Concurrent systems

Lecture 7: Crash recovery, lock-free programming, and transactional memory

Dr Robert N. M. Watson

1

Reminder from last 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

2

This time

- Transaction durability: crash recovery, logging
  - Write-ahead logging
  - Checkpoints
  - Recovery
- Advanced topics
  - Lock-free programming
  - Transactional memory
- A few notes on supervision exercises

3

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

The small print: we must keep in mind the limitations of **fail-stop**, even as we assume it. Failing hardware/software do weird stuff. Pay attention to hardware price differentiation.

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

5

Write-ahead logging

- **Log**: an 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>
- We build composite-operation atomicity from fundamental atomic unit: **single-sector write**.
  - Much like building high-level primitives over **LL**/**SC** or **CAS**!

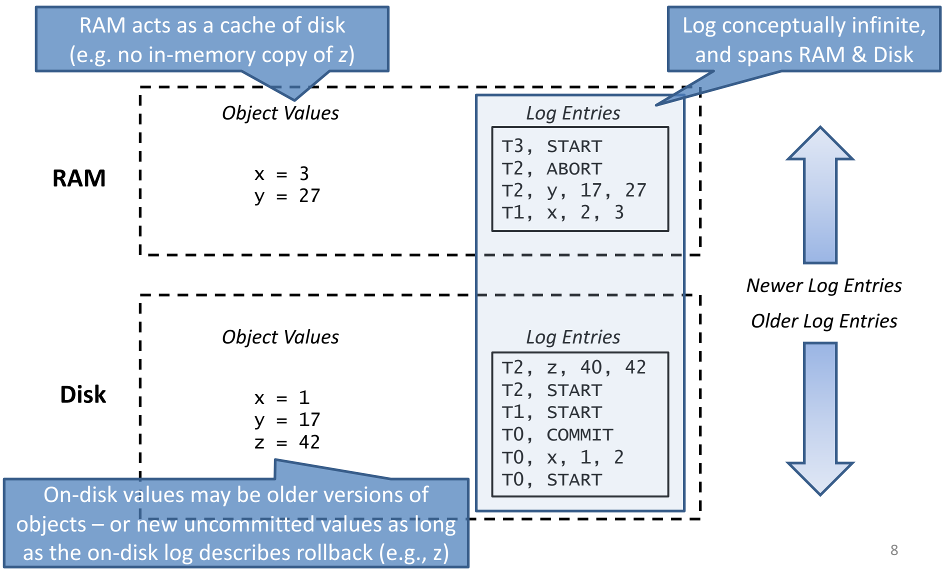
6

Using a write-ahead log

- When executing transactions, perform updates to objects in memory with **lazy write back**
  - I.e. the OS can delay disk writes to improve efficiency
- **Invariant**: write log records **before** corresponding data
- But when wish to **commit** a transaction, must first **synchronously** flush a commit record to the log
  - Assume there is a `fsync()` or `fsyncdata()` operation or similar which allows us to force data out to disk
  - Only report transaction committed when `fsync()` returns
- Can improve performance by delaying flush until we have a number of transaction to commit - **batching**
  - Hence at any point in time we have some prefix of the write-ahead log on disk, and the rest in memory

