Week 11

Atomicity and Software Transactions
Example

public final class StringBuffer {
    ...
    private int count;    // guarded by ‘‘this’’
    private char[] value; // guarded by ‘‘this’’

    public synchronized StringBuffer append(StringBuffer sb) {
        if (sb == null) { sb = null; }
        int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length) expandCapacity(newcount);

        // other threads may have changed sb.length(),
        // so len does not reflect the length of sb
        sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }

    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
    ...
}
Example

class Account { int balance; }

void transfer(Account from, Account to, int a) {
    from.balance -= a;
    to.balance += a;
}

int balance(Account a1, Account a2) {
    return a1.balance + a2.balance;
}

This program is improperly synchronized

Race condition on balance

A1
1500

A2
100

T1
transfer(A1,A2,50)

T2
balance(A1,A2)

150
Example

class Account { int balance; }

void transfer(Account from, Account to, int a) {
    synchronized(from) { from.balance -= a; }
    synchronized (to) { to.balance += a; }
}

int balance(Account a1, Account a2) {
    int t1,t2;
    synchronized(a1) { t1 = a1.balance; }
    synchronized(a2) { t2 = a2.balance; }
    return t1 + t2;
}

Program is properly synchronized.
No race conditions.
But, still incorrect.

transfer(A1,A2,50)

A1 190 A2 150

T1

T2

balance(A1,A2)
Traditional Approaches

Locks and monitors

- Enforce strict synchronization invariants
- Poor interaction with interrupts or other externally generated events
- Deadlocks possible
- Priority inversion
- Limited scalability

Solutions to these issues are often cumbersome or unwieldy
Atomicity

Serial (sequential) Execution vs. Serializable (atomic) Execution

Interleaving #1: \( z = x + y \)

VIOLATES SEQUENTIAL SEMANTICS!!

Interleaving #2: \( z \neq 7 \)

Interleaving #1: \( z = 7 \)
class Account { int balance; }
Account c, s;  // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
class Account { int balance; }
Account c, s; // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
class Account { int balance; }
Account c, s; // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}

synchronized int balance () {
    return c.balance + s.balance;
}
class Account {
    int balance;
}
Account c, s; // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
class Account { int balance; }
Account c, s; // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
class Account {
    int balance;
}
Account c, s;  // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
class Account {
    int balance;
}
Account c, s; // checking & savings accounts
synchronized void transfer (int amount) {
    c.balance -= amount;
    s.balance += amount;
}
synchronized int balance () {
    return c.balance + s.balance;
}
Transactional Computing

Transactions define a locus of computation that satisfy the following properties:

- Atomicity
- Consistency
- Isolation
- Durability

Model critical sections as “lightweight” transactions.

Key Question: How do we incorporate such notions into realistic programming systems?
Software Transactions

Instead of strict synchronization semantics induced by lock-based abstractions,

- Define a relaxed synchronization model:

Decouples shared access from synchronization machinery
Allow concurrent access to shared data provided serialization invariants are not violated.

- Separate specification of program correctness from implementation of a specific solution

Define a guarded region of code protected by a specific concurrency control protocol.
Ideally, applications should be able to overspecify the scope of these regions:

- The burden of how and when tasks can concurrently access shared data within these regions is shifted from the application to the implementation.
Threads and Transactions

- Single
- Multi
- Nested
- Nested & Multi
Approaches

Pessimistic

- Ensure no conflicts can occur
- Acquire locks to shared data prior to reads or writes
- Release locks when transaction commits

Optimistic

- Hope no conflicts occur
- Perform updates on local copies
- Commits perform consistency checking
- May abort if conflicts exist
Basic Actions

Start
  ▶ monitor access within the dynamic extent of a transaction region

Log
  ▶ Record updates performed by a transaction in case an abort occurs

Abort
  ▶ Restore global state and retry

Commit
  ▶ Check consistency invariants
Actions

Locking non-committed updates
  ▶ Enforce isolation by preventing other transactions from seeing intermediate values

Read and write barriers
  ▶ Trigger locking and logging actions

Continuation capture
  ▶ Allow aborted transactions to revert to a previous state and be re-executed
Example revisited

class Account { int balance; }

void transfer(Account from, Account to, int a) {
    atomic {
        from.balance -= a;
        to.balance += a;
    }
}

int balance(Account a1, Account a2) {
    int t1, t2;
    atomic {
        t1 = a1.balance;
        t2 = a2.balance;
    }
    return t1 + t2;
}

log contents reveal serializability violation
Strong Atomicity

Monitoring state changes may be necessary even outside atomic regions.

- **Strong Atomicity**

- **Non-repeatable reads**

- **Lost updates**

- **Dirty Reads**
Barriers

Dirty reads
  ▪ Read barriers that detect simultaneous writes by a transaction

Non-repeatable reads and lost updates
  ▪ Write barriers to prevent simultaneous access by a transaction

Barriers used to prevent non-transactional code from executing until an executing transaction commits.

