Transactional Memory

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

The Problem with Locks

Problem

Write a procedure to transfer money from one bank account to another. To keep things simple, both accounts are held in memory: no interaction with databases is required. The procedure must operate correctly in a concurrent program, in which many threads may call transfer simultaneously. No thread should be able to observe a state in which the money has left one account, but not arrived in the other (or vice versa).

The Problem with Locks

Assume some infrastructure for accounts:

$$\textbf{type}\ \mathrm{Balance} = \mathrm{Int}$$
$$\textbf{type}\ \mathrm{Account} = \mathrm{MVar}\ \mathrm{Balance}$$

$$\mathit{withdraw} :: \mathrm{Account} \to \mathrm{Int} \to \mathrm{IO}\ ()$$
$$\mathit{withdraw}\ a\ m = \mathit{takeMVar}\ a \gg\!= (\mathit{putMVar}\ a\ \circ \mathit{subtract}\ m)$$

$$\mathit{deposit} :: \mathrm{Account} \to \mathrm{Int} \to \mathrm{IO}\ ()$$
$$\mathit{deposit}\ a\ m = \mathit{withdraw}\ a\ (-m)$$

Attempt #1

$$transfer\ f\ t\ m = \textbf{do}\ withdraw\ f\ m; deposit\ t\ m$$

Problem

The intermediate states where a transaction has only been partially completed are externally observable.

In a bank, we might want the invariant that at all points during the transfer, the total amount of money in the system remains constant. We should have no money go missing[a].

[a] We're not CBA

Attempt #2

$$
\begin{aligned}
&\mathit{transfer}\ f\ t\ m = \mathbf{do} \\
&\qquad \mathit{fb} \leftarrow \mathit{takeMVar}\ f \\
&\qquad \mathit{tb} \leftarrow \mathit{takeMVar}\ t \\
&\qquad \mathit{putMVar}\ t\ (\mathit{tb} + m) \\
&\qquad \mathit{putMVar}\ f\ (\mathit{fb} - m)
\end{aligned}
$$

Problem

We can have *deadlock* here, when two people transfer to each other simultaneously and both transfers proceed in lock-step.

Also, not being able to compose our existing *withdrawal* and *deposit* operations is unfortuitous from a software design perspective.

Solution

We should enforce a *global* ordering of locks.

$$\textbf{type}\ \mathrm{Account} = (\mathrm{MVar\ Balance}, \mathrm{AccountNo})$$

```
transfer (f, fa) (t, ta) m = do
    (fb, tb) ← if fa ≤ ta
      then do
        fb ← takeMVar f
        tb ← takeMVar t
        pure (fb, tb)
      else do
        tb ← takeMVar t
        fb ← takeMVar f
        pure (fb, tb)
    putMVar t (tb + m)
    putMVar f (fb − m)
```

It Gets Complicated

Problem

Now suppose that some accounts can be configured with a "backup" account that is withdrawn from if insufficient funds are available in the normal account.

Should you take the lock for the backup account?

To make life even harder: What if we want to *block* if insufficient funds are available?

Conclusion

Lock-based methods have their place, but from a software engineering perspective they're a nightmare.

- Remember not to take too many locks.
- Remember not to take too few locks.
- Remember what locks correspond to each piece of shared data.
- Remember not to take the locks in the wrong order.
- Remember to deal with locks when an error occurs.
- Remember to signal condition variables and release locks at the right time.

Most importantly, *modular programming* becomes impossible.

The Solution

Represent an account as a simple shared variable containing the balance.

$$\begin{aligned} &\mathit{transfer}\ f\ t\ m = \mathit{atomically}\ \$\ \mathbf{do} \\ &\qquad \mathit{withdraw}\ f\ m \\ &\qquad \mathit{deposit}\ t\ m \end{aligned}$$

Where *atomically* P guarantees:

Atomicity The effects of the action P become visible all at once.

Isolation The effects of action P is not affected by any other threads.

Problem

How can we implement *atomically*?

The Global Lock

We can adopt the solution of certain reptilian programming languages.