7

The Big Picture



8

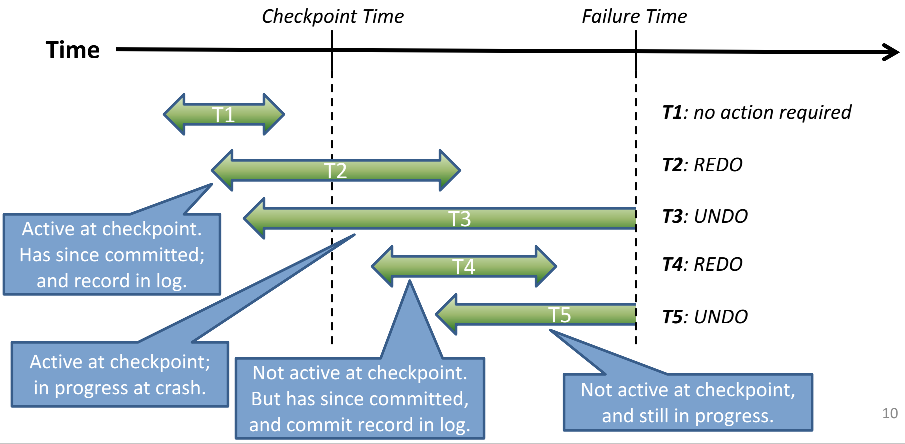
Checkpoints

- As described, log will get very long
  - And need to process every entry in log to recover
- Better to periodically write a **checkpoint**
  1. Flush all current in-memory log records to disk
  2. Write a special **checkpoint record** to log with a list of active transactions
     (pointers to earliest undo/redo log entries that must be searched during recovery)
  3. Flush all 'dirty' objects (i.e. ensure object values on disk are up to date)
  4. Flush location of new checkpoint record to disk
     (atomic single-sector write truncates unneeded log)
- (Not fatal if crash during final write)

9

Checkpoints and recovery

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



10

Recovery algorithm

- Initialize undo list **U** = { set of active txactions }
- Also have redo list **R**, initially empty
- Walk log forward as indicated by checkpoint record:
  - If see a **START record**, add transaction to **U**
  - If see a **COMMIT record**, move transaction from **U**->**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**
- After recovery, we have effectively checkpointed
  - On-disk store is consistent, so can **truncate** the log

The order in which we apply undo/redo records is important to properly handling cases where multiple transactions touch the same data

Write-ahead logging: assumptions

- What can go wrong writing commits to disk?
- Even if **sector writes are atomic**:
  - All affected objects may not fit in a single sector
  - Large objects may span multiple sectors
  - Trend towards copy-on-write, rather than journaled, FSes
  - Many of the problems seen with in-memory commit (ordering and atomicity) apply to disks as well!
- Contemporary disks may not be entirely honest about sector size and atomicity
  - E.g., unstable write caches to improve efficiency
  - E.g., larger or smaller sector sizes than advertised
  - E.g., non-atomicity when writing to mirrored disks
- These assumes **fail-stop** – not true for some media

12

Transactions: summary

- Standard mutual exclusion techniques not programmer friendly when 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!
- Appropriate only if operations are "transactional"
  - E.g., discrete events in time, as must commit to be visible
- Transactions used in databases and filesystems

13

Advanced Topics

- Will briefly look at two advanced topics
  - lock-free data structures, and
  - transactional memory
- Then, next time, on to a case study

14

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!
- Recall **TAS**, **CAS**, **LL/SC** from our first lecture: what if we used them to implement something other than locks?