We can leverage these techniques for other kinds of concurrency control.
STM Haskell

- An extension of Haskell with support for software transactions

- Addresses significant deficiencies in prior approaches:
  - safety in the presence of interaction with other threads
  - handling blocking operations
  - composition: explicit support for retry and non-blocking select
Motivation

- Consider a hash-table with thread-safe insert and delete operations
  - Remove an element A from table T1 and insert into table T2
  - Difficult to realize using locks
    - Hard to do using software transactions if the computation should block waiting for A to be visible in T1
    - Example: `Item get() { atomic (n_items > 0) { ... remove ... }}`
      - How do we atomically remove two items?

- Consider procedure p1 that waits for data on two pipes and p2 that does the same thing (on different pipes). How can we select between p1 and p2?
  - What is the semantics of: `atomic { p1 'orElse' p2 }`
Concurrent Haskell

- Extension to Haskell, a lazy purely functional programming language
- Support for explicitly-forked threads, and shared-memory communication
- Model communication effects using monads

**Intuition:** a value of type $M \, \epsilon$ is a computation which when run may perform $M$-actions before returning a value of type $\epsilon$.

- think of a monad as a design pattern that “amplifies” its underlying type, providing some additional computational or representational expressiveness
- alternatively, it represents a way of building workflows without exposing cross-cutting concerns
  - Strong encapsulation of its actions/effects
IO actions and mutable state

```haskell
putChar :: Char -> IO ()
getChar :: IO Char

main = do { c <- getChar; putChar c; putChar c }

newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()

forkIO :: IO a -> IO ThreadId

main = do { forkIO (print 'x'); print 'y' }
```
The STM Monad

-- The STM monad itself
data STM a
instance Monad STM
  -- Monads support "do" notation and sequencing

-- Exceptions
throw :: Exception -> STM a
catch :: STM a -> (Exception->STM a) -> STM a

-- Running STM computations
atomic :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional variables
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
Composable Transactions

Implement a resource manager that holds an integer-valued resource

- getR r n: acquire n units of resource r, blocking if r has an insufficient quantity
- putR r n: deposit n units of resource r

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

To commit an STM action:

```haskell
atomic :: STM a -> IO a
main = do { ...; atomic (putR r 3); ... }
```
STM Actions

Only STM actions and pure computations can be performed inside a memory transaction. IO operations cannot.

Conversely, no STM actions can be performed outside a transaction: e.g., can’t read or write a TVar unless it’s in the dynamic scope of an atomic.

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

Blocking and retry: retried transaction executes only when a read TVar is updated
Composition

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

- blocks if either r1 or r2 have insufficient resource
- no opportunity for deadlock
- caller oblivious to implementation of getR
- making getR an STM, rather than IO, action enables compositionality
Composing Alternatives

atomic (getR r1 3 ‘orElse’ getR r2 7)

nonBlockGetR :: Resource -> Int -> STM Bool
nonBlockGetR r i = do { getR r i ; return True }
    ‘orElse’ return False

blockGetR :: Resource -> Int -> STM ()
blockGetR r i =
    do { s <- nonBlockGetR r i;
         if s then return () else retry }

Laws

M1 ‘orElse’ (M2 ‘orElse’ M3)
    = (M1 ‘orElse’ M2) ‘orElse’ M3
retry ‘orElse’ M = M
M ‘orElse’ retry = M
MVars

```haskell

-- The type alias for MVars

-- The constructors for MVars

type MVar a = TVar (Maybe a)
newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing

-- The operations on MVars

takeMVar :: MVar a -> STM a
takeMVar mv
  = do { v <- readTVar mv
       ; case v of
         Nothing   -> retry
         Just val  -> do { writeTVar mv Nothing
                           ; return val } }

putMVar :: MVar a -> a -> STM ()
putMVar mv val
  = do { v <- readTVar mv
         ; case v of
           Nothing   -> writeTVar mv (Just val)
           Just val  -> retry }
```

communication channel with single buffered item
Multicast channels

data MChan a
data Port a
newMChan :: STM (MChan a)
    -- Write an item to the channel:
writeMChan :: MChan a -> a -> STM ()
    -- Create a new read port:
newPort :: MChan a -> STM (Port a)
    -- Read the next buffered item:
readPort :: Port a -> STM a

A buffer is represented as a linked list of items

type Chain a = TVar (Item a)
data Item a = Empty | Full a (Chain a)

An MChan points to the "write" end of a chain; a Port points to the "read" end

type MChan a = TVar (Chain a)
type Port a = TVar (Chain a)

newMChan = do { c <- newTVar Empty; newTVar c }
newPort mc = do { c <- readTVar mc; newTVar c }

readPort p
    = do { c <- readTVar p
        ; i <- readTVar c
        ; case i of
            Empty  -> retry
            Full v c' -> do { writeTVar p c';
                                return v } }

writeMChan mc v
    = do { c  <- readTVar mc
        ; c' <- newTVar Empty
        ; writeTVar c (Full v c')
        ; writeTVar mc c' }
Summary

- STM Haskell provides optimistic software transactions in Haskell
- Supports compositional construction of transactions in the presence of blocking and retry actions
- Monadic structure separates transactional effects (e.g., actions over logs) from pure computation