Problem

Atomicity is guaranteed, but what about *isolation*?

Also, performance is predictably garbage.

Ensuring Isolation

Rather than use regular shared variables, use special *transactional variables*.

$$
\begin{array}{lcl}
\mathit{createTVar} & :: & a \to \mathrm{STM}\ (\mathrm{TVar}\ a) \\
\mathit{readTVar} & :: & \mathrm{TVar}\ a \to \mathrm{STM}\ a \\
\mathit{writeTVar} & :: & \mathrm{TVar}\ a \to a \to \mathrm{STM}\ () \\
 & & \\
\mathit{atomically} & :: & \mathrm{STM}\ a \to \mathrm{IO}\ a
\end{array}
$$

The type constructor STM is also an instance of the *Monad* type class, and thus supports the same basic operations as IO.

$$
\begin{array}{lcl}
\mathit{pure} & :: & a \to \mathrm{STM}\ (\mathrm{TVar}\ a) \\
(\gg\!=) & :: & \mathrm{STM}\ a \to (a \to \mathrm{STM}\ b) \to \mathrm{STM}\ b
\end{array}
$$

Implementing Accounts

```
type Account = TVar Int

withdraw :: Account → Int → STM ()
withdraw a m = do
    balance ← readTVar m
    writeTVar a (balance − m)

deposit a m = withdraw a (−m)
```

Observe: *withdraw* (resp. *deposit*) can only be called inside an *atomically* ⇒ We have isolation.

But, we'd still like to run more than one transaction at once — one global lock isn't good enough.

Optimistic Execution

Each transaction (`atomically` block) is executed *optimistically*. This means they do not need to check that they are allowed to execute the transaction first (unlike, say, locks, which prefer a *pessimistic* model).

Implementation Strategy

Each transaction has an associated *log*, which contains:

- The values written to any TVars with *writeTVar*.
- The values read from any TVars with *readTVar*, consulting earlier log entries first.

First the log is *validated*, and, if validation succeeds, changes are *committed*. Validation and commit are *one atomic step*.

What can we do if validation fails? We re-run the transaction!

Liam: Demonstrate!

Re-running transactions

```
atomically $ do
  x ← readTVar xv
  y ← readTVar yv
  if x > y then launchMissiles else pure ()
```

To avoid serious international side-effects, the transaction must be *repeatable*. We can't change the world until *commit* time.

A real implementation is smart enough not to retry with exactly the same schedule.

Blocking and *retry*

Problem

We want to *block* if insufficient funds are available.

We can use the helpful action $retry :: \mathrm{STM}\ a$.

```
withdraw' :: Account → Int → STM ()
withdraw' a m = do
    balance ← readTVar m
    if m > 0 && m > balance then
        retry
    else
        writeTVar a (balance − m)
```

Choice and *orElse*

Problem

We want to transfer from a backup account if the first account has insufficient funds, and *block* if neither account has insufficient funds.

We can use the helpful action

$$\mathit{orElse} :: \mathrm{STM}\ a \to \mathrm{STM}\ a \to \mathrm{STM}\ a$$

$$\mathit{wdBackup} :: \mathrm{Account} \to \mathrm{Account} \to \mathrm{Int} \to \mathrm{STM}\ ()$$
$$\mathit{wdBackup}\ a_1\ a_2\ m = \mathit{orElse}\ (\mathit{withdraw}'\ a_1\ m)\ (\mathit{withdraw}'\ a_2\ m)$$

Evaluating STM

STM is *modular*. We can compose transactions out of smaller transactions. We can hide concurrency behind library boundaries without worrying about deadlock or global invariants.

Lock-free data structures and transactional memory based solutions work well if contention is low and under those circumstances scale better to higher process numbers than lock-based ones.

Most importantly, the resulting code is often simpler and more robust. **Profit!**

Progress

One transaction can force another to abort only when it commits.

At any time, at least one currently running transaction can successfully commit.

Traditional deadlock scenarios are impossible, as is cyclic restarting where two transactions constantly cancel each other.