15

Assumptions

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

```
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 – or **LL/SC** on RISC)
- Typically used to build spinlocks (or mutexes, or semaphores, or whatever...)

16

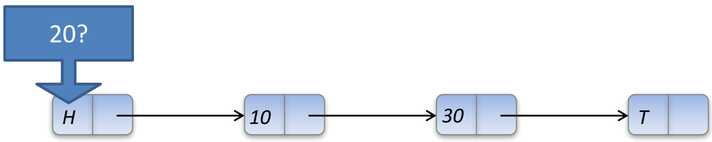
Lock-free approach

- Directly use **CAS** to update shared data
- For example, consider a lock-free linked list of integers
  - list is singly linked, and sorted
  - Use **CAS** to update pointers
  - Handle **CAS** failure cases (i.e., races)
- Represents the 'set' abstract data type, i.e.
  - find(int) -> bool
  - insert(int) -> bool
  - delete(int) -> bool
- Return values required as operations may fail, requiring retry (typically in a loop)
- Assumption: hardware supports atomic operations on pointer-size types

17

Searching a sorted list

- find(20):

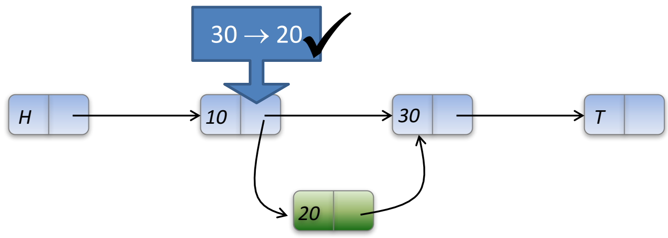


find(20) -> false

18

Inserting an item with CAS

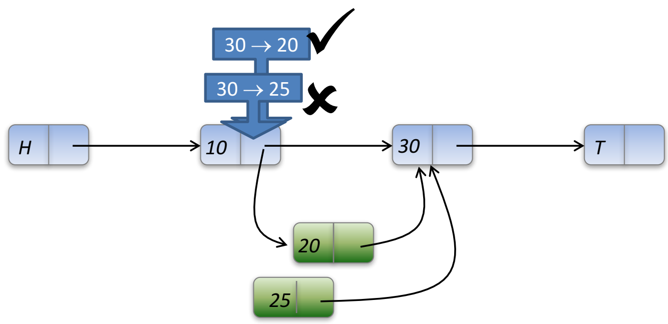
- insert(20):



insert(20) -> true

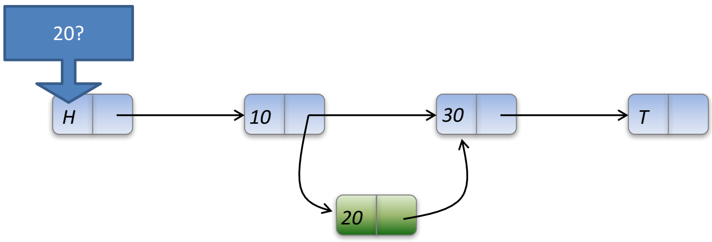
Inserting an item with CAS

- insert(20):
- insert(25):



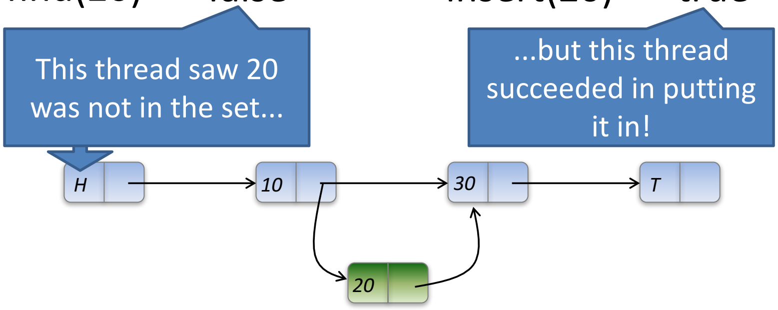
Concurrent find+insert

- find(20) -> false
- insert(20) -> true



Concurrent find+insert

- find(20) -> false
- insert(20) -> true



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

Linearisability

- 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) are consistent with some serial view: we call this a **linearisable 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!
- NB: On current hardware, synchronisation does more than just provide atomicity
  - Also provides ordering for memory visibility; on some hardware, "happens-before"; on others, .. not so much
  - Lock-free structures must take this into account as well

23

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));
```

24

TM advantages

- Simplicity:
  - Programmer just puts **atomic { }** around anything he/she wants to occur in isolation
- Composability:
  - Unlike locks, **atomic { }** blocks nest, e.g.:

```
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);
}
```

25

TM advantages

- Cannot deadlock:
  - No locks, so don't have to worry about locking order
  - (Though may get live lock if not careful)
- No races (mostly):
  - 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
- There is still a simplicity vs. performance tradeoff
  - Too much **atomic {}** and implementation can't find concurrency. Too little, and race conditions.

26

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 (take 1)~~
  - some hardware support emerging (take 2)
- But not a panacea
  - Contention management can get ugly
  - Difficulties with irrevocable actions / side effects (e.g. I/O)
  - Still working out exact semantics (type of atomicity, handling exceptions, signaling, ...)
- Recent x86 hardware has started to provide direct support for transactions; not widely used
  - … And promptly withdrawn in errata
  - Now back on the street again – but very new

27

Supervision questions + exercises

- Supervision questions
  - S1: Threads and synchronisation
    - Semaphores, priorities, and work distribution
  - S2: Transactions
    - ACID properties, 2PL, TSO, and OCC
  - Other C&DS topics also important, of course!
- Optional Java practical exercises
  - Java concurrency primitives and fundamentals
  - Threads, synchronisation, guarded blocks, producer-consumer, and data races

28

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!

29

Summary + next time

- Transactional durability: crash recovery and logging
  - Write-ahead logging; checkpoints; recovery
- Advanced topics
  - Lock-free programming
  - Transactional memory
- Notes on supervision exercises

- Next time:
  - Concurrent system case study the FreeBSD kernel
  - Brief history of kernel concurrency
  - Primitives and debugging tools
  - Applications to the network stack

- And then on to Distributed Systems!

30