Transactional Memory and Concurrency

Advanced Operating Systems (M)
Lecture 18
Concurrent, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory

- The processor/language memory models specify how concurrent access to shared memory works
  - Generally enforce synchronisation via explicit locks around *critical sections* (e.g. Java synchronized methods and statements; pthread mutexes)
  - Very limited guarantees about unlocked concurrent access to shared memory
Limitations of Locking

- Limitations of locks for managing concurrency:
  - Difficult to enforce locking
    - Users of shared data must acquire and release the locks
    - Encapsulating shared data in objects that manage the lock can help
  - Difficult to guarantee freedom from deadlocks
    - Usual solution: acquire and release locks in a fixed order
    - But, conflicts with encapsulation of locks within objects to enforce locking
  - Failures are silent
    - Race conditions due to incorrect locking generally only show under load
    - Extremely difficult to locate and debug
  - Balancing performance and correctness is difficult
    - Too many locks inhibit concurrency and reduce performance; too few lead to subtle bugs

- Implication: ensuring correct use of locks is difficult
Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from \( a_1 \) and move it to \( a_2 \), without exposing an intermediate state where the money is in neither account
  - Can’t be done without locking all other access to \( a_1 \) and \( a_2 \) while the transfer is in progress
  - The individual operations are correct, but the combined operation is not
- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object
Transactions for Managing Concurrency

• An alternative approach: use *atomic transactions* to manage concurrency

  • A program is a sequence of concurrent atomic actions
  • Atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads
  • The runtime must ensure actions have the usual ACID properties:
    • Atomicity – all changes to the data are performed, or none are
    • Consistency – data is in a consistent state when a transaction starts, and when it ends
    • Isolation – intermediate states of a transaction are invisible to other transactions
    • Durability – once committed, results of a transaction persist

• Advantages:
  • Transactions can be composed arbitrarily, without affecting correctness
  • Avoid deadlock due to incorrect locking, since there are no locks

```plaintext
atomic {
    a1.debit(v)
    a2.credit(v)
}
```
Transactional Memory Programming Model

• **Simple programming model:**
  - Blocks of code can be labelled \texttt{atomic} {...}
  - Run concurrently and atomically with respect to every other \texttt{atomic} {...} blocks – controls concurrency and ensures consistent data structures

• **Implemented via optimistic synchronisation**
  - A thread-local \textit{transaction log} is maintained, records every memory read and write made by the atomic block
  - When an atomic block completes, the log is \textit{validated} to check that it has seen a consistent view of memory
  - If validation succeeds, the transaction \textit{commits} its changes to memory; if not, the transaction is rolled-back and retried from scratch
Limitations of the Programming Model

- Transactions may be re-run automatically, if their transaction log fails to validate

- Places restrictions on transaction behaviour:
  - Transactions must be referentially transparent
    - They produce the same answer each time they’re executed
  - Transactions must do nothing irrevocable
    ```
    atomic {
      if (n > k) then launchMissiles();
      doMoreStuff;
    }
    ```
    - Might launch the missiles multiple times, if it gets re-run due to validation failure caused by `doMoreStuff`
    - Might accidentally launch the missiles if a concurrent thread modifies `n` or `k` while the transaction is running (this will cause a transaction failure, but too late to stop the launch)
  - These restrictions *must* be enforced, else we trade hard-to-find locking bugs for hard-to-find transaction bugs
Managing Communication and I/O

• Communication and I/O must be limited during a transaction
  • Pure functions can be executed normally
  • Functions that only perform memory actions can be executed normally, provided transaction log tracks the memory actions and validates them before the transaction commits
  • Functions that perform I/O are prohibited within a transaction

• Difficult to ensure through programmer discipline – needs language support
Implementations of Transactional Memory

- Transactions can be implemented in hardware or software
  - Need to track memory accesses, and potentially perform rollback
  - Can be done by a run-time support library, or using dedicated hardware
  - To date, have used software-based implementations; hardware-based implementations likely in future
    - e.g., Intel has announced support in their Haswell platform, due in 2013

- Difficulty enforcing transactions are side-effect free, so they can safely be rolled-back
  - Requires programming language (type-system) support
Controlling Side Effects

• Monads → well-defined way to control side-effects in functional languages

• A monad $M$ $a$ describes an action (i.e., a function) that, when executed, produces a result of type $a$; along with rules for chaining actions

• A common use is for controlling I/O operations:
  • The `putChar` function takes a character, and returns an I/O action that can display the character when performed
  • The `getChar` function is an I/O action; when performed it reads and returns a character
  • The `main` function is itself an I/O action, which wraps and performs the other actions

• The definition of the I/O monad type ensures that a function that is not passed an I/O action cannot perform I/O

• This is one part of the puzzle for transactional memory: define `atomic {...} to so that it doesn’t take an I/O action

```
putChar :: Char -> IO ()
getchar :: IO Char
```
Controlling Side Effects in Transactions

• How to track side-effecting memory actions?