Starvation is possible (when?), however uncommon in practice. So, we technically don't have eventual entry.

Performance

A concurrent channel using STM was implemented and compared to an MVar version. The STM version performs within 10% of the MVar version, and uses half of the heap space $\longrightarrow$ **Profit**! [1]

The implementation is a bit simpler as well. Let's do it if we have time! Just a linked list, really.

[1] Mostly, the MVar implementation performed poorly due to lots of overhead to make it exception-safe.

Database Guarantees

Atomicity ✓ Each transaction should be 'all or nothing'.

Consistency ✓ Each transaction in the future sees the effects of transactions in the past.

Isolation ✓ The transaction's effect on the state cannot be affected by other transactions.

Durability The transaction's effect on the state survives power outages and crashes.

STM gives you 75% of a database system. The Haskell package *acid-state* builds on STM to give you all four.

Hardware Transactional Memory

The latest round of Intel processors support *Hardware Transactional Memory* instructions:

`XBEGIN`	Begin a hardware transaction
`XEND`	End a hardware transaction
`XTEST`	Test if currently executing a transaction.
`XABORT`	Abort the transaction and jump to the abort handler.

The "log" we described earlier is stored in *L1 cache*. Speculative writes are limited to the amount of cache we have. If a speculative read overflows the cache, it may sometimes generate a *spurious conflicts* and cause the transaction to abort.

For this reason, progress can only be ensured through the *combination* of STM and HTM. Work is currently underway to implement this for Haskell, and prototypes show promising performance improvements.

That's it

We have now covered all the content in COMP3161/COMP9161. Thanks for sticking with the course.

- **Syntax Foundations**
  Concrete/Abstract Syntax, Ambiguity, HOAS, Binding, Variables, Substitution, λ-calculus
- **Semantics Foundations**
  Static Semantics, Dynamic Semantics (Small-Step/Big-Step), Abstract Machines, Environments, Stacks, Safety, Liveness, Type Safety (Progress and Preservation)
- **Features**
  - Algebraic Data Types, Recursive Types
  - Exceptions
  - Linear Types, Monads
  - Polymorphism, Type Inference, Unification
  - Overloading, Subtyping, Abstract Data Types
  - Concurrency, Critical Sections, STM

MyExperience

https://myexperience.unsw.edu.au

Further Learning

- UNSW courses:
  - COMP3141 — Software System Design and Implementation
  - COMP6721 — (In-)formal Methods
  - COMP3131 — Compilers
  - COMP4141 — Theory of Computation
  - COMP6752 — Modelling Concurrent Systems
  - COMP3151 — ¿Foundations of Concurrency?
  - COMP4161 — Advanced Topics in Verification
  - COMP3153 — Algorithmic Verification
- Online Learning
  - Oregon Programming Languages Summer School Lectures (https://www.cs.uoregon.edu/research/summerschool/archives.html) Videos are available from here! Also some on YouTube.
  - Bartosz Milewski's Lectures on Category Theory are on YouTube.
- Books — see my newly published Book List!

What's next?

The exam is on the **17th of November 2018** at 14:00 (i.e. 2pm).

- Consult exam timetable for location.
- It runs for 2 hours.
- It will be harmonically averaged with your class mark.
- You are allowed *two* **single-sided** sheets of **handwritten** notes.
- Should I postpone the revision lecture until Week 13 or even stuvac?

Week 13 tutes will be more revision questions that I have yet to write.

Bibliography

Simon Peyton Jones

Beautiful Concurrency

In "Beautiful Code", ed. Greg Wilson, O'Reilly, 2007

http://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf

Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy

Composable Memory Transactions

PPoP '05

Association for Computer Machinery

www.microsoft.com/en-us/research/wp-content/uploads/2005/01/2005-ppopp-composable.pdf

David Himmelstrup

Acid-State Library

https://github.com/acid-state/acid-state

Ryan Yates and Michael L. Scott

A Hybrid TM for Haskell

TRANSACT '14

Association for Computer Machinery

http://transact2014.cse.lehigh.edu/yates.pdf