsync returns
- Can improve performance by delaying flush until we have a number of transaction to commit
  - Hence at any point in time we have some prefix of the write-ahead log on disk, and the rest in memory
The Big Picture

RAM acts as a cache of disk (e.g. no copy of z)

<table>
<thead>
<tr>
<th>Object Values</th>
<th>Log Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 3</td>
<td>T3, START</td>
</tr>
<tr>
<td>y = 27</td>
<td>T2, ABORT</td>
</tr>
<tr>
<td></td>
<td>T2, y, 17, 27</td>
</tr>
<tr>
<td></td>
<td>T1, x, 2, 3</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th>Object Values</th>
<th>Log Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>T2, z, 40, 42</td>
</tr>
<tr>
<td>y = 17</td>
<td>T2, START</td>
</tr>
<tr>
<td>z = 42</td>
<td>T1, START</td>
</tr>
<tr>
<td></td>
<td>T0, COMMIT</td>
</tr>
<tr>
<td></td>
<td>T0, x, 1, 2</td>
</tr>
<tr>
<td></td>
<td>T0, START</td>
</tr>
</tbody>
</table>

On-disk values may be older versions of objects

Log conceptually infinite, and spans RAM & Disk

RAM acts as a cache of disk (e.g. no copy of z)

On-disk values may be older versions of objects

Log conceptually infinite, and spans RAM & Disk

Newer Log Entries

Older Log Entries

On-disk values may be older versions of objects

Log conceptually infinite, and spans RAM & Disk
Checkpoints

• As described, log will get very long
  – And need to process every entry in log to recover
• Better to periodically write a **checkpoint**
  – Flush all current in-memory log records to disk
  – Write a special checkpoint record to log which contains a list of active transactions
  – Flush all ‘dirty’ objects (i.e. ensure object values on disk are up to date)
  – Flush location of new checkpoint record to disk
• (Not fatal if crash during final write)
Checkpoints and Recovery

- Key benefit of a checkpoint is it lets us focus our attention on possibly affected transactions.

<table>
<thead>
<tr>
<th>Time</th>
<th>Checkpoint Time</th>
<th>Failure Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T1: no action required</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2: REDO</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>T3: UNDO</td>
</tr>
<tr>
<td></td>
<td>T3</td>
<td>T4: REDO</td>
</tr>
<tr>
<td></td>
<td>T4</td>
<td>T1: UNDO</td>
</tr>
<tr>
<td></td>
<td>T5</td>
<td>Not active at checkpoint, and still in progress.</td>
</tr>
</tbody>
</table>

Active at checkpoint. Has since committed; and record in log.

Active at checkpoint; in progress at crash.

Not active at checkpoint. But has since committed, and commit record in log.

Not active at checkpoint, and still in progress.
Recovery Algorithm

• Initialize undo list $U = \{ \text{set of active txactions} \}$
• Also have redo list $R$, initially empty
• Walk log forward from checkpoint record:
  – If see a START record, add txaction to $U$
  – If see a COMMIT record, move txaction from $U \rightarrow R$
• When hit end of log, perform undo:
  – Walk backward and undo all records for all Tx in $U$
• When reach checkpoint record again, Redo:
  – Walk forward, and re-do all records for all Tx in $R$
Transactions: Summary

• Standard mutual exclusion techniques not great for dealing with >1 object
  – intricate locking (& lock order) required, or
  – single coarse-grained lock, limiting concurrency

• Transactions allow us a better way:
  – potentially many operations (reads and updates) on many objects, but should execute as if atomically
  – underlying system deals with providing isolation, allowing safe concurrency, and even fault tolerance!