• Use another monad \( STM \ a \) to wrap the transaction

• Manage side-effect via a \( TVar \) type
  
  • The \( newTVar \) function takes a value of type \( a \), returns a new \( TVar \) to hold the value, wrapped in an STM monad action
  
  • \( readTVar \) takes a \( TVar \) and returns an STM monad action; when performed this returns the value of that \( TVar \); \( writeTVar \) function takes a \( TVar \) and a value, and returns an STM action that assigns the value to the \( TVar \)

• Define \( atomic \ \{ \ldots \} \) to perform an STM transaction, and return an I/O action that performs the I/O and side effects that run the transaction

• The \( newTVar \), \( readTVar \), and \( writeTVar \) functions need an STM action, and so can only run in the context of an atomic block that provides such an action
Transactional Memory in Haskell

- Transactional memory is a good fit with Haskell
  - Pure functions and monads ensure transaction semantics are preserved
  - I/O and side-effects contained in STM action of an atomic {...} block
    - The TVar implementation is responsible for tracking side effects
    - The atomic {...} block validates, then commits the transaction (by returning an IO action to perform the transaction)
  - Untracked I/O or side-effects cannot be performed within an atomic {...} block, since there is no way to access an IO action directly
    - There is no IO action in scope, so code requiring one will not compile
    - Only way to access to an IO action is via the STM action passed to the atomic {...} block
    - A TVar requires an STM action, but these are only available in an atomic {...} block; hence can’t update a TVar outside a transaction (and hence can’t break atomicity guidelines)
STM Haskell Example: Resource Manager

- Implement a resource manager, granting access to integral chunks of the resource, enforcing access control between threads
  - `getR r n` should return `n` units of the resource `r` blocking until it is available
  - `putR r n` should return `n` units of the resource `r` to the available pool – implementation on the right

```haskell
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do {
  v <- readTVar r;
  writeTVar r (v + i)
}
```

- The use of the STM monad requires that the `putR` function be called from within an `atomic { ... }` block (this is enforced by the compiler)

```haskell
main = do {
  ...;
  atomic (putR r 3);
  ...;
}
```
Blocking Memory Transactions

- Transactions control access to resources, they do not provide synchronisation
  - Address by providing a `retry` operation for atomic blocks
    - e.g., consider the `getR` implementation on the right

- The `retry` function has type `STM a`, so must run within an STM action
  - That is, it must run within an atomic block

- Calling `retry` function aborts and restarts the current transaction, but blocks until one of its associated TVars has been modified
  - The system tracks access to the TVars to maintain the transaction log, so this is easily implemented

- The `retry` function is generally called when other concurrency approaches would block waiting for a signal

```haskell
getR :: Resource -> Int -> STM ()
getR r i = do {
  v <- readTVar r;
  if (v < i) then
    retry
  else
    writeTVar r (v - i)
}

retry :: STM a
```
Sequential Composition

• The entire set of operations surrounded in an atomic block appears to take place indivisibly
  • e.g., the operation on the right atomically gets 3 units of r1 then 7 of r2, the do notation provides for sequential composition of STM actions

| atomic (do {getR r1 3; getR r2 7}) |

• Note:
  • The type system ensures STM actions can only be executed in an atomic block
  • Actions accumulated over the entire atomic block execute or are rolled back when the transaction log for that block is validated
  • Either call to getR can invoke retry, causing the entire atomic block to be restarted
  • Any STM action can be robustly composed with other STM actions, and the resulting sequence of actions will still execute atomically
Composition of Alternatives

- May want to try one operation, and if that fails, try something else
  - Useful for error handling

- STM Haskell defines the orElse function which takes two STM actions, and returns one
  - Calling \( s_1 \ `\text{orElse}` \ s_2 \) first tries to run \( s_1 \); if \( s_1 \) calls retry then it’s abandoned without effect and \( s_2 \) is run instead
  - If \( s_2 \) also calls retry, then the entire action is restarted

- An alternative error handling method is throwing an exception
  - Throwing an exception causes the transaction abort and be validated;
  - If the transaction validates, the exception propagates; else the exception is caught and the transaction retried (the exception might be due to the inconsistency that caused validation to fail)
Transactional Memory in Other Languages

- STM Haskell is very powerful – but relies on the type system to ensure safe composition and retry

- Integration into mainstream languages is difficult
  - Most languages cannot require use of pure functions
  - Most languages cannot limit the use of I/O and side effects
  - Transaction memory can be used without these, but requires programmer discipline to ensure correctness – and has silent failure modes

- Unclear that the approach generalises to other languages
Discussion and Further Reading

- T. Harris, S. Marlow, S. Peyton Jones and M. Herlihy, “Composable Memory Transactions”, CACM, 51(8), August 2008. DOI:10.1145/1378704.1378725

- Is software transactional memory a realistic technique?
- Do its requirements for a purely functional language, with monadic I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?