• Transactions widely used in database systems
Advanced Topics

• Will briefly look at two advanced topics
  – lock-free data structures, and
  – transactional memory

• This is informational & not examinable!
  – but worth knowing at least something about

• (Those of you who are super keen are invited to attend Tim Harris’s ACS course:
  – 4pm-6pm on Thu Nov 3, 10 and 17; in SW01)
Lock-free Programming

• What’s wrong with locks?
  – Difficult to get right (if locks are fine-grained)
  – Don’t scale well (if locks too coarse-grained)
  – Don’t compose well (deadlock!)
  – Poor cache behavior (e.g. convoying)
  – Priority inversion
  – And can be expensive

• Lock-free programming involves getting rid of locks ... but not at the cost of safety!
Assumptions

• We have a shared memory system
• Low-level (assembly instructions) include:

```c
val = read(addr);          // atomic read from memory
(void) write(addr, val);   // atomic write to memory
done = CAS(addr, old, new); // atomic compare-and-swap
```

• Compare-and-Swap (CAS) is **atomic**
  • reads value of addr (‘val’), compares with ‘old’, and updates memory to ‘new’ iff old==val -- without interruption!
  • something like this instruction common on most modern processors (e.g. cmpxchg on x86)
• Typically used to build spinlocks (or mutexes, or semaphores, or sequencers, or whatever...)

Lock-free Approach

• Directly use CAS to update shared date
• As an example consider a lock-free linked list of integer values
  – list is singly linked, and sorted
• Represents the ‘set’ abstract data type, i.e.
  – find(int) -> bool
  – insert(int) -> bool
  – delete(int) -> bool
Searching a sorted list

• find(20):

find(20) \rightarrow \text{false}
Inserting an item with CAS

• insert(20):

\[
\text{insert}(20) \rightarrow \text{true}
\]
Inserting an item with CAS

• insert(20):

• insert(25):
Searching and finding together

- find(20) -> false
- insert(20) -> true
Searching and finding together

• find(20) -> false

This thread saw 20 was not in the set...

• insert(20) -> true

...but this thread succeeded in putting it in!

• Is this a correct implementation of a set?
• Should the programmer be surprised if this happens?
• What about more complicated mixes of operations?
Linearizability

• As with transactions, we return to a conceptual model to define correctness
  – a lock-free data structure is ‘correct’ if all changes (and return values) consistent with some serial view: we call this a linearizable schedule
• Hence in the previous example, we were ok:
  – can just deem the find() to have occurred first
• Gets a lot more complicated for more complicated data structures & operations!
  – see Tim Harris’s course for more gory details...
Transactional Memory (TM)

- Steal idea from databases!

- Instead of:

  ```
  lock(&mylock);
  shared[i] *= shared[j] + 17;
  unlock(&mylock);
  ```

  Use:

  ```
  atomic {
    shared[i] *= shared[j] + 17;
  }
  ```

- Has “obvious” semantics, i.e. all operations within block occur as if atomically

- Transactional since under the hood it looks like:

  ```
  do { txid = tx_begin(&thd);
    shared[i] *= shared[j] + 17;
  } while !(tx_commit(txid));
  ```
TM Advantages

• Simplicity:
  – programmer just puts atomic { } around anything he/she wants to occur in isolation

• Composability:
  – unlike locks, atomic { } blocks nest, e.g:

```plaintext
credit(a, x) = atomic {
    setbal(a, readbal(a) + x);
}
debit(a, x) = atomic {
    setbal(a, readbal(a) - x);
}
transfer(a, b, x) = atomic {
    debit(a, x);
    credit(b, x);
}
```
TM Advantages

• Cannot deadlock:
  – No locks, so don’t have to worry about locking order
  – (Though may get livelock if not careful)

• No races (kinda):
  – Cannot forget to take a lock (although you can forget to put atomic `{ }` around your critical section ;-)

• Scalability:
  – High performance possible via OCC
  – No need to worry about complex fine-grained locking
TM is very promising...

• Essentially does ‘ACI’ but no D
  – no need to worry about crash recovery
  – can work entirely in memory
  – some hardware support emerging (or promised)

• But not a panacea
  – Contention management can get ugly
  – Difficulties with irrevocable actions (e.g. IO)
  – Still working out exact semantics (type of atomicity, handling exceptions, signaling, ...)

• For more details, see Tim Harris’s course
Concurrent Systems: Summary

• Concurrency is essential in modern systems
  – overlapping I/O with computation
  – exploiting multi-core
  – building distributed systems

• But throws up a lot of challenges
  – need to ensure safety, allow synchronization, and avoid issues of liveness (deadlock, livelock, ...)

• Major risk of over-engineering
  – generally worth building sequential system first
  – and worth using existing libraries, tools and design patterns rather than rolling your